Paper Number 5026

Using Constraints to Query R^*-Trees

Jonathan Goldstein **Raghu Ramakrishnan** **Uri Shaft**
Jie-Bing Yu
Department of Computer Sciences
University of Wisconsin, Madison
Madison, Wisconsin 53706
{(jgoldst, raghu, uri, jiebing) @ cs.wisc.edu}

February 22, 1996

Abstract

The R^*-Tree index is a popular multidimensional index used in several extensible and GIS-oriented database systems. In this paper, we show that a simple refinement of the search algorithm of the R^*-Tree—which is common to all variants of the R-Tree—offers significant speedups in most cases, *with little or no worst-case performance penalty.* The idea is essentially to use a conjunction of linear constraints (rather than a minimum bounding retangle) to approximate the query and to use this tighter bounding envelope to determine when the query overlaps with an R^*-Tree node. This raises an important question: How can we efficiently check whether the query envelope overlaps the minimum bounding box for a tree node? Linear Programming (LP) offers one solution, but it is susceptible to numeric approximation errors. One of the contributions of this paper is a new algorithm for performing this check check that is more efficient than LP and free from numeric errors. We also present several theoretical results characterizing this algorithm.

From a practical standpoint, adding the proposed *constraint query* refinement to existing R^*-Tree implementations is straightforward. Using implementations of R^*-Trees on top of the SHORE storage manager, we present experimental results (using TIGER census data for California and Wisconsin, and the Sequoia 2000 benchmark data set) that provide strong evidence in support of the proposed refinement. Our results demonstrate that the CPU overhead of our more complex overlap test, vis-a-vis the traditional minimum bounding box intersection test, is minor. On the other hand, the (CPU and I/O) gains can be considerable, especially for queries that are asymmetrically oriented with respect to the R^*-Tree axes. The results are especially surprising given that we make no changes at all to the insertion and deletion algorithms.

Finally, linear constraints are a very powerful tool for *formulating* queries, and a very important application of our results is that we can efficiently support a broad new class of such queries on multidimensional datasets drawn from a variety of domains that go well beyond GIS and spatial applications. We illustrate this, with experimental results, using queries over a five-dimensional projection of the widely used Compustat financial database (which contains over 300 dimensions!).

1 Introduction

A rapidly growing number of database applications require the ability to store and access large sets of multi-dimensional point, line and region data. Such applications include computer-aided design (CAD), geographic information systems (GIS), On-Line Analytic Processing (OLAP), and image repositories. A key component of a database management system that aims to support such applications is a *multidimensional access method*.

The R^*-Tree index [1] is a variant of the R-Tree [10], In this paper, we show that a simple refinement of the search algorithm of the R^*-Tree offers significant speedups in most cases, *with little or no worst-case performance penalty*. Since the search algorithm is essentially the same for all variants of R-Trees, our refinement is equally applicable to variants other than R^*-Trees; indeed, it applies to related structures such as k-d-B trees [24] as well. However, our discussion and performance study is limited to R^*-Trees.

Our main contribution is the proposal that we should use a conjunction of linear constraints, rather than the usual *minimum bounding box* (MBB), to approximate the query. This conjunction of constraints identifies a convex region that is always contained in the MBB. Search in the tree involves repeatedly checking whether the query envelope (conjunction of constraints or MBB) overlaps MBBs representing regions in the tree, as we go from the root to the leaves. Our approach requires a more expensive overlap test, but is likely to detect that the query does not overlap certain nodes that do overlap the MBB.

An important question is how to efficiently check whether the query envelope overlaps the MBB for a tree node. Linear Programming (LP) offers one solution, but it is susceptible to numeric approximation errors. In particular, it is possible that the LP test says that there is no overlap when indeed there is an overlap, leading to a failure to retrieve some overlapped objects! (This is possible, though unlikely.) A second contribution of this paper is a new algorithm to do this check that is more efficient than LP and safe with respect to numeric errors.

We present the results of several experiments designed to evaluate our approach, using an implementation of R^*-Trees on top of the SHORE storage manager [2]. A wide range of query types and system parameters are explored, the results provide strong evidence in support of the proposed refinement. We demonstrate that the approach is not limited to 2 and 3 dimensional datasets, as is common in GIS systems, but can deal effectively with additional dimensions, and is therefore valuable for general multidimensional data analysis.

We show that the CPU overhead of the constraint based overlap test (either LP or our new algorithm), vis-a-vis the traditional MBB intersection test, is usually minor. On the other hand, the (CPU and I/O) gains can be considerable, especially for queries that are asymmetrically oriented with respect to the R^*-Tree axes. Our test is in fact sufficiently inexpensive that we use it as a filter on *all* retrieved objects, including those found in leaf nodes. While testing objects in leaf nodes for overlap with the query does not reduce the page I/O done in R-Tree search, it greatly reduces the number of objects returned as candidate answers by the R-Tree search, and this can have a major impact on the cost of post-retrieval processing of the answers.

The improvements due to the constraint search algorithm (using either LP or our new overlap test) are especially surprising given that we make no changes at all to the insertion and deletion algorithms. Adding the proposed refinement to existing R^*-Tree implementations should therefore be very easy. [1] Of course, to the extent that insertion and deletion require searching, our idea can be used to refine insertion and deletion algorithms as well.

A very important application of our results is due to the observation that linear constraints can serve as a powerful query language over multidimensional point datasets. We can efficiently support a broad new class of such queries on multidimensional datasets drawn from a variety of domains that go well beyond GIS and spatial applications.

