An Experimental Analysis of Iterated Spatial Joins in Main Memory

Benjamin Sowell[†*] , Marcos Vaz Salles[‡*], Tuan Cao[§*] , Alan Demers[¶], Johannes Gehrke[¶]

[†]Amiato, Inc. Palo Alto, CA

[‡]University of Copenhagen Copenhagen, Denmark

[§]Google, Inc. Mountain View, CA

[¶]Cornell University Ithaca, NY

ben@amiato.com, vmarcos@diku.dk, tuancao@google.com, {ademers, johannes}@cs.cornell.edu

ABSTRACT

Many modern applications rely on high-performance processing of spatial data. Examples include location-based services, games, virtual worlds, and scientific simulations such as molecular dynamics and behavioral simulations. These applications deal with large numbers of moving objects that continuously sense their environment, and their data access can often be abstracted as a *repeated spatial join*. Updates to object positions are interspersed with these join operations, and batched for performance. Even for the most demanding scenarios, the data involved in these joins fits comfortably in the main memory of a cluster of machines, and most applications run completely in main memory for performance reasons.

Choosing appropriate spatial join algorithms is challenging due to the large number of techniques in the literature. In this paper, we perform an extensive evaluation of repeated spatial join algorithms for distance (range) queries in main memory. Our study is unique in breadth when compared to previous work: We implement, tune, and compare ten distinct algorithms on several workloads drawn from the simulation and spatial indexing literature. We explore the design space of both index nested loops algorithms and specialized join algorithms, as well as the use of moving object indices that can be incrementally maintained. Surprisingly, we find that when queries and updates can be batched, repeatedly re-computing the join result from scratch outperforms using a moving object index in all but the most extreme cases. This suggests that — given the code complexity of index structures for moving objects — specialized join strategies over simple index structures, such as Synchronous Traversal over R-Trees, should be the methods of choice for the above applications.

1. INTRODUCTION

A wide variety of applications rely on high-performance spatial processing. These include location-based services [28], games [40], virtual worlds [13], and scientific simulations, such as molecular dynamics [36] and behavioral simulations [39]. In these applications, moving objects continuously explore and sense their environment. For example, pedestrians moving in a city may wish to find friends and businesses in their vicinity, and agents in games or behavioral simulations must query their surroundings in the virtual environment in order to decide what to do next.

*Work was performed while the author was at Cornell University

Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/13... $ 10.00.

Since many moving objects may issue similar spatial queries repeatedly, the *spatial join* is a key primitive in many of these applications. It is also common for applications such as simulations and games to batch updates in order to support evolving data more efficiently. In these applications, batches correspond naturally to the logical timesteps built into the application model. Thus we will consider applications that process repeated or *iterated* spatial joins, intermixed with batches of updates.

We have observed that all of the example applications above, as well as many others, can be run completely in main memory. Furthermore, several recent studies have argued that many moving-object applications, including more traditional location based services such as flight tracking, can tolerate some query staleness [9, 32]. However, these applications require very low response times even in the presence of significant query and update rates.

In this paper, we experimentally study techniques to efficiently process iterated spatial joins for moving object applications. Determining the most efficient techniques to process these spatial joins is challenging for a variety of reasons. First, there have been a tremendous number of join algorithms proposed in the literature, many of which have never been compared directly. A number of existing algorithms are based on an index nested loops join, and thus require that we select from a wide variety of spatial indices for the inner relation [9, 14, 21, 22, 29, 30, 37, 38]. Other approaches process the entire join in bulk, relying on lightweight auxiliary structures instead of a full-blown spatial index [3, 5, 17, 31].

Second, developers must choose a method to process updates. Since objects move through space, we can take advantage of techniques for indexing moving objects to avoid applying position updates [30, 37] or even update the join result itself incrementally [17]. Alternatively, since updates can be batched in many moving object applications, we can either rebuild a static index or reread the data for a bulk join method.

Third, many existing algorithms for spatial joins were optimized for disk resident data since their workloads traditionally did not fit in memory. As main memory sizes continue to grow, this constraint no longer holds, so in addition to all of the decisions above, developers of high-performance spatial applications must decide which methods make sense for a main memory environment. In particular, they must consider whether memory hierarchy optimizations developed for disks translate naturally into cache optimizations for main memory.

Although previous benchmarking studies provide some guidance to developers, they fall short of comprehensively addressing the breadth of options for moving object applications. Those that explicitly addressed main-memory execution have informed our choice of data structures, but they focused on a small number of static spatial indices rather than the full range of spatial join algorithms evaluated in this study [15, 20, 38]. Another recent study focused exclusively on moving object indices for non-predictive range and nearest neighbor queries [6]. However, that study only considered disk-resident data and only addressed monitoring scenarios in which there are a large number of moving objects but relatively few queriers. We consider main memory moving object applications in which a large fraction of the objects issue queries.

The novelty of our work lies in revisiting the vast literature on spatial joins in light of emerging workloads and commodity hardware. No previous experimental study has clarified the performance tradeoffs for the workloads we are interested in. We derive the non-obvious conclusion that if batching of queries and updates is permissible,then static methods that rebuild index structures from scratch outperform incremental methods in all but the most extreme cases. More specifically, we make the following contributions:

1. A Study With Many, Varied Algorithms: Our study compares ten representative techniques from the literature. We compare index nested loops joins using a variety of static and moving spatial indices as well as several special purpose spatial join algorithms. Some of these approaches have been evaluated independently in the literature, but to our knowledge no existing study has compared them all.

2. Experiments with Multiple Moving Object Workloads: We tune all of the algorithms for main memory, and evaluate them on uniform and skewed random workloads as well as two workloads motivated by high-performance spatial applications: a behavioral simulation of schooling fish [7] and a simple model of motion on road networks, which has previously been used to evaluate moving object indices [6]. We experiment with a wide range of parameters so that our results can be applied to many different scenarios.

3. A Benchmark with Open-Source, Extensible Code: To motivate further evaluation of future and existing iterated spatial join techniques, we have organized our benchmark as an extensible framework and made it available as infrastructure for the community [1]. Our APIs, code, and scripts give developers of novel algorithms easy access to an environment in which they can objectively test against previous work. Until now, such testing has required re-implementation of previous work in a common environment.

4. Integration of Parallelism and Predictive Queries: In addition to comprehensively evaluating alternative methods to perform iterated spatial joins on a single processor, we also experiment with the best methods on more complex predictive queries as well as with multi-core partitioned parallelism. These additional experiments suggest directions for future research on parallel efficiency and use of static methods for even more complex query types.

The remainder of the paper is organized as follows. Section 2 states requirements for iterated spatial processing techniques and shows our processing model. We describe the ten spatial join algorithms we evaluate in Section 3, and discuss our experimental setup and workloads in Section 4. Section 5 documents our tuning of the various algorithms, and Section 6 presents our experimental results. We discuss other spatial benchmarks in Section 7.

2. BACKGROUND

2.1 Moving Object Applications

Moving object applications process a large number of objects moving and interacting in some low-dimensional space. Unlike applications in which queries are posed by an external user, we focus on *agent-oriented* moving object applications in which objects issue *queries* to sense their environment and *updates* to change their positions and velocities. When taken collectively, the queries issued simultaneously by multiple objects are logically equivalent to a spatial join. While there have been many algorithms proposed to answer spatial joins over moving objects, emerging trends have led to important new requirements:

1. Bulk Operations: In many moving object applications, the actions of multiple objects occur simultaneously, e.g., the movement of pedestrians in a city center or the simulation of characters in a game. Thus, representing time in these applications is a challenging issue. A common approach, which we use in this study, is to discretize time, leveraging the observation that there is a granularity of either perception of simulated events or measurement of events in the real world. This allows the applications to achieve high throughput by processing both queries and updates in batches that correspond to logical timesteps. However, this also creates a tension in the choice of method. On one hand, we can use methods specifically designed for *moving objects*, such as the TPR-Tree [37]. On the other hand, if queries and updates can be processed in bulk, then it may be faster to re-execute a *static join* or *rebuild a static index* to deal with every batch of queries and updates [9, 40]. It is clear that rebuild approaches should outperform incremental updates when batch sizes approach the total number of objects. However, it is unclear which approach is best for the batch sizes found in practice.

2. Varying Query and Update Rates: Agent-oriented moving object applications support a wide range of query and update rates, and both the query and update rates are correlated with the number of objects in the application. For example, in a widely used model of animal movement, nearly every object issues a query and changes direction at every timestep, leading to a huge number of queries and updates [7]. This differs considerably from the traditional model of moving objects, which typically targets scenarios with a relatively small number of monitoring queries. A recent experimental study evaluated range queries on up to a million moving points, but issued batches of only 100 queries – four orders of magnitude below what we expect in many simulations [6]. At the other end of the spectrum, in certain location-based services only a small fraction of individuals or cars issue queries about their vicinity at any given time. We evaluate spatial join algorithms over this entire range of query and update rates to determine the best indices for each scenario.

3. Main Memory Execution: Moving object applications frequently must support either extremely high throughput (e.g., scientific simulations) or low response time (e.g., location- based services). To support these demanding requirements, many moving object applications are being executed entirely in main memory. Unfortunately, many spatial join methods have been optimized for data stored on disk, and it is not clear that the optimizations developed for disk-resident data will carry over when data is stored in-memory. As memory sizes become larger and computers become limited by memory bandwidth [23], it is becoming increasingly important to reevaluate spatial join techniques with these new metrics in mind. We conjecture that most moving object applications will move to main memory in the future. The raw representation of a billion moving objects fits comfortably under a hundred gigabytes – so even a service managing moving objects at planetary scale could be run in the main memory of a modest cluster.

2.2 Processing Model

As noted above, many moving object applications batch queries and updates, either for performance reasons, to mask latency and communication costs, or as a fundamental part of the model (as in some scientific simulations). We model these batches using atomic

Join Approach	Indexing Approach	
	Static	**Moving**
Index Nested Loops	R-Tree [14, 22] CR-Tree [21] Linearized KD-Trie [9, 29] Static Grid [38]	TPR-Tree [37] STRIPES [30]
Specialized	Plane Sweep [3] PBSM [31] Synchronous Traversal [5]	AE [17]

Table 1: Algorithms For Iterated Spatial Joins

timesteps called *ticks*. Each tick conceptually consists of a *query phase*, in which we process batches of object queries, followed by an *update phase*, in which we process batches of updates.

The **query phase** of a tick processes a batch of spatial queries issued by some fraction of the objects which read the state of other objects as of the previous tick. In this paper we will restrict ourselves to orthogonal range (box) queries, as these typically form the first and most expensive step of processing general range queries. Following previous studies [6, 15, 20, 38], we focus our attention on two-dimensional data, as this is sufficient to demonstrate the main conclusions of our study while controlling for its scope. For the same reason, we leave the study of other kinds of spatial queries, e.g., nearest-neighbor queries, for future work.

The **update phase** of a tick logically processes all updates in bulk before the start of the next tick. An *update* may change an object's velocity or position, and different join algorithms handle updates differently. Note that while ticks are atomic, we do not require that they correspond to a fixed unit of simulated (or real) time. The amount of computation in consecutive ticks may vary widely, allowing us to model many applications in the same framework.

3. SPATIAL JOIN TECHNIQUES

The spatial join algorithms evaluated in this paper can be divided into four categories based on whether they index *static* points at each tick or *moving* points across ticks and whether they use an *index nested loops* algorithm or a more *specialized* method to compute or maintain join results. Table 1 shows the ten algorithms we implemented according to this categorization. We consider each quadrant separately in the following subsections.

3.1 Static Index Nested Loops Join Methods

The simplest way to process iterated spatial joins is to build a static index on the points and then probe it repeatedly for each query. If most points are moving, it is more efficient to rebuild the index at every tick than to delete and insert every point that moves. Thus we need not support updates and can use efficient bulk-loading methods. We evaluate four different choices of static index: the R-Tree [14, 22], the CR-Tree [21], the Linearized KD-Trie [9, 29] and a Simple Static Grid [38].

R-Tree. The R-Tree hierarchically decomposes spatial objects so that each internal node corresponds to the minimum bounding rectangle (MBR) containing all of its children [14]. These MBRs are not guaranteed to be disjoint, and the performance of range search depends on the extent of their overlap. To reduce overlap and improve query performance, we use the bulk-loading technique developed by Leutenegger et al., which first sorts data objects according to their *x*-coordinate, and then divides the objects into roughly equal-sized stripes and sorts each stripe by *y*-coordinate [22].

CR-Tree. The CR-Tree is a cache-optimized variant of the R-Tree for use in main memory. It reduces the amount of space required to store each internal node so that more nodes can be packed into a single cache line and read from memory at once [21]. To accomplish this, the CR-Tree delta-encodes and quantizes keys, which may lead to false positives that must be filtered out during search. We use the same bulk loading algorithm for the CR-Tree as we did for the R-Tree.

Linearized KD-Trie. The Linearized KD-Trie maps the two-dimensional (or in general, *k*-dimensional) search problem to one dimension by using a space-filling curve such as the *z*-curve [9, 29]. In this approach, each point is represented as a bit-string, where each bit determines into which half of the space the point falls in one dimension, much in the same way that space is partitioned in a standard KD-Trie. Range probes, which are rectangles in two-dimensional space, become sequences of interval probes along the *z*-curve, which we can answer using binary search. Since the values are packed sequentially in an array, this structure can yield better cache behavior and main-memory performance than the standard pointer-based KD-Trie [9].

Simple Grid. Recent evidence suggests that a simple grid-based index performs well for orthogonal range queries [38]. This index partitions space uniformly into a fixed number of cells stored as a two-dimensional array. Each cell contains a pointer to a linked list of buckets storing the points that fall within that cell. The search algorithm must examine every cell that intersects the query region. As with the previous methods, we rebuild the index at every tick.

3.2 Static Specialized Join Methods

A number of specialized spatial join algorithms have been developed that go beyond traditional index nested loops methods with static indices. These algorithms often preprocess the queries and/or data points in much the same way that standard merge/hash joins do for relations. In this study we focus on two approaches: plane-sweep algorithms [3, 31] and synchronous traversal algorithms [5]. Plane sweep methods, including the more recent PBSM technique, can be thought of as extensions to the sort-merge join, while synchronous traversal algorithms utilize two static indices to speed up join processing. As in the previous case, these algorithms operate on static points and must be re-executed at every tick.

Plane-Sweep. Plane-Sweep (or *sweepline*) algorithms are standard and widely used algorithms for joining spatial data [8]. The basic idea is to sort the query rectangles and points together by the *x*-coordinate (or any other single coordinate), and sweep a line through this list in sorted order. Whenever this line crosses the left side of a query rectangle, that rectangle becomes *active* and is inserted into a *sweep structure*. Whenever it crosses the right side of a rectangle, the rectangle is removed from the sweep structure. This ensures that the *x*-coordinate of every point encountered during the search falls within all of the active rectangles. We output as join results those active rectangles that also contain the point in the *y*-dimension.

We implemented several alternatives proposed by Arge et al. for the sweep structure, including a simple linked list and a striped sweep structure partitions the space into vertical strips and maintains a separate linked list for each one [3]. We also implemented the *forward sweep* variant that does not use an explicit sweep structure. Instead, it maintains separate sorted lists for the points and query and whenever it encounters an object in one list, it searches forward in the other list and reports matching objects.

PBSM. The Partition Based Spatial-Merge Join (PBSM) of Patel and Dewitt is an extension to the standard plane-sweep algorithm for external memory databases [31]. We experiment with PBSM to determine whether its optimizations also improve cache performance in main memory. The algorithm starts by partitioning both the data points and the query rectangles into the same uniform grid, where the query rectangles are replicated in all of the partitions they intersect. Each grid cell is then joined using a standard plane sweep method. One problem with this approach is data skew. If the data is highly skewed, then some partitions may contain a large number of objects and may not fit into main memory (resp. cache). To address this, PBSM partitions the data into a much larger number of *tiles*, which are assigned to partitions using a round-robin hashing scheme. This effectively handles data skew, but it does introduce extra replication since query rectangles may intersect many tiles.

Synchronous Traversal. Synchronous traversal algorithms are a family of techniques that join two data sets by using R-Trees (or similar structures) to prune pairs of nodes that do not join. There are a number of variants, but we implement the version by Brinkhoff et al [5]. At the beginning of each tick, we build one R-Tree on the query rectangles and another on the data points. We then perform a depth-first traversal on the trees starting at the roots. At each pair of nodes, we compare MBRs of all the children of each node and visit each pair that intersects. Comparing the MBRs can be thought of as a join between sets of rectangles, and we evaluate three different strategies (Section 5). The first is a simple quadratic nested loops strategy that compares the MBRs for each pair of children. The second is an optimization that prunes those MBRs that do not intersect with the parent's MBR in the other tree. Finally, we can use forward plane-sweep to join the rectangles.

3.3 Moving Index Nested Loops Join Methods

Moving object indices maintain a set of moving points over time. Since they maintain the velocities of the points as well as their positions, they do not need to be rebuilt at every tick, though they do need to be updated when a point's velocity changes. Many moving object indices support sophisticated predictive queries [37], but our workload requires only *timeslice* queries that return the result of a range query at a specific time. With this functionality, a moving object index can be used in a nested-loops join in the same way as a static index, though it does not need to be rebuilt at every tick.In this study, we evaluate the TPR-Tree [37] and the STRIPES [30] index, which were found to perform well in a recent study [6].