The paper is organized as follows. We present some necessary background material in Section 2 to keep

[1] Indeed, we found it straightforward to modify an existing R^*-Tree implementation to do so.

the paper self-contained. We describe our constraint based overlap test algorithms in Section 3 and their application to the R-Tree search refinement in Section 4. Details of the performance study are introduced in Section 5 and their results are presented and analyzed in two sections; Section 6 covers synthetic queries in which the query parameters were systematically varied, and Section 7 covers 'real' queries. In Section 8, we illustrate the use of linear constraints as a general query facility, with experimental results, using queries over a five-dimensional projection of the widely used Compustat financial database (which contains over 300 dimensions!).

2 Background

In this section, we present a brief introduction to R-Trees and the use of linear constraints to approximate a query.

2.1 The R-Tree Access Method

The R-Tree [10] is a height-balanced tree structure designed specifically for indexing multi-dimensional spatial objects. As in a B-Tree, each node (a disk page) in the R-Tree is either an internal node or a leaf node. Entries in internal nodes point to other nodes. Entries in leaf nodes contain the object IDs of the actual spatial objects (or, optionally, the objects themselves), as well as the *minimum bounding box* (MBB) of the object. Each internal node entry also contains an associated MBB; this is guaranteed to completely cover all MBBs in the subtree pointed to by the entry. Note that two entries in the same node might have associated MBBs that overlap. This means that unlike B-Tree searches, an R-Tree search may explore more than one path from the root to a leaf, even for point queries.

All MBBs are oriented orthogonally with respect to the (fixed) axes for the R-Tree. If the number of dimensions is greater than two, MBBs with an appropriate number of dimensions are used, but the basic idea remains the same.

The above description is sufficiently general that it applies to several variants of the R-Tree, including R^+-Trees [25] and R^*-Trees [1]. (Note that MBBs in non-leaf nodes of an R^+-Tree are non-overlapping, and therefore point queries search a single path from root to leaf. Nonetheless, region queries must explore several paths, in general.) These variations were motivated by the goals of improving space utilization, and therefore also search time (by reducing tree height and minimizing MBB overlaps); we will not discuss the variations further.

2.2 Linear Constraints As Queries

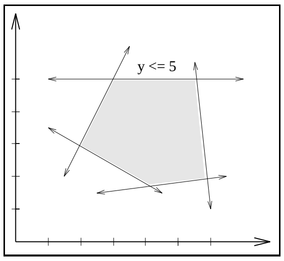

Figure 1: A Convex Polygon as a Conjunction of Constraints

Boxes that are orthogonal to the axes are the standard approximation for data objects and queries in R-Tree indexes. However, a box is often a poor approximation, especially for queries and data that are irregularly shaped and not suitably oriented with respect to the axes.

An alternative is to use a conjunction of linear constraints to describe the bounding envelope of data objects and queries. Storing conjunctions of linear constraints can be expensive in terms of space; we do not consider this approach. We do propose, however, that queries be approximated by such conjunctions; the only overhead is a more complex overlap test, which we demonstrate can be carried out very efficiently.

As an example of the use of linear constraints, a convex polygon can be represented as a *conjunction of two dimensional linear constraints.* Each constraint identifies a half-plane; a conjunction of constraints can therefore be used to identify any convex region, as illustrated in Figure 1. For instance, the following is a valid linear constraint:

$$x + y \leq 2$$

but not

$$x + y \leq 2, 5 \leq x \leq 7$$

since the above is actually the conjunction of the following three linear constraints:

$$x + y \leq 2 \qquad \text{and} \qquad x \geq 5 \qquad \text{and} \qquad x \leq 7$$

Note that the constraints did not have to form a closed region. For instance, the region defined by using just one of the constraints above is an entire half plane. Queries defined using these infinite sets are perfectly legal, and are, in fact, extremely useful for querying point data.

3 Linear Constraints and Box Intersection Algorithms

The results in this section address how we can check whether the areas specified by a conjunction of linear constraints and a box intersect. Linear programming (LP) can be used to solve this problem. The popular LP algorithms are "Simplex" and "Karmarkar's". For m constraints and n dimensions the complexity of Karmarkar's algorithm is $\mathcal{O}((m+n)^4)$ while the Simplex algorithm has an exponential run-time although its complexity seems to be $\mathcal{O}(m^2 n)$ in practice (for example, see [6]). In addition to the high complexity, these algorithms are prone to numerical error, and modifying them to be 'safe' would make them even more expensive.

Results from Computational Geometry all deal with representations of polyhedra as sets of vertices [22]. This representation for a hyper-box grows exponentially as the dimensionality increases, making the resulting algorithms hyper-exponential for our problem. Further, specifying high dimensional polyhedra in terms of vertices is not a natural way for users to pose queries, and converting from a conjunction of linear constraints representation—which *is* very natural—to a set of vertices representations is extremely expensive.

These problems with the known approaches motivate the results in this section.

3.1 Problem Representation and Some Notations

We consider algorithms for the following problem: Given (1) a region bounded by a conjunction of linear constraints, (2) a bounding box for this region that is orthogonal to the axes, and (3) a region bounded by a box that is orthogonal to the axes, do these regions overlap?

In more detail, the **input** to the algorithm consists of a set of m linear constraints in n dimensions, and two n dimensional hyper-boxes. The linear constraints are given by $\vec{A}_1, .., \vec{A}_m \in \mathbb{R}^n$ and $c_1, .., c_m \in \mathbb{R}$. Constraint P_i is defines as

$$P_i = \left\{ \vec{X} \in \mathbb{R}^n \,\middle|\, \vec{A}_i \cdot \vec{X} \geq c_i \right\}$$

We use the notation $\vec{A}_i \cdot \vec{X} = \sum_{j=1}^{n} A_{ij} X_j$. The query region for the problem is $P = \bigcap_{i=1}^{m} P_i$. The boxes in the input are given as

$$l_1, .., l_n, h_1, .., h_n \in Real \qquad \text{and} \qquad L_1, .., L_n, H_1, .., H_n \in \mathbb{R} \cup \{\infty, -\infty\}$$

$B = [l_1, h_1] \times \cdots \times [l_n, h_n]$ is the box we get from the R-tree. $B_P = [L_1, H_1] \times \cdots \times [L_n, H_n]$ is a box that bounds the region P. We do not insist on B_P being available or on it being a tight bounding box for P. If it is not available then we can use $B_P = \mathbb{R}^n$. Note that B can't have infinite bounds because it is a box that denotes a region in an R-tree. We may also use the bounding box of the root of the R-tree instead of $\mathbb{R}^n$ as a conservative estimate for B_P when B_P is not available and avoid infinite bounds for B_P as well.