TPR-Tree. The TPR-Tree extends the R-Tree to support moving objects by changing the MBRs to be functions of time [37]. A point is inserted with a reference position and a velocity, and the bounding rectangles grow over time so that they continue to enclose the same set of points. This causes overlap and may reduce query performance, as in a standard R-Tree. Timeslice queries in a TPR tree can be performed exactly as in an R-Tree where we evaluate the the MBRs as of the query time. Velocity updates are also similar to position updates in an R-Tree, though the splitting procedure is slightly different. Rather than optimizing for area, perimeter, or overlap, the TPR-Tree splitting algorithm attempts to optimize the integral of these quantities over time. Even if an update does not force a split, the authors suggest tightening the bounding rectangles at every update in order to improve query performance. There have been a number of extensions to the TPR-Tree, including the TPR*-Tree [33], but a recent experimental study found that the standard TPR-Tree performed better for simple timeslice queries [6].

STRIPES. STRIPES is a moving object index that is based on a technique called the *dual transformation* [30]. This transformation represents moving objects in two dimensions as a static points in four dimensions. Two dimensional range queries can be transformed into simplices in this same space. STRIPES is implemented using two PR quadtrees. The first has reference time 0 and stores objects updated between time 0 and L (where L is a parameter). The second quadtree has reference time L and stores those objects updated between time L and $2L$. After time $2L$, the first index is emptied and used to index points updated in the next L time units, and so on. Queries must be made to both quadtrees.

Interface 1: Iterated Spatial Join

```
IteratedSpatialJoin::startTick(vector<point>)
IteratedSpatialJoin::enumerationJoin(callback, vector<point>,
                                     regionGen
IteratedSpatialJoin::afterInsert(point)
IteratedSpatialJoin::beforeDelete(point)
IteratedSpatialJoin::beforeUpdateVelocity(point, newVelocity)
IteratedSpatialJoin::endTick()
```

3.4 Moving Specialized Join Methods

The final class of algorithms maintains queries as continuously moving rectangles. Since objects always query for objects within a fixed distance, the queries move continuously along with the objects, and can be indexed with a structure such as the TPR-Tree. These algorithms report join results incrementally by notifying a callback whenever a pair is added to or deleted from the results.

While a number of related approaches have been explored in the literature, including SINA [24], Conceptual Partitioning [25], AE and NE [17], in this paper we implement the AE algorithm [17].

AE. The *All Events (AE)* algorithm is an example of an *event-driven* continuous spatial join algorithm [17]. It precomputes the set of all *within events* in a fixed time window (the *event generation cycle*) corresponding to the times when a data point enters or exits a query rectangle. These events are stored in a priority queue ordered by time, and are used to update the join result. For example, when a point (resp. query) is inserted, it is joined with the existing queries (points) to produce new within events to be added to the queue.

Unlike Conceptual Partitioning, AE is designed for spatial joins and not only k-NN queries. Unlike SINA, it is straightforward to adapt the method to operate exclusively in main memory instead of having separate data structures and processing phases for disk. Unlike NE, AE batches event generation, a type of optimization we found often important in main memory. We believe AE to be a representative method for moving specialized join algorithms.

4. EXPERIMENTAL SETUP

4.1 Implementation

To ensure a fair comparison between the different join strategies, we built an experimental framework to automate the process of comparing methods and collecting experimental results. This framework is responsible for managing and updating the base data and issuing query and update calls to the join methods when necessary. We experiment exclusively with secondary indices, so the individual algorithms operate on pointers and never update the base data directly. This is reasonable for many spatial applications where each moving object may have multiple non-spatial attributes accessed independently by other portions of the application logic. We store the base data in an array and the update in the same way regardless of the secondary index employed. As such, our measurements include the time it takes to update the primary data, which should be roughly the same for all methods.

We implemented all of the join methods against a common API, shown in Interface 1. The model is explicitly time-stepped, and each join method is notified at the beginning (`startTick`) and end (`endTick`) of every tick. Some strategies, such as the static nested-loops algorithms, rebuild an index during the `startTick` method, while other algorithms that incrementally maintain an index may do nothing in this method after the first tick.

Bulk queries are posed using the `enumerationJoin` method. The method accepts a callback for reporting results, a set of query points, and a `RegionGen` instance, which contains a method to generate a query rectangle from a point. During experiments, the join results are discarded at the end of each tick, but they can also be saved and used for regression testing.

In order to keep indices that persist across multiple ticks up-to-date, the framework notifies each join strategy of all updates made to the moving objects at every tick. It does this by calling the `afterInsert`, `beforeDelete`, and `beforeUpdateVelocity` methods to indicate object insertion, deletion, and velocity updates, respectively. Note that the framework will only call the `beforeUpdateVelocity` method on moving algorithms. For static algorithms, it will convert these velocity updates to position updates, which are implemented with a delete followed by an insert.

Since the AE algorithm operates on continuous moving queries, it requires a slightly different API. The algorithm maintains continuous query results in a special callback object, which is stored as a hash table and contains both `addResult` and `deleteResult` methods. We omit the details of this extended API for brevity.

We optimized the individual join algorithms and tuned their parameters to perform well in main memory (Section 5). Though our implementations are all two-dimensional, we have made them as generic as possible with respect to key type. Since some of the main memory optimizations, such as quantization for the CR-Tree and Linearized KD-Trie, are limited to numeric types, all of our experiments use 64-bit floating point keys.

Our experimental framework and spatial join algorithms are implemented in approximately 30,000 lines of C++ which we make available for the community to use and extend [1].

4.2 Workloads

We experiment with four different workloads and a wide range of parameters drawn from behavioral simulations as well as more traditional moving object scenarios. We use a single binary trace format for all workloads to ensure repeatability. A trace consists of a set of initial points followed by a sequence of ticks, each of which contains a sequence of queries and updates. Queries are specified as a set of object identifiers and an *x* and *y* range for the query window around each object. For simplicity, we fix the query window size and vary the fraction of objects issuing the query at each tick. We select the queriers uniformly at random at each tick, except when experimenting with the AE algorithm. Since this algorithm maintains queries across ticks, it performs best when the same points query at each tick. Thus, to evaluate the AE Join in a favorable setting, we fix the set of queriers at the beginning of each experiment.

Updates are divided into four types: insert, delete, position update, and velocity update, and each update is represented as a type code, an object identifier, and the new object data to be inserted or updated. The framework is responsible for interpreting these updates and issuing the appropriate calls to the indices.

We group workloads into two categories: synthetic workloads used to test specific parameters, and workloads based on simulation models. We borrow several workloads from a recent benchmark by Chen et al., but we experiment with a wider range of query and update rates [6]. This ensures our results are comparable with previous work while still illuminating new parts of the parameter space.

Synthetic Workloads. To ensure that we can vary parameters independently, we experimented with two workloads consisting of randomly generated moving points. In the *uniform workload* of Chen et al. [6], points are instantiated uniformly at random positions in a fixed size square space and move in random directions. The speed of each point is chosen uniformly at random from a fixed set of possible speeds. We change the query workload so that at each tick, a fixed uniformly-chosen fraction of objects issue queries, and a fixed fraction of objects change velocity. To handle boundaries, we make the space a torus, so that objects that exit the space in the upper right corner reappear in the lower left corner with the same velocity. We accomplish this by inserting an explicit position update into the trace. Thus the total number of updates at each tick will slightly exceed the fraction of points issuing velocity updates.

Parameter	Setting			
	Uniform	Medium Uniform	Large Uniform	Gauss
Num. Ticks	100	100	500	120
Num. Points	10K .. 50K .. 90K	500K	1M	50K
Space Size	$10K^2$ .. $22K^2$ .. $30K^2$	$224K^2$	$577K^2$	$22K^2$
Max. Speed	200	2000	600	200
Query Size	400	4000	600	400
% Queriers	10% .. 50% .. 90%	50%	0.01%,1%	50%
% Updaters	10% .. 50% .. 90%	50%	0.01%,1%	N/A

Table 2: Parameter Settings for Synthetic Workloads

The *gaussian workload* models a skewed workload in which objects cluster around a fixed set of *hotspots* as in previous work [6]. The hotspots are initialized at uniform random locations, and points are distributed into rings around each hotspot according to a Gaussian distribution. The rings affect the speed and update frequency of the points – those points closer to a hotspot move faster and update more frequently. This ensures that the points maintain a skewed distribution over time [6]. Note that a point's velocity is updated at regular intervals depending on its ring. Thus if the points were to be inserted all at once at the beginning of the trace, as in the uniform workload, updates would clump together every few ticks. To spread out update load, we inserted the points over a period of 20 ticks. We do not include the time for these ticks in our results.