Although we use the notation $\mathbb{R}$ for the set of values that a number in the input can have, the situation in practice is that $\mathbb{R}$ obviously can't be represented as input. We assume that the input numbers are IEEE double precision floating point numbers. Operations on such numbers yield an approximate result and thus numerical errors can occur.

The **output** of any algorithm for the overlap problem is either "Yes" or "No". The answer can contain two kinds of errors:

False positive: There is no intersection between the box and the linear constraints but the algorithm says there is.

False negative: There is an intersection between the box and the linear constraints but the algorithm says there is not.

In the context of selection queries a false positive may cause us to include results that don't fit the query. This requires post-processing and is a waste of resources so we want to avoid it as much as possible. A false negative however, may result in not retrieving answers to the query. An algorithm that does not allow a false negative answer for any input is called *safe*. The reason is that we may safely use such an algorithm knowing that the answer will be a superset of the correct answer. A safe algorithm that may yield a false positive answer is called *conservative*. The algorithm that we present is safe.

Finally, Algorithm $\mathcal{A}_1$ is said to be *less conservative* or *more accurate* than Algorithm $\mathcal{A}_2$ if both are safe and for any input on which $\mathcal{A}_1$ gives a false positive answer $\mathcal{A}_2$ also gives a false positive answer.

3.2 The Bounding Box Test

As an obvious first step, any overlap algorithm should calculate $B' = B \cap B_P$, and if $B' = \emptyset$ then answer "No', otherwise continue with B' instead of B as input. We call this the *Bounding Box* algorithm. Note that:

$$B' = [l'_1, h'_1] \times \cdots \times [l'_n, h'_n] = [\max\{l_1, L_1\}, \min\{h_1, H_1\}] \times \cdots \times [\max\{l_n, L_n\}, \min\{h_n, H_n\}]$$

$B' = \emptyset$ if and only if exists j s.t. $h'_j < l'_j$. It is interesting to observe that the $\min, \max$ operations and the $<, >, =$ comparisons done on IEEE standard floating point numbers have no numerical errors in them.

The Bounding Box test is very fast ($\mathcal{O}(n)$ complexity) and it is conservative. Currently, queries by linear constraints are dealt with (if at all) in database systems by assuming that B_P is given and using the Bounding Box test; the use of P is left for post-processing.

3.3 The Clipping Algorithm

We now present our main algorithm for checking overlap, called the Clipping algorithm. Our presentation is intended to be intuitive; the formal algorithm is listed in an appendix. We assume that the Bounding

Box test has been performed, and the input to the overlap algorithm is B', P, where $B' = B \cap B_P$. The intuition for the Clipping algorithm comes from the following result:
Proposition: For any $1 \leq i \leq m$ if $B' \subset B$ and $B \cap P_i \subset B'$ then $B \cap P = B' \cap P$.
Proof: We can represent B' as $(B' \cap P_i) \cup (B' \cap (\mathbb{R}^n - P_i))$. Since $(B \cap P_i) \cap (B' \cap (\mathbb{R}^n - P_i)) = \emptyset$ we have that $B \cap P_i \subseteq B' \cap P_i$. Therefore $(B \cap P_i) \cap P \subseteq (B' \cap P_i) \cap P$ and this is the same as $B \cap P \subseteq B' \cap P$. We also habe $B` \subseteq B$ so $B` \cap P \subseteq B \cap P$. We have inclusion in both directions so $B \cap P = B' \cap P$. ■

This result means that when can look at a specific constraint (say P_i) and find some $B' \subseteq B$ s.t. $B \cap P_i \subseteq B'$. For such a B', $B' \cap P = B \cap P$ holds, and so we can safely replace B with B'. If we also have $B' \neq B$ (i.e. $B' \subset B$) then we have safely "clipped" the query box into a smaller box.

The basic step of the Clipping algorithm consists of taking a constraint P_i and box B and finding the smallest possible B' s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$; the box B is then replaced by B'. By "smallest possible" we do not mean the actual smallest B', but the smallest B' we can find using safe numeric calculations. An *iteration* of the Clipping algorithm consists of applying this step to each constraint, considering the constraints in a round-robin order. After each such application, we check to see if the resulting B is empty or not. If it is empty then we can safely stop with a "No" answer. If it is not empty we continue.

The Clipping algorithm halts if there is no change in box B during an iteration. We also stop the algorithm after a predetermined number of iterations even if the last iteration has produced some change in B. An exact form of the algorithm is in Appendix A.

The only detail left is how to find B' for a specific P_i. First we look at "clipping" B along one dimension j. This means that we try to change the values of l_j, h_j while keeping the values of the other dimensions the same. There are three different cases to consider:

If $A_{ij} > 0$: Define a vector $\vec{z} = (z_1, .., z_n)$ as follows: If $A_{ik} > 0$ then $z_k = h_k$ otherwise $z_k = l_k$. Let $a \in \mathbb{R}$ be such that $A_{ij}a + \sum_{k \neq j} A_{ik} z_k = c_i$. We have that

$$a = h_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}$$

For every $\vec{x} \in B$ s.t. $x_j < a$ we have that $\vec{A}_i \cdot \vec{x} < c_i$ so $\vec{x} \notin P_i$. This means that B' will differ from B only in its low value for dimension j, which will be $\max\{l_j, a\}$. For safety purposes we make every possible numerical error in the calculation of a on the low side so the safe value of a may eventually be lower than the optimal value but not higher. If we deduce that $a > h_j$, then the resulting B' is $\emptyset$.

If $A_{ij} < 0$: Define the vector $\vec{z}$ to be the same as in the previous case. Also find the same value for a as in the previous case but using

$$a = l_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}$$

Note that for every $\vec{x} \in B$ s.t. $x_j > a$ we have that $\vec{A}_i \cdot \vec{x} < c_i$ so $\vec{x} \notin P_i$. This means that B' will differ from B only in its higher value for dimension j, which will be $\min\{h_j, a\}$. Note that this time we safely estimate a on the higher side to get a safe "clip". If we get $a < l_j$ then the result is $B' = \emptyset$.

If $A_{ij} = 0$: There is no way to clip B in dimension j using P_i.

Using these observations we can compute B' by going through the calculations and clipping B for each dimension. The exact details are in Appendix A.

Note that the only time when a clip of one dimension affects a clip of another dimension is when the result is $\emptyset$. The result for B' does not change when we change the order of dimensions in any way. The complexity of this method for the basic step of clipping a box with a constraint is $\mathcal{O}(n^2)$, and therefore the complexity of an iteration is $\mathcal{O}(mn^2)$. We observe that $\vec{z}$ remains the same when we deal with different dimensions, and the real cost is re-calculating $c_i - \vec{A}_i \cdot \vec{z}$ repeatedly. If we do this calculation once and use the result for each dimension, the cost for a basic clipping step is $\mathcal{O}(n)$, and the cost for an iteration is $\mathcal{O}(mn)$.

The reason we use a predetermined number of iterations as an additional stopping criterion is that the number of iterations is not bounded. The following input demonstrates the problem:

$$P \equiv \begin{cases} x_1 + x_2 & \geq \varepsilon \\ -x_1 - x_2 & \geq \varepsilon \\ -x_1 + x_2 & \geq \varepsilon \\ x_1 - x_2 & \geq \varepsilon \end{cases} \qquad B \equiv [-4, 4] \times [-4, 4]$$

For $\varepsilon > 0$ we have that $P = \emptyset$, and the number of iterations is at least $1/\varepsilon$. A similar example exists in 3 dimensions where we have $1/\varepsilon$ iterations and $P \neq \emptyset$. These examples show us that we can't afford to wait until the algorithm finishes all the iterations. If we stop the algorithm after as many iterations as we want we get a conservative algroithm.

We currently do not know if there is a situation for which the algorithm does not stop at all when I is infinite. An interesting observation is that the safe version of the algorithm does always stop but remains conservative. The reason is that after each iteration, B is still represented by IEEE standard floating point double precision numbers so the "clippings" occur in a discrete space, and thus the number of iterations is finite. This result does not help much in practice since the number of iterations can still be too large.

3.3.1 Some Properties of the Clipping Algorithm

Note that even if the algorithm does not stop, the result for B converges to some box, which we denote by B^*.

The main result about the Clipping algorithm is the following:

Theorem: The algorithm is safe and, in general, conservative.

That is, it sometimes yields false positives but never yields false negatives. The following theorem sheds additional light on the algorithm.

Theorem: For the theoretical algorithm (that uses absolute accuracy in the representation of "real" numbers) the following hold:

- For $n = 2$, if $P \neq \emptyset$ the algorithm reaches the correct result (i.e., no false positives) after one iteration. The complexity in this case is linear in the input.
- For $n \leq 3$, the algorithm converges in the limit to the correct answer (i.e. replacing B by B^* does not yield a false positive).
- For $n \leq 3$, if $B \cap P = \emptyset$ the algorithm stops even without imposing a predetermined bound on the number of iterations.
- For any $n \geq 2$ and any $I > 0$, we can always find some input on which the algorithm stops after exactly $I + 1$ iterations when the algorithm is not restricted to a predetermined number of iterations.
- For $n > 3$, there are cases when the algorithm yields a false positive.
- For $n > 3$ there are cases when the algorithm stops with a false positive even when the algorithm is not restricted to a predetermined number of iterations.

Proofs for these claims are in [7].

4 Improving Search in R-Trees

We now present our refinement to the R-Tree search algorithm. As noted in Section 2.1, each entry in an internal node of an R-Tree points to another node, and also contains an associated MBB that is guaranteed to completely cover all MBBs in the node (in fact, subtree) pointed to by the entry. Consider a typical *overlap* query, where a query shape is given and we are asked to retrieve all objects that overlap the query shape. The search begins at the root node. For each entry, we compare the MBB of that entry with the MBB for the query, and recursively search the subtree pointed to by the entry if there is an overlap. This is illustrated in Figure 2. Even if the query is a *containment* query (i.e., we wish to retrieve all objects contained in the query shape or all objects that contain the query shape), the search must be guided by overlap. Essentially the same search algorithm must be used; we therefore concentrate on overlap queries.

Note that two entries in the same node might have associated MBBs that overlap. This means that unlike B-Tree searches, an R-Tree search may explore more than one path from the root to a leaf, even for point queries. Consequently, search is potentially a much more expensive operation in an R-Tree than in a B-Tree.

As Figure 2 illustrates, the standard search algorithm may sometimes visit a node (Node 2 in the figure) because the bounding box for the node intersects the MBB for the query, even though the query itself does not intersect the node's MBB. This is the motivation for our refinement, which is essentially the following:

1. Use, as your query shape, a conjunction of a set of linear constraints P.
2. In order to determine whether a node must be visited, check the node's MBB against the set of constraints P (rather than the query MBB) for overlap.
3. (Optional:) For each object in a leaf node whose MBB overlaps the query bounding box, check if the MBB also overlaps P.