Table 2 shows the parameter settings we used for the uniform and gaussian workloads. We experimented primarily with between 10,000 and 90,000 moving points. This is quite small compared to the number of points that fit in main memory, but we believe it is a reasonable estimate for the number of objects that are processed on a single core for many spatial applications, since these applications often include a considerable amount of computation in addition to the spatial joins, limiting the number of objects that can be feasibly handled on a single core. To evaluate how the methods perform when the data set is larger than the cache size, we also employed two additional uniform workloads with "Medium" and "Large" numbers of points. The "Large Uniform" workload, in particular, contains one million points and has a much smaller fraction of queries and updates. As we scale the number of points, we also scale the size of the space so that the density remains constant for the uniform traces. This allows us to evaluate the scalability of the join methods without the number of joining points becoming quadratic and dominating the performance. Note that increasing the query window size generates the same effect of quadratic scaling on join result, and we thus only report one window size setting for each of our workloads in the interest of brevity.

We vary the fraction of points issuing queries or velocity updates per tick from 10%-90% of the total number of points. These are very high rates compared to many existing spatial database workloads, but as we noted in Section 2.1, applications such as simulations often have very high query and update rates, and we believe it is valuable to consider this part of the parameter space. In order to determine when moving object indices perform best, we have also conducted several experiments with lower query and update rates, between 0.01% and 1% of the points per tick. We execute this large trace for 500 ticks rather than 100, in order to better observe the benefits of incremental computation at low update rates.

Simulation Workloads. To model constrained motion in space, we use the network-based workload proposed by Chen et al. [6]. This

Parameter	Setting
Num. Nodes	6,105, 11,414, 175,343
Num. Edges	7,035, 15,641, 223,308
Num. Objects	100,000
Space Size	$100{,}000^2$

(a) Network Settings

Parameter	Setting
Num. fish	100K...900K
Num. goals	10
% informed	10%
α	25
ρ	160

(b) Fish Settings

Table 3: Simulation Workload Settings

workload models random motion on three real road networks from Oldenburg, Singapore, and San Francisco that were originally used by the Brinkhoff moving object generator [4]. Objects are placed randomly on network edges and move randomly over time. When an object reaches a network node, it chooses a random edge incident to that node and proceeds with a new speed. If an object overshoots the next node during a tick, its position is corrected using a special position update in the trace. This must be applied to both moving and static indices. As with the previous workloads, each object queries with a fixed probability at every tick. Table 3a shows the parameter settings for the Oldenburg, Singapore, and San Francisco networks, respectively. The networks are scaled to the same space size, so the edges in the larger networks will be shorter than those in the smaller networks, yielding more updates.

To evaluate our spatial join algorithms on a realistic application, we implemented a model of schooling fish proposed in Nature by Couzin et al [7]. In this model, fish move and interact according to two simple rules: (1) they avoid (turn away from) other fish within a small fixed distance α from their current position, and (2) if there are no fish within distance α, then they turn toward fish within a distance $\rho > \alpha$. Additionally, a certain number of "informed" fish have a desired goal that influences their direction of motion.

Rather than reading updates from a trace, we implement and execute the simulation itself using two spatial joins between fish corresponding to the steps described above. Note that the second join is performed only for those fish with no neighbors within distance α. As in our other experiments, we perform orthogonal range queries rather than proper distance queries, since answering the former is typically the first and most expensive part of processing the latter. Table 3b shows the settings we choose for the simulation.

4.3 Hardware

We ran all experiments on an Intel Xeon 5500 2.66 GHz with 48 GB RAM and four cores running Ubuntu Linux. Except for the multi-core experiment in Section 6.5, our code is run on a single thread and thus utilizes only one core. Each core has 32 KB of L1 cache and 256 KB L2 cache and they share 8 MB of L3 cache. To account for experimental overhead, we present build, query, and update times separately where appropriate. Unless otherwise noted, we ran each experiment on the synthetic data for 100 ticks with three randomly generated traces and report the average, min, and max values (the later two using error bars).

4.4 Parallel Execution

We also extend the framework to support parallel (multi-core) execution for static methods. We use a simple spatial partitioning method motivated by our distributed simulation platform [39]. While this method is unlikely to outperform special-purpose parallel join methods such as the TwinGrid index [32], it does not require modifying any of the join algorithms.

The parallel partitioning algorithm we use divides the points into disjoint rectangular regions at each timestep using a KD-Trie. Suppose we have n points and the desired degree of parallelism is p. (1) First we sample the data set and construct a partial KD-Trie on the base data with p leaves. This step is performed sequentially. For our experiments we use a sample size of 1000, which is sufficient for uniform data. (2) In parallel, thread i examines $1/p$th of the data set, and partitions the points according to the KD-Trie. It stores the resulting p partitions in shared memory. (3) Finally, in parallel, thread i takes the ith partition from each thread in the previous step and merges it into a list of points that will processed during the join. In order to ensure a fair distribution of points, we construct the KD-Trie with $r > p$ partitions and then assign multiple partitions to each thread. In our experiments $r = 4p$. Queries are partitioned in the same way using the same KD-Trie.

Method	CPI	Total Ops (billion)	LOAD Ops (billion)	LOAD Ops Breakdown				
				L1	LFB	L2	L3	Mem
R-Tree	2.2	2,353.1	973.7	93.4%	2.5%	0.5%	0.3%	1.0%
CR-Tree	1.4	2,732.9	1,133.5	93.8%	2.2%	1.7%	0.4%	1.0%

Table 5: Profiling: 50% queries and updates, 500K points

Note that if we execute the join independently on each partition, the result may be incorrect, as points near the boundary of a partition will not be joined with points from outside the partition. As in [39], we address this by replicating all points within distance q of the boundary of each region, where q is the maximum radius of a query rectangle that makes up the join. Note that we do not replicate the query points in order to avoid over-counting. Finally, since we do not migrate points between partitions, this method does not work with the moving or incremental join methods described in Sections 3.3 and 3.4.

In order to better understand the performance on a larger number of cores, we run our multi-core experiments on a machine with two 6-core Intel Xeon X5680 processors clocked at 3.3 Ghz. Each core has 32 KB L1 data cache and 256 KB L2 cache, and each CPU has 12 MB L3 cache. The system has 48 GB of RAM.

5. PARAMETER TUNING

In this section, we discuss tuning the parameters for all the spatial join methods we implemented. Due to space constraints, we present detailed results for our tuning of the R-Tree and CR-Tree only, since these results are surprising in light of previous work. For all remaining methods, we simply document the parameters used in subsequent experiments in the interest of repeatability and completeness.

R-Tree and CR-Tree. We organized each R-Tree and CR-Tree node as in [15] to avoid unnecessary pointer-dereferencing (and hence cache misses). Figure 4 shows the effect of increasing the node size on the performance of the R-Tree and the CR-Tree in which keys are quantized to 1, 2, or 4 bytes from their original 8 bytes. We found that both structures did best with a node size corresponding to a small multiple of the cache line size, though the decrease in performance due to larger node sizes is relatively minor. The CR-Tree was also relatively insensitive to the quantization level, and we use 2 bytes for all remaining experiments. Interestingly, the R-Tree and CR-Tree perform comparably in this experiment, with the R-Tree actually outperforming the CR-Tree by a small amount. This result differs from previous work, and we expect that it is because existing comparisons do not evaluate bulk-loaded R-Trees [15, 21]. The CR-Tree trades precision of MBRs during query processing, via quantization, for better packing of data. The hope is to increase cache efficiency and thus gain in performance. However, the improved precision of a bulk-loaded R-Tree during query processing may offset the gains obtained by the CR-Tree with data packing.

To better understand this effect, we profiled cache performance of the two methods using the Medium Uniform workload described

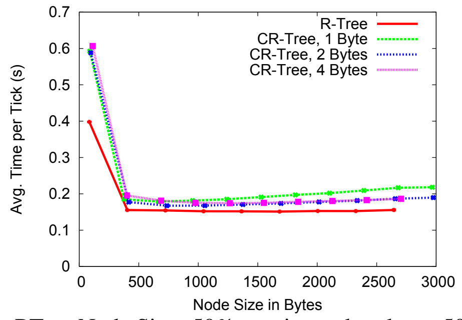


Figure 4: RTree Node Size: 50% queries and updates, 50K points

in Section 4.2. This dataset is larger than the last-level cache, and we used the Intel VTune Performance Analyzer to analyze the memory access behavior [16]. For each method, Table 5 shows the cycles-per-instruction count (CPI), the number of total instructions and load instructions, and a breakdown of the percentage of LOADs serviced from different levels of the memory hierarchy. Note that a request is serviced from a line fill buffer (LFB) if another request for the same data item is already outstanding.

From this table, we see that the R-Tree and CR-Tree display very similar cache performance. They both service over 95% of the LOADs from the L1 cache or the line fill buffers and only 1% of the accesses from main memory, suggesting that the aggressive compression used by the CR-Tree provides relatively little benefit when compared to a bulk-loaded R-Tree. Since the R-Tree does not have to filter false positives or perform quantization, it executes fewer total instructions. These results suggest that when bulk-loading is used, one should prefer the simpler R-Tree to the CR-Tree.

Remaining Methods. For completeness, we document below the parameters we found most effective for all remaining methods [1]. We found empirically that full partitioning of queries is best for the **Linearized KD-Trie**, while the **Simple Grid** performs best when configured with 13 cells per side with our standard uniform workload. To adjust for data size, we fix the average number of points per cell to 296, with four points per bucket, for other workloads assuming a uniform distribution. While these settings are much less granular than grid settings used by continuous spatial query approaches, such as Conceptual Partitioning [25], the computational overhead of static methods is significantly smaller than that of moving methods, a point we will explore in Section 6.

In accordance with the results of Arge et al. [3], we found striped-sweep to be the best choice for **Plane Sweep**, and 128 stripes to be a good setting. However, we employed forward sweep for **PBSM**, both because it is was used in the PBSM proposal and because striped-sweep effectively increases the number of partitions, complicating the evaluation [31]. We found 45 partitions per side to be optimal with our standard uniform workload, and to adjust for data size we fix the number of partitions so that each partition holds an average of 25 data points. We set the number of tiles to be equal to the number of partitions, since we found the effect of tiling to be modest even with skewed workloads. Since we found the optimal R-Tree node size to be quite small (8 entries), we adopted the nested loops algorithm to join individual R-Tree nodes in **Synchronous Traversal**, as this simple method is just as effective as more complex methods at such small node sizes.

We made several optimizations to the standard **TPR-Tree**. To improve update performance, we use the R*-Tree splitting algorithm and maintain a hash table from point id to tree node so that we do not have to perform an additional search for deletes. In contrast to static methods, we found per-node computation cost (e.g., the cost of maintaining the time-parameterized rectangles) to be considerable for the TPR-Tree, so that reducing the number of nodes continues to improve performance beyond what we would expect by caching effects. We found a good setting for node size to be 5,204B (68 entries). For high query and update rates, we found the lifetime parameter to have relatively little effect on performance, as the bounding rectangles are already compressed during updates. For experiments with the uniform and gaussian workloads, we choose a lifetime of 60 ticks, so that the tree will be rebuilt only once during a 100 tick experiment. The lifetime has a more significant impact at lower query and update rates. We found settings above 100 ticks to be best, with modest improvements even for larger settings. We choose a lifetime of 250 ticks when running experiments with the large trace.

Recall that **STRIPES** uses two quadtrees which index points updated between time 0 and L and time L and $2L$ respectively, where L is a *lifetime* parameter. We observed the lifetime of STRIPES to be best at the low setting of just two ticks for the uniform workload, and at the somewhat higher setting of 50 ticks for the large uniform workload. The corresponding parameter for the **AE Join** is the event generation cycle. As we expect the join to perform poorly for high query and update rates, we focused our optimization efforts on the large (1M point) trace. With 0.01% queries and 0.1% updates, the performance improved by over a factor of seven as we increased the event generation cycle from 2 to 22. The performance continued to improve modestly as we increased the event generation cycle further, but memory usage became a problem since the number of events stored in the queue grows with the event generation cycle length and the number of queries. We picked 20 ticks as the event generation cycle for the experiments, as this seemed to provide a good tradeoff between performance and space.

6. EXPERIMENTAL RESULTS

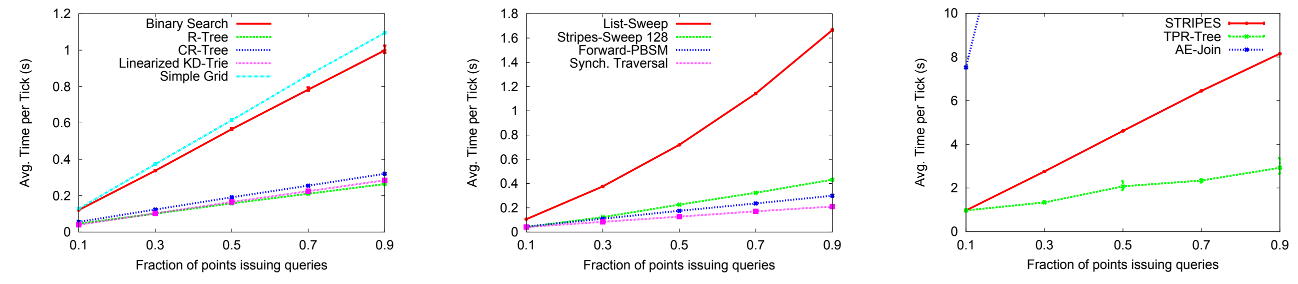
In this section, we analyze the effect of query and update rates and data skew on the performance of the various methods. We also measure the scalability of the methods and their effectiveness on several realistic workloads. In addition to the ten methods described in Section 3, we implemented a simple baseline algorithm which sorts the data points by one coordinate, and uses a nested loops algorithm with binary search to compute the join result.

6.1 Effect of Query and Update Rates

Scaling the Query Rate. Figure 6 shows the performance of the methods as we scale the fraction of points that issue queries from 10% to 90% for 50,000 points. The update rate is fixed at 50%.

Part (a) shows the performance of the five static indices, including the Binary Search method described above. Among these strategies, the R-Tree, CR-Tree, and Linearized KD-Trie perform best and are very similar, though the R-Tree is up to 20% faster than the CR-Tree. The Simple Grid is over a factor of four slower than the R-Tree when 90% of the points query, while Binary Search is slightly faster than the Simple Grid. We found this result to be stable, even though a recent study indicated grids not to perform significantly worse than trees in main memory [38]. However, in our study static structures are rebuilt at every tick instead of updated over time, and this changes the performance trade-offs, as discussed in more detail below.

Part (b) of Figure 6 shows the performance of specialized join methods. The slowest join, the standard Plane Sweep with a linked-list sweep structure, is 1.5 times slower than the Simple Grid at 90% query rate (note the different scales). However, the plane sweep methods that partition the space, PBSM and Striped Sweep, do much better, and SynchTraversal does best of all, outperforming List Sweep by a factor of 7.9 and the R-Tree by a factor of 1.25.



(a) Static Indices (b) Static Joins (c) Moving Indices

Figure 6: Scaling the Query Rate

Method	Build (s)	Query (s)	Update (s)
R-Tree	0.02	0.14	0.002
CR-Tree	0.02	0.17	0.002
Lin. KD-Trie	0.007	0.16	0.001
Simple Grid	0.002	0.61	0.003
List Sweep	0.008	0.71	0.001
Striped Sweep	0.008	0.22	0.001
Forward PBSM	0.008	0.16	0.001
Synch. Traversal	0.02	0.11	0.002
TPR-Tree	0.03	1.12	0.58
STRIPES	0.001	4.58	0.06

Table 7: Breakdown: 50% queries and updates, 50K points

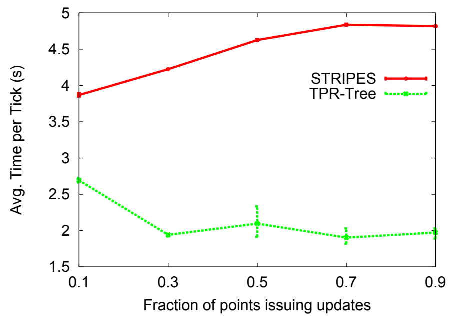


Figure 8: Scaling the Update Rate

Query Rate	Update Rate	Method	Build (s)	Query (s)	Update (s)	Total
0.0001	0.0001	R-Tree	0.57	0.0002	0.03	0.61
		SynchTraversal	0.60	0.0007	0.02	0.63
		TPR-Tree	0.15	0.12	0.05	0.32
		STRIPES	0.004	0.10	0.05	0.15
		AE-Join	1.12	0.0003	0.05	1.17
0.0001	0.01	R-Tree	0.57	0.0002	0.03	0.61
		SynchTraversal	0.60	0.0007	0.02	0.63
		TPR-Tree	0.15	0.12	0.44	0.71
		STRIPES	0.004	0.12	0.08	0.20
		AE-Join	1.15	0.0005	0.30	1.45
0.01	0.0001	R-Tree	0.57	0.02	0.03	0.63
		SynchTraversal	0.60	0.05	0.02	0.67
		TPR-Tree	0.15	11.39	0.05	11.59
		STRIPES	0.004	9.46	0.05	9.52
		AE-Join	2.02	0.07	0.12	2.21