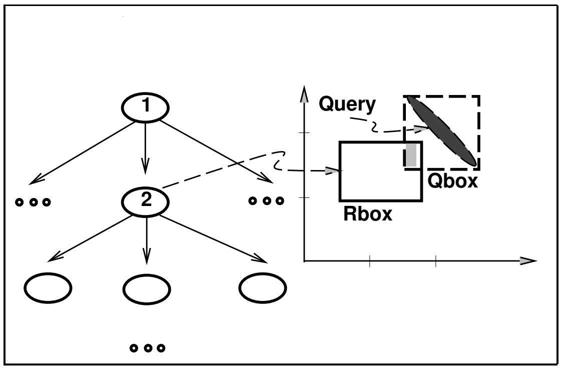

Figure 2: Search Path in an R-Tree

Clearly (as in the figure), there are cases where P will not overlap the node MBB even if the query MBB overlaps it. In such cases, we save the cost of exploring the subtree rooted at that node. The price paid for this optimization is the cost of a more expensive overlap test.

To appreciate the third step, which is optional, note that the R-Tree in general returns a set of *candidates*, rather than objects that overlap the query. This is a consequence of the way queries are approximated by a bounding envelope of some sort. Each candidate must subsequently be processed further to determine whether it really overlaps the query. The third step cannot reduce the number of page I/Os in the R-Tree search. We can simply use bounding box intersection to return candidate answers, like the standard search algorithm. However, applying this additional test can filter out several objects (which are guaranteed not to overlap the query), and thereby reduce the set of objects to be processed subsequently. We note that if this third step is included, the cost of LP versus our Clipping algorithm becomes significant for most queries, not just queries in which P has many constraints.

To summarize, our refinement yields two distinct benefits. First, it avoids some unnecessary I/O done by the standard search algorithm. Second, it reduces the set of objects in the answer set returned by the R-Tree. The potential gain of the refinement that we propose is considerable, as we demonstrate through a detailed performance study in the following three sections.

4.0.2 Reuse of Results using Clipping

Suppose that we run the Clipping algorithm on a box B and constraints P and get the answer "Yes" (i.e., there is potential for overlap). We denote the clipped box that is the final result of the clipping algorithm by $B_{clipped}$. B is a bounding box of some R-tree node N. Any entry in the R-tree that has B as an ancestor represents a region that is a subset of B. Any such region overlaps P iff it overlaps $B \cap P$. Since $B \cap P \subseteq B_{clipped}$, any such overlap must occur inside $B_{clipped}$. Therefore, we can use $B_{clipped}$ to screen all such entries, and only those that overlap $B_{clipped}$ should be tested against P. The easiest method of doing so is to use $B_{clipped}$ as a new value for B_P for those entries. This method reuses all the results achieved by earlier applications of the Clipping algorithm during the course of the search, and thus enhances its accuracy and effectiveness. (Recall that an application of the algorithm may terminate due to the predetermined bound on the number of iterations.)

5 Introduction to the Performance Study

In this section, we describe the search algorithms that we evaluated, the queries and datasets used, the parameters that we varied, and our performance metrics.

5.1 The Search Algorithms

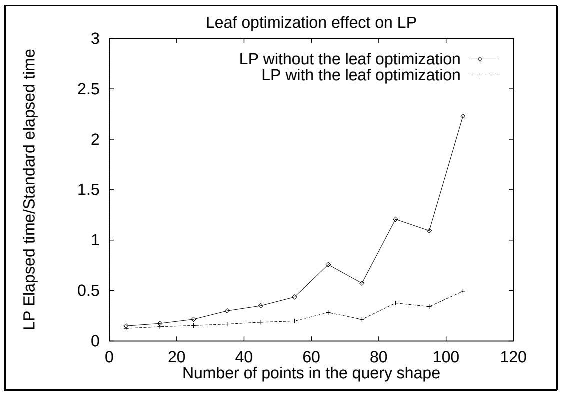

Figure 3: Effect of LP Overlap Tests in Leaf Nodes

We compared the standard R-Tree search algorithm, which uses MBB overlap tests, with our refined search algorithm presented in Section 4. We refer to the standard algorithm as STD in the rest of this paper. We evaluated two versions of the latter, one of which used LP and the other used our Clipping algorithm presented in Section 3. As for any overlap algorithm, we use the simple optimization of applying the Bounding Box test before using LP. The LP version did *not* apply the overlap test to the objects in leaf nodes. While such a test usually reduces the set of objects returned, and therefore reduces the post-processing time, the LP version performed poorly in most cases with respect to search time when applying overlap tests for leaf level objects. As demonstrated in Figure 3, even though the query shape used was a small percentage of the query MBB in this case, the cost of LP overlap tests becomes noticeable with queries having about 100 points; with overlap tests inside leaf nodes, performance degrades even more quickly. We therefore chose to present only the numbers for LP without overlap tests inside leaf nodes.

The version of our Clipping algorithm *did* apply the test at the leaf nodes as well to filter out objects that did not overlap the query polygon. The cost of the Clipping overlap test is sufficiently small that it does not significantly affect the performance of most queries, and we chose to do the tests inside the leaf nodes as well in order to measure the reduction in objects retrieved. Thus, it should be noted that our numbers on elapsed time are somewhat biased against the Clipping algorithm, since it does many more overlap tests than either the STD or the LP algorithm.

5.2 Queries and Datasets

In all experiments, polygon queries were of the type illustrated in Figure 4. Note that d controls the overlap between the query shape and the query MBB, while s controls the overall *selectivity* of the query, i.e., the fraction of the objects in the database that are retrieved by the STD algorithm. We chose this query shape because it is easy for us to systematically vary the size and shape of the query.