Table 9: Breakdown: 1M points

Method	CPI	Total Ops (billion)	LOAD Ops (billion)	LOAD Ops Breakdown				
				L1	LFB	L2	L3	Mem
R-Tree	2.03	463.4	165.8	86.5%	4.0%	3.1%	3.0%	1.2%
Synch. Traversal	2.2	438.2	159.4	85.0%	4.8%	3.2%	4.0%	1.3%
TPR-Tree	0.88	558.9	211.7	95.1%	1.0%	0.07%	0.05%	0.06%
STRIPES	2.33	101.7	34.3	91.6%	3.8%	0.4%	0.2%	1.8%
AE-Join	0.69	2,037	945.3	99.3%	0.3%	0.1%	0.1%	0.04%

Table 10: Profiling: 0.01% queries and 0.01% updates, 1M points

Part (c) of Figure 6 shows the performance of moving methods. As expected, both STRIPES and the TPR-Tree scale linearly with the query rate. However, their performance is over an order of magnitude worse than that of the best static methods. We also show the performance of the AE-Join starting at 10% queries, but it scales poorly and already spends more than 20s per tick at 30%. For this reason we exclude this method for all experiments except those with very low query and update rates below.

To better understand where the time is spent, we further break down the performance into build, query, and update times. Table 7 shows this breakdown for all methods on 50,000 points with 50% queries and updates. Note that update times are still reported as non-zero for static methods, because we invoke the corresponding empty operations from our API. The query cost dominates for all of the indices, which suggests that we should prefer methods that are optimized for query performance, even at the cost of build and update time. As moving indices trade query for update performance, their total processing time deteriorates dramatically.

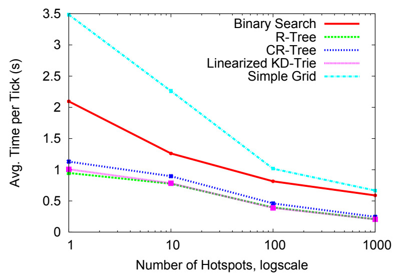
Scaling the Update Rate. We also vary the fraction of points that are updated at every tick. Figure 8 shows the performance of the moving object indices as we increase the update rate to 90% of the points per tick. The query rate is fixed at 50%. We do not report results for the static nested loops or static join methods since they are re-executed every tick and thus are insensitive to update rate.

Somewhat unintuitively, the TPR-Tree actually gets slightly faster as the update rate increases. The phenomenon was observed in a previous study, and is due to the time-parameterized bounding rectangles being retightened after each insert [6]. This increases the update time slightly, but is more than made up for by the decrease in overlap and the resulting increase in query performance. STRIPES is dominated by the TPR-Tree across the whole spectrum of update rates. At 90% updates, the gap between the methods is more than a factor of 2.5.

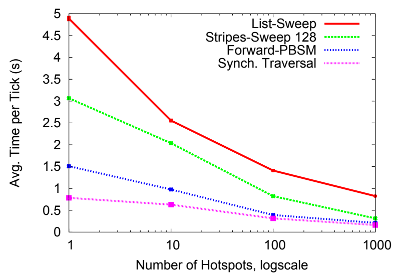
We also note that for most update rates, the query cost is still the dominant component in the runtime. At 90% updates, query cost for the TPR-Tree is 0.93s per tick, only slightly smaller than the update cost of 1.01s. For STRIPES, the query cost at the same point is 4.74s, significantly more than its update cost of 0.08s.

Even at their best performance, the moving object algorithms are still considerably worse than the static methods. At 90% update rate, the TPR-Tree is still an order of magnitude slower than Synchronous Traversal, for instance. In an attempt to find a scenario in which the moving indices outperform the static methods, we also experimented with much lower query and update rates. Intuitively, moving object indices should perform well with these settings, since it is necessary to rebuild the static index at every tick. To ensure that the build cost is non-trivial, we use the "Large Uniform" workload described in Section 4.2 with one million points.

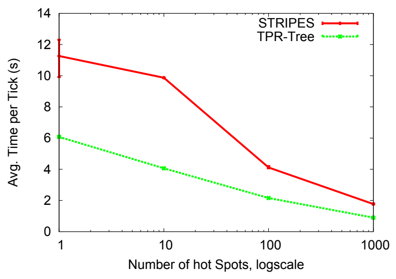
Table 9 shows the performance breakdown for the three moving methods and the best methods from the other quadrants. We experiment with query and update rates of 0.01% and 1% . We construct



(a) Static Indices

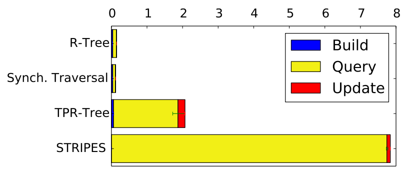


(b) Static Joins

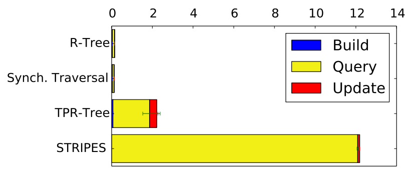


(c) Moving Indices

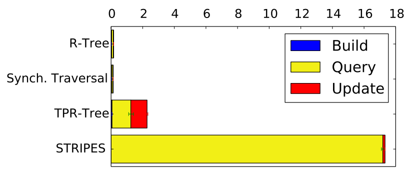
Figure 11: Scaling the Number of Hotspots



(a) Oldenburg



(b) Singapore



(c) San Francisco

Figure 12: Road Network Simulation

three extreme scenarios illustrating the effect of changing either the query or update rate between these values. In the base scenario in the first row of Table 9, only 100 agents issue queries and updates at every tick. As the table shows, STRIPES outperforms the R-Tree at this point by a factor of 4, and the TPR-Tree outperforms the R-Tree by a factor of 1.9. The AE-Join remains slower, however, as it spends most of its time enqueueing events.

Increasing the update rate by two orders of magnitude has a very minor impact on the performance of STRIPES, but the TPR-Tree is more than twice as slow. This is likely due to the cost of retightening the bounding rectangles. This should improve query time, but the query rate is so low at this point, that the effect is not visible. Increasing the query rate instead of the update rate has a dramatic impact on the TPR-Tree and STRIPES. Both slow down by more than an order of magnitude, while the static methods degrade by only a small amount. Interestingly, the AE-Join performs the best of the moving object indices, as the low update rate means that the event queue will have to be changed relatively infrequently.

We confirm the observations above by detailed profiling numbers for 0.01% queries and 0.01% updates (Table 10). Even with 1M points, the five methods shown exhibit good cache hit rates. Over 89% of LOADs are served from either the L1 cache or the line fill buffers. Out of the remaining accesses only a small fraction on the order of 2% or lower have to fetch data from main memory. A smaller percentage of LOADs are served from L1 for static methods than for moving methods. Moreover, the CPI values indicate that static methods spend roughly half of their time stalled. Even though instruction efficiency is better for moving methods, the latter use a larger number of instructions. The AE-Join executes roughly 4.6 times more instructions than Synchronous Traversal, for instance. Interestingly, STRIPES behaves more like a static method in this case, likely because the dual transformation converts moving queries to static queries in a quadtree.

6.2 Effect of Data Skew

To validate our experimental results on realistic workloads, we report on several experiments with the Gaussian as well as the network simulation workloads described in Section 4.2.

Gaussian Workload. To test how the spatial join algorithms handle skew, we vary the number of hotspots in the Gaussian workload. Figure 11 shows the results as we scale the number of hotspots from 1 to 1,000 using a log scale. Recall that the points cluster around each hotspot, so the workload with 1 hotspot is the most skewed. The workload with 1,000 hotspots is essentially uniform.

We observe from this set of graphs that all of the methods improve considerably as the skew decreases. The CR-Tree is almost five times faster with 1000 hotspots than with 1 hotspot, for instance, and the TPR-Tree is seven times faster. The relative ordering of the methods remains essentially the same as in the previous experiments. However, we do observe that for the static join methods (part (b)) the algorithms that partition the data in both dimensions (PBSM and SynchTraversal) improve by a factor of two or more relative to the Striped Sweep method which uses vertical stripes that only partition one dimension of the space.