Each experiment involved varying one or two parameters. For each value of the varied parameters, a query of the shape described above was executed using each of the three candidate search algorithms (STD, LP and Clipping). For some experiments we use datasets from the "Sequoia 2000 Storage Landmark" [27]. These data contain thousands of landuse polygons that cover the region of California. For other experiments the underlying data set was TIGER Type I data from the U.S. Census Bureau for Orange County, California. The TIGER dataset contains topological data on spatial features. The Type I data contains mostly line segments. Queries were generated by randomly selecting an object from the database and using its lower left corner as the lower left corner of the query. The size and shape of the query depended upon the individual experiment.

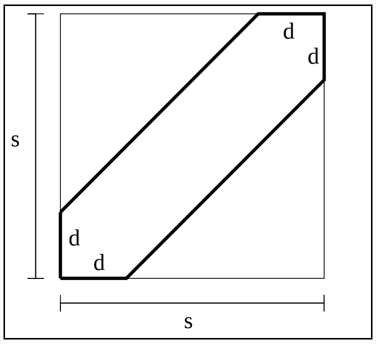

Figure 4: Query shape used in experiments

The measurements were then grouped together into several ranges (for the parameter varied along the X axis), each of which was verified to have at least ten points. The ratios of completion times, page I/O, and candidates retrieved for the traditional R-tree search algorithm versus the constraint based search algorithms were then computed. The actual points displayed in graphs represent the median points for the measured ratios in a particular range (for the parameter varied along the X axis).

All experiments in this section were run on an HP 715/33 with 32 MEG of memory. The page size was 1KB and the buffer pool contained 100 pages. The database disk buffers were cleared before each query.

5.3 Performance Metrics

We studied the following metrics for each of the constraint based search algorithms as a percentage of the value for the standard R-Tree search algorithm: *completion time*, *number of pages read*, and *number of objects returned by the query*. In general, we observed a strong correlation between the number of pages read and the completion time, indicating that queries are I/O bound, as expected.

6 Performance Evaluation of Synthetic Queries

In this section, we present the results of several experiments in which we systematically varied the following parameters: *selectivity* (the fraction of database objects returned by the standard R-Tree search algorithm), *query vs. query MBB overlap percentage*, *number of points in the query bounding polygon*, *degree of overlap in the data* and *page size*. [2]

We also varied the data set. First, the experiments were run on several data sets including the Sequoia 2000 benchmark data, and the TIGER data sets for (parts of) California and Wisconsin. There was no significant change in the results across any of these data sets. Second, several artificial data sets were generated in which the amount of overlap between objects in the database was varied. This is discussed in Section 6.4.

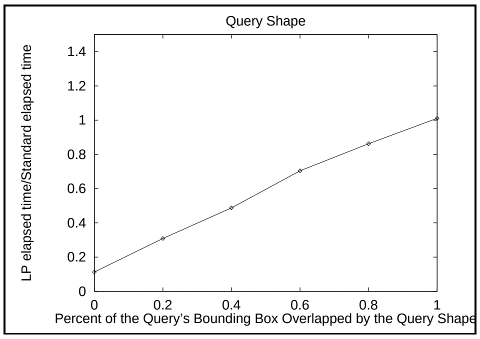

Figure 5: Varying Query Shape (LP)

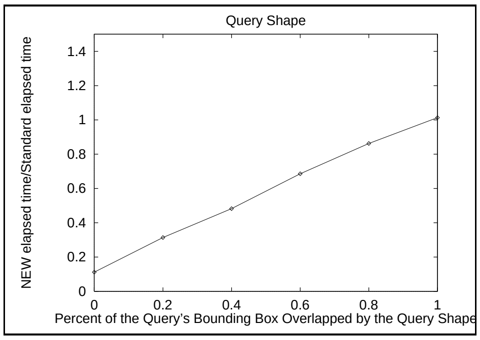

Figure 6: Varying Query Shape (Clipping)

6.1 Overlap of the Query and the Bounding Box of the Query

In this experiment, the query shape was varied to modify the overlap between the query and the MBB for the query. This experiment used the TIGER/Line data set mentioned earlier. Note that by modifying d in the query shape, the ratio (query shape area)/(area of query MBB) can be varied between 0 and 1. Geometrically, the query shapes for these two extremes are a diagonal line and a box respectively. Obviously, there is no advantage to using polygons to bound a query that is a box, and there is maximal advantage to doing so when the query is a diagonal line. While we varied the query shape, we kept the size of the MBB within a given range, so that (the standard algorithm) for each query retrieved five to ten percent of the objects in the database.

As Figure 5 indicates, the query to MBB overlap ratio roughly equals the improvement achieved by the constraint search algorithm; note that that at no point does the cost of the constraint algorithm exceed the cost of the standard search algorithm. This is because the overhead of polygon overlap tests is very small (even using LP) for queries in which the bounding polygon has only a few points. Figure 6 is similar to 5.

6.2 Selectivity of the Query

Next, we varied the selectivity of the query by varying the query shape variable s. The results are shown in Figure 7.

Figure 7 has multiple lines. Each line represents the effect of selectivity on a particular query shape. Small queries are similar in size to individual objects in the database, and all answers are likely to be clustered close together. As a result, there is less to be gained from a more accurate representation of such queries. This explains the similarity in performance between the different query shapes for small

[2] We also varied the number of buffers, but as expected this made no difference and so we do not report these numbers.