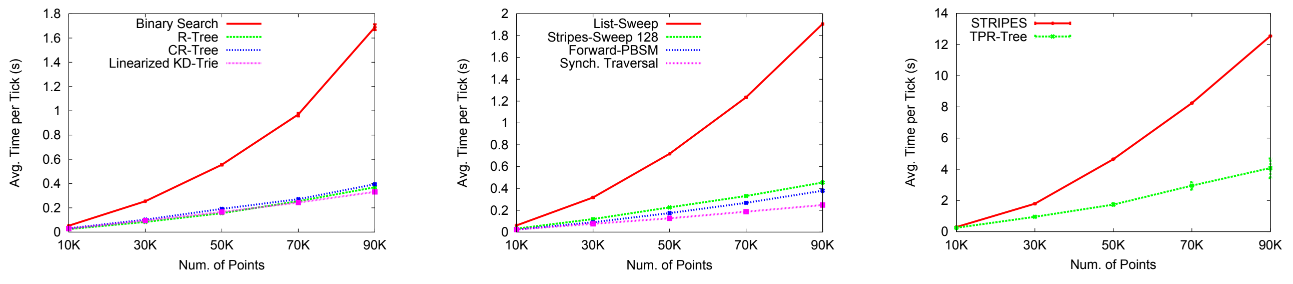
Network Simulation Workload. Figure 12 shows the results of one trial of the network simulation on road networks from Oldenburg (part (a)), Singapore (part (b)), and San Francisco (part (c)). Recall that we fix the size of the space and scale the networks so that they fill the available space. Since the networks increase in size from left to right, the individual road segments become shorter and the number of updates becomes larger.

STRIPES, as we saw previously, degrades when the number of updates is large, and takes 2.2 times longer on the San Francisco network than on the Oldenburg network. Once again, we see that this is almost entirely due to the query cost – STRIPES must search in two trees for those points that have been updated, so the query cost grows with the update rate. As expected, the TPR-Tree shows the opposite effect, with query cost decreasing under high update rates. While the cost of applying the updates increases, the total cost is still smaller for the San Francisco network when compared to Singapore. As observed previously, static methods comfortably dominate moving methods, with the synchronous traversal method slightly outperforming the R-Tree.

6.3 Scalability

In this section, we first evaluate the scalability of the various methods in the number of moving objects under a uniform random workload. We then present the scalability behavior of the best methods of each quadrant when integrated into a real simulation, which includes time not only for data access but also for computation of the simulation model.

Uniform Random Workload. Figure 13 shows how the indices perform as we increase the number of points from 10,000 to 90,000. Recall that we keep the point density and the query size fixed as we



(a) Static Indices (b) Static Joins (c) Moving Indices

Figure 13: Scaling the Number of Points

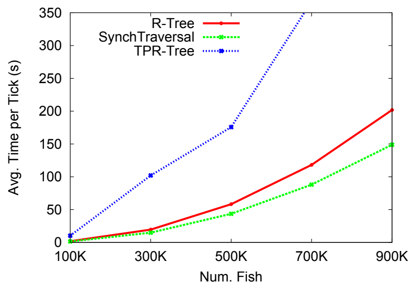
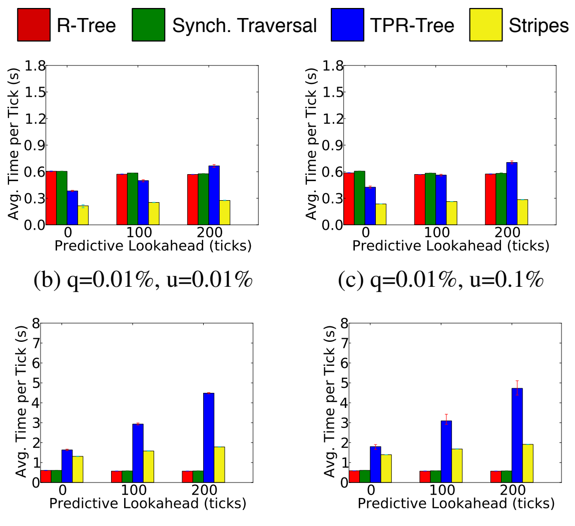


Figure 14: Scaling the Fish Simulation



(b) q=0.01%, u=0.01% (c) q=0.01%, u=0.1%

(d) q=0.1%, u=0.01% (e) q=0.1%, u=0.1%

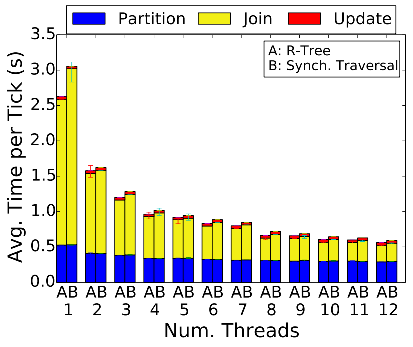


Figure 16: Multicore Scalability Results

scale, so the average number of points returned for a single query remains constant. As the number of queriers is a fixed percentage of the number of points, all strategies (except binary search) scale roughly linearly. The results confirm our previous observations about the relative ordering of the methods. The fastest static join, Synchronous Traversal, is roughly 1.5 times faster than the fastest static index and more than an order of magnitude faster than the moving strategies.

Fish Simulation. Figure 14 shows the performance of the fish simulation as we scale the number of fish from 100,000 to 900,000. We scale to a much larger number of objects in this example to demonstrate the impact of join method on large scenarios. Even when computation of the simulation model is introduced, the contribution of data processing method to performance is significant. The results we observe are similar to the synthetic experiments reported in Figure 13. Moving methods are dominated by static index nested loops joins, which in turn are dominated by static specialized joins. At 900,000 fish, the simulation with Synchronous Traversal runs by a factor of 1.5 faster than with the R-Tree. Note that due to the size of this experiment, we ran only a single trial.

6.4 Predictive Queries

In addition to current time queries, moving object indices like the TPR-Tree and STRIPES can be used to answer predictive queries about the state of the database in the future. The simplest form of predictive query is the timeslice query, which returns the result of a spatial query (or join) k ticks in the future assuming that all points continue to move with the same velocity. We can adapt the static join methods to support predictive queries by simply copying the data points, moving the copied points by k ticks, and computing the join over these updated points.

Figure 15 shows the impact of increasing the prediction time from 0 to 200 ticks at four different query and update rates with 1M points and 500 ticks. Note that the scale of the y-axis in the top row (0.01% query rate) differs from that in the bottom row (0.1% query rate). As expected, the static methods are insensitive to the predictive lookahead because they are rebuilt at every tick and the cost of copying and moving the points is negligible. The performance of the moving methods degrades as the prediction time increases, however. This is due to the larger boundary rectangles for the TPR-Tree and the larger query rectangles for STRIPES. This degradation is more dramatic for the TPR-Tree. With 1.0% queries and updates, the average time per tick is over three times larger for a predictive query at 200 ticks than for a current time query.

As in the current time case, STRIPES outperforms the static methods by up to a factor of two when the query rate is very low (0.01%). However, when the query rate increases to 1.0%, the static methods outperform both moving methods by more than an order of magnitude regardless of update rate. These results suggest that even for simple predictive queries, static methods work well for all but the lowest query rates.

6.5 Multi-Core Experiments

In this section, we evaluate the performance of the R-Tree and Synchronous Traversal methods on multiple cores using the space partitioning method described in Section 4.4. Due to limited space, we are only able to present one graph. Figure 16 shows the average time per tick as we vary the number of cores from 1 to 12 for the large uniform workload with 1 million points and a 1% query and update rate. We break each tick into three parts: the time to do the partitioning (shown in blue), the time to do the join (shown in yellow), and the time to update the points at the end of each tick (shown in red). The updates are performed linearly at the end of each timestep but have a negligible impact on overall performance.

As we increase the number of threads from 1 to 2, the performance improves considerably – by a factor of 1.66 for the R-Tree and a factor of 1.88 for Synchronous Traversal. However, we quickly hit the point of diminishing returns. For instance, the improvement going from 11 to 12 cores is only a factor of 1.07 for the R-Tree. Overall the R-Tree is 4.68 times faster on 12 cores than on 1, and Synchronous Traversal is 5.17 times faster.

We have identified several factors that contribute to this relatively disappointing speedup. First, since we must synchronize the all threads after each step of the partitioning algorithm and every phase of the join, the overall performance is limited to that of the slowest thread in both parts. This makes the algorithm particularly sensitive to the size of the partitions, and while we construct the KD-Trie with more leaves than cores in part to help balance the partition sizes, there will still be some variability unless the tree is perfectly balanced. Second, since we must replicate points on the boundaries of each partition, the number of duplicate points increases with the number of partitions. When we go from 2 cores to 12 cores, the number of duplicated points increases by a factor of 17.7 to 3.7% of the data set.

Overall, these results suggest that while it is possible to achieve some speedup using the simple partitioning method described above, the potential is limited due to the need to replicate boundary points and the difficulty of producing equal-size partitions using the KD-Trie method. We hope that these results provide a baseline for future work developing and evaluating more sophisticated parallel spatial join methods.

7. RELATED WORK

In this section we provide an overview of existing spatial benchmarks. We discussed specific algorithms in Section 3.

Spatial indexing has been an active area of research for decades, and there have been a large number of efforts to benchmark alternative algorithms. For indexing stationary points, the surveys by Gaede and Günther [11] and Ahn et al. [2] provide summaries of many of these results. More related to our own work are several studies that compared the performance of spatial indexing structures in main memory. Hwang et al. compared a number of main-memory R-Tree variants, though they considered only static data and did not use bulk loading algorithms [15]. More recent studies by Kalashnikov et al. and Šidlauskas et al. found that a simple uniform grid index was particularly effective in main memory and performed similarly to an optimized R-Tree [20, 38]. These results informed our choice of static indices.

There have been fewer experimental evaluations of spatial joins. A recent survey by Jacox and Samet provides an excellent overview of different approaches, but it does not include experimental results [18]. Nearly a decade earlier, Günther et al. evaluated several algorithms for joining static collections of rectangles, including a naïve nested-loops join and several index-based approaches using quadtrees [12]. They found that the index-based approaches considerably outperformed the naïve nested-loops join when the output size was much less than the full cross product, but they did not include the cost of building the indices or test any other algorithms.

Efforts to evaluate moving object indices have largely fallen into one of two categories: data generators and benchmarks for generating moving objects and queries, and actual experimental studies of the indices themselves. Early work in the first category, such as the GSTD and G-TERD generators, provided support for generating moving rectangles that move randomly with parameters drawn from a set of standard distributions [34, 35]. In order to generate more realistic patterns of motion, more recent generators such as the Brinkhoff generator [4] and BerlinMOD [10] provide high-level models of vehicular traffic.

There have been several experimental evaluations falling into the second category. Myllymaki and Kaufman performed some of the earliest experiments with their Dynamark and LOCUS benchmarks [26, 27, 28]. They focus on indexing point data and support simple (non-predictive) range and k-NN queries. These benchmarks were based on IBM's CitySimulator, which produced traces of people moving in a city. Unfortunately, CitySimulator is no longer available and these benchmarks are no longer usable. The COST Benchmark also defined a workload to evaluate several moving object indices, but it focused on scenarios in which the object locations are inaccurate [19].

The study most related to ours is a recent experimental paper by Chen et al. that compared six different moving object indices for non-predictive range and nearest neighbor queries [6]. They used several synthetic datasets and a network dataset based on the Brinkoff Generator [4]. While we use similar datasets, our work (i) considers memory-resident data, while their study assumed all data was stored on disk; (ii) uses a much higher number of queries per tick to reflect the fact that all objects in an application may be querying simultaneously; and (iii) rebuilds static indices each tick rather than just evaluating moving object indices.

Finally, several recent projects have exploited the benefits of bulk operations in settings where slightly stale results are acceptable. The MOVIES index periodically discards and rebuilds a static index rather than updating it in place [9]. The authors show that MOVIES outperformed existing index structures by more than an order-of-magnitude for a large distributed main-memory workload. Rather than rebuilding a single index, the TwinGrid index maintains separate indices for reads and writes and copies data between them using an efficient parallel memcpy [32]. Both structures may become out of date for short periods of time. Our work extends these results by considering a broader range of static and moving indices and join algorithms.

8. RECOMMENDATIONS

In this paper we presented a comprehensive experimental study of iterated spatial join techniques, with an emphasis on emerging workloads and commodity hardware.

We summarize our conclusions as a list of recommendations to developers of high-performance spatial applications:

1. Surprisingly, static methods outperform moving methods at all but the lowest query and update rates. Even though the former methods re-scan the whole data at every tick, their improved query performance amply compensates for this cost. STRIPES is very sensitive to the query rate, and the TPR-Tree is very sensitive to both the query and update rate. At extremely low query and update rates, STRIPES and the TPR-Tree do outperform the static methods, by at most a factor of $2-4\times$. Otherwise, the static methods often outperform moving methods by more than $10\times$.

2. For all but the lowest query and update rates, specialized join methods that prune results using all spatial dimensions outperform index nested loops methods as well as plane sweep methods that

partition at first in only one dimension. This result is the combination of improved query performance with very efficient bulk build procedures. Among the methods we evaluated, Synchronous Traversal was the top performer and significantly improved processing time of a real simulation of fish schools.
3. Our profiling results indicate that higher-order algorithmic effects still dominate performance in main memory, but there is room for improvement in the instruction efficiency of static methods.
4. Simple space partitioning provides limited utility when scaling to more than a small number of cores. Studying specialized parallel join methods remains an exciting area of future work.

To enable development of new methods and comparison with existing techniques, the experimental framework developed in this paper is freely available to the community at [1].

Acknowledgements In order to solicit feedback from the community, we assembled an advisory board of distinguished researchers to help guide our work. We would like to thank the members: Michael Gertz, Christian Jensen, Beng Chin Ooi, Jignesh Patel, and Bernhard Seeger for their feedback. This research has been supported by the National Science Foundation under Grants IIS-0911036 and IIS-1012593, by the Air Force Office of Scientific Research under Award FA9550-10-1-0202, the iAd Project funded by the Research Council of Norway, and by gifts from NEC and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] Project Website. `www.cs.cornell.edu/bigreddata/spatial-indexing/`.

[2] H. K. Ahn, N. Mamoulis, and H. M. Wong. A survey on multidimensional access methods. Technical Report UU-CS-2001-14, Department of Information and Computing Sciences, Utrecht University, 2001.

[3] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. VLDB*, pages 570–581, 1998.

[4] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. SIGMOD*, pages 237–246, 1993.

[6] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *Proc. VLDB*, pages 1574–1585, 2008.

[7] I. D. Couzin, J. Krause, N. R. Franks, and S. A. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Cheong. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[9] J. Dittrich, L. Blunschi, and M. A. Vaz Salles. MOVIES: indexing moving objects by shooting index images. *Geoinformatica*, 15(4):727–767, 2011.

[10] C. Düntgen, T. Behr, and R. H. Güting. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal*, 18(6):1335–1368, 2009.

[11] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[12] O. Günther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. Benchmarking spatial joins à la carte. In *Proc. SSDBM*, 1998.

[13] N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White. Scalability for virtual worlds. In *Proc. ICDE*, 2009.

[14] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. SIGMOD*, pages 47–57, 1984.

[15] S. Hwang, K. Kwon, S. Cha, and B. Lee. Performance evaluation of main-memory R-tree variants. *Advances in Spatial and Temporal Databases*, pages 10–27, 2003.

[16] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune.

[17] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of k-nn and spatial join queries on continuously moving points. *ACM Trans. Database Syst.*, 31(2):485–536, 2006.

[18] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1):7, 2007.

[19] C. Jensen, D. Tiešytė, and N. Tradišauskas. The cost benchmark—comparison and evaluation of spatio-temporal indexes. *Database Systems for Advanced Applications*, pages 125–140, 2006.

[20] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distrib. Parallel Databases*, 15(2):117–135, 2004.

[21] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. *SIGMOD Rec.*, 30(2):139–150, 2001.

[22] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. Technical report, Institute for Computer Applications in Science and Engineering (ICASE), 1997.

[23] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9:231–246, December 2000.

[24] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 623–634, New York, NY, USA, 2004. ACM.

[25] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 634–645, New York, NY, USA, 2005. ACM.

[26] J. Myllymaki and J. Kaufman. Locus: A testbed for dynamic spatial indexing. *IEEE Data Eng. Bull.*, 25:48–55, 2002.

[27] J. Myllymaki and J. Kaufman. Dynamark: A benchmark for dynamic spatial indexing. In *Proc. MDM*, pages 92–105, 2003.

[28] J. Myllymaki and J. Kaufman. High-performance spatial indexing for location-based services. In *Proc. WWW*, pages 112–117, 2003.

[29] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. PODS*, pages 181–190, 1984.

[30] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: an efficient index for predicted trajectories. In *Proc. SIGMOD*, pages 635–646, 2004.

[31] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. *SIGMOD Rec.*, 25(2):259–270, 1996.

[32] D. Šidlauskas, K. Ross, C. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *Advances in Spatial and Temporal Databases*, volume LNCS 6849, pages 186–204. Springer Berlin / Heidelberg, 2011.

[33] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pages 790–801, 2003.

[34] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *Proc. SSD*, pages 147–164, London, UK, 1999.

[35] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. On the generation of time-evolving regional data. *Geoinformatica*, 6(3):207–231, 2002.

[36] L. Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159(1):98, 1967.

[37] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. SIGMOD*, pages 331–342, 2000.

[38] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proc. GIS*, pages 236–245, 2009.

[39] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White. Behavioral simulations in mapreduce. *Proc. VLDB*, 3:952–963, September 2010.

[40] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proc. SIGMOD*, 2007.