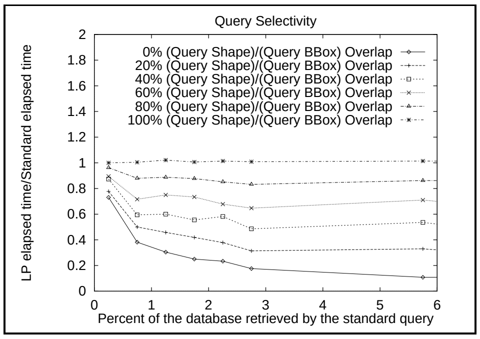

Figure 7: Varying Selectivity (LP)

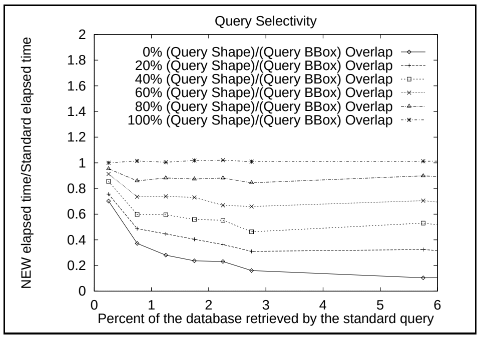

Figure 8: Varying Selectivity (Clipping)

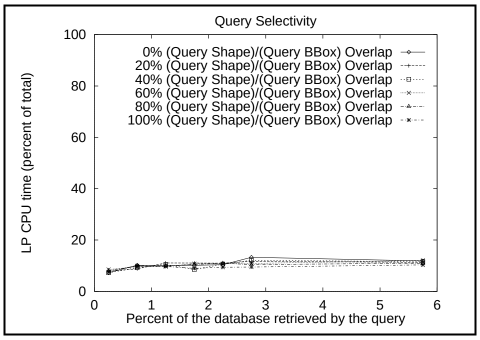

Figure 9: Varying Selectivity (LP) – CPU

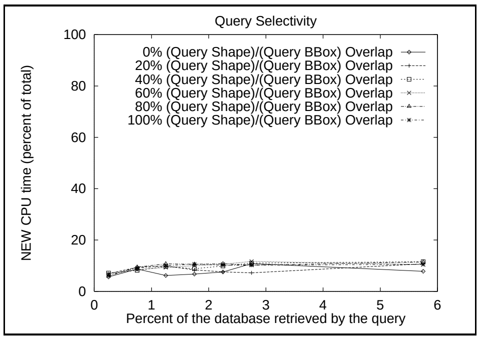

Figure 10: Varying Selectivity (Clipping) – CPU

selectivities as well as the query shapes' subsequent divergence. The worst case for the constraint algorithm is obviously when the query is itself a box (100% overlap); note that even here, the constraint algorithm costs essentially the same as the standard algorithm regardless of the query shape. Figure 8 is similar to Figure 7.

Observe in Figures 9 and 10 the low CPU usage for both LP and Clipping algorithms. This behavior was observed in all experiments except in Section 6.3 , where CPU graphs will be provided.

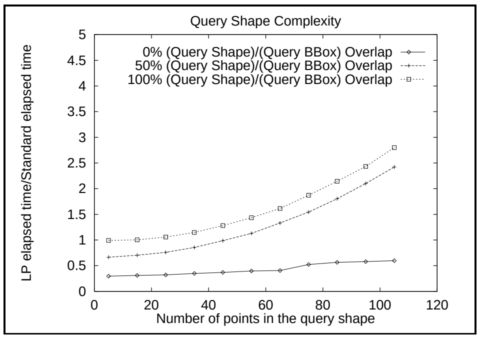

Figure 11: Varying Query Complexity (LP)

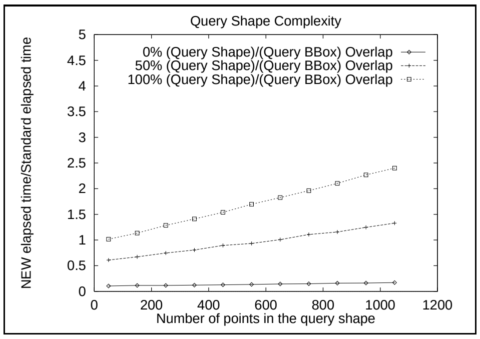

Figure 12: Varying Query Complexity (Clipping)

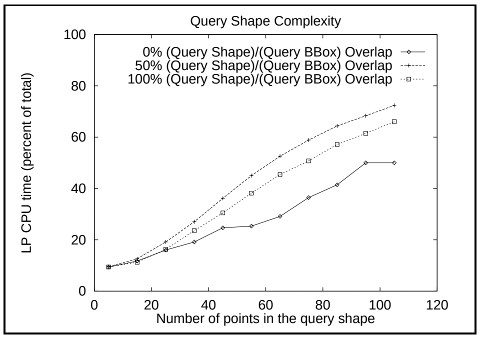

Figure 13: Varying Query Complexity (LP) – CPU

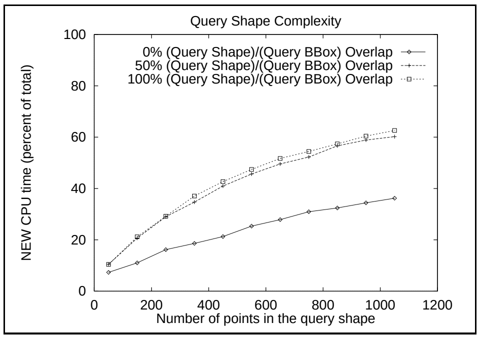

Figure 14: Varying Query Complexity (Clipping) – CPU

6.3 Number of Points in the Query Polygon

The complexity of a query, in terms of the number of points in the bounding polygon, affects the cost of the constraint overlap test. This experiment uses the same data set as the one that varied query selectivity, but generates queries a little differently. The same basic query shape is still used, but the number of edges in the bounding polygon is artificially increased by treating each side of the polygon as a series of line segments. Each segment adds a linear constraint to the bounding polygon. Although these new segments are all described by essentially the same constraint, neither LP nor our overlap algorithm exploits this in any way.

Note from Figure 11 that in the worst recorded case, a 100 point query, the constraint search algorithm with LP overlap test takes 2 to 3 times as long as the standard MBB approach. With the Clipping overlap test, a 1000 point query is required before a two-fold slow-down occurs.

Among all our experiments, this is the only one in which the constraint algorithms were seen to perform worse than the standard algorithm. This experiment also indicates that the constraint algorithm is superior as long as the number of points is kept under some maximum (about 30 with the LP overlap test, several hundred with the Clipping overlap test); in all the other experiments that we ran, the MBBs computed for queries—including the 'real' queries discussed in Section 7—never had more than about 30 points. In fact, it is straightforward to impose a maximum limit on the number of sides while generating a bounding polygon for a query. Using simple heuristics, we can usually compute a bounding polygon that is quite close to the convex hull even with a limit of about 30 edges. (With a limit of over a hundred edges if we use Clipping, it is extremely unlikely that even the convex hull will have so many edges, and it seems unnecessary to incorporate any explicit bound on the number of edges.)

Observe in Figures 13 and 14 the increasing CPU usage as query shape complexity rises. Note, however, the differences in both scale and shape of the two graphs. These graphs clearly demonstrate the performance advantage gained by using Clipping over LP.

6.4 Overlap of Objects Within the Database

In this experiment, the underlying data sets were generated by creating a square object at every integral coordinate in a square region of the X-Y plane. The size of each object was varied to vary the degree of overlap amongst objects. All queries were diagonal line queries. The metric used in this experiment was the ratio (sum of the areas of the objects) / (area in the X-Y plane covered by the database). When this number goes above one, there is some overlap in the database.

The results of this experiment are in Figure 15. As the degree of overlap increases, the relative gains of the constraint search algorithms decreases. (In the real datasets used in all other experiments, the degree of overlap was very low. This may be a characteristic of GIS data.)

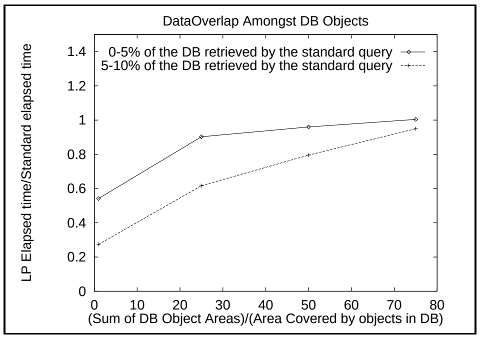

Figure 15: Varying Overlap of Objects(LP)

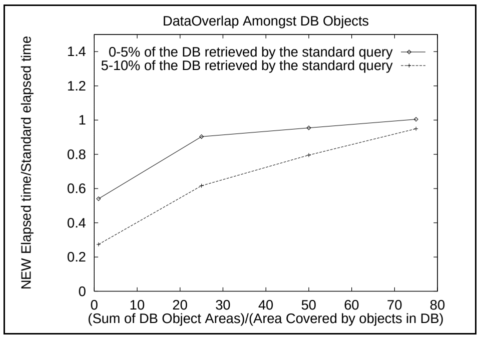

Figure 16: Varying Overlap of Objects(Clipping)

6.5 Page Size

In this experiment, we kept the (query area)/(MBB area) ratio at 0.5 and varied the selectivity and page size. Page size had no effect on relative performance. Details are in [7].

7 Performance Study of LP with a GIS

In this section, we describe a set of realistic tests in the context of a GIS system. We show that the conclusions drawn from the experiments using synthetic queries (Section 6) are borne out in a more realistic environment. The experiments in this section were run on a SUN SPARC 10/40 with 32Meg of main memory. The page size was 1KB and the buffer pool contained 100 pages.

7.1 The Data Set and Spatial Object Representation

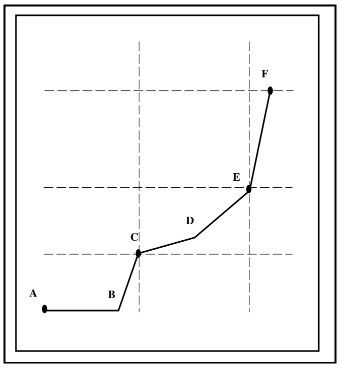

Figure 17: Example of Chains in TIGER file

We used the TIGER/Line data for Milwaukee County. The dataset contains 36,983 *chains* representing spatial features like streets, highways, railroads, rivers and shore lines. Some non-spatial attributes such as name, classification code and zip code are also stored. A *chain* is a set of straight line segments used to represent (part of) a spatial object.

The TIGER data contains quite detailed information. For example, a single street spanning different zip code zones is represented as a collection of connected chains separated by the zip code zone boundaries. Note that we can have two different representations of the same spatial object: it can be treated either as a single object or a collection of objects spatially connected together. See Figure 17 for an example. The street ABCDEF is broken into three chains ABC, CDE and EF due to the fact that it expands different zip code zones. It is stored as three separate chains with identical names and classification codes. Since the end points of the chains match, they can be easily 'stitched' together to form a single object that represents the entire street, if this is desired. Different GIS systems may choose to either stitch or not stitch the data in this manner.

We used two datasets derived from the TIGER data for Milwaukee County in our tests:

- *Unstitched Data:* Identical to the original TIGER data, with each chain as a separate object, with total object count 36,983.
- *Stitched Data:* Obtained from the TIGER data by stitching all chains that represent (part of) a single spatial object.

Data Type	# of Entries	# of 1K Pages	Tree Height
Unstitched Data	36983	2036	4
Stitched Data	10043	580	3

Table 1: R^*-Tree Statistics for Milwaukee County TIGER Data

The two datasets and the (separate) R-Trees constructed using them are summarized in Table 1.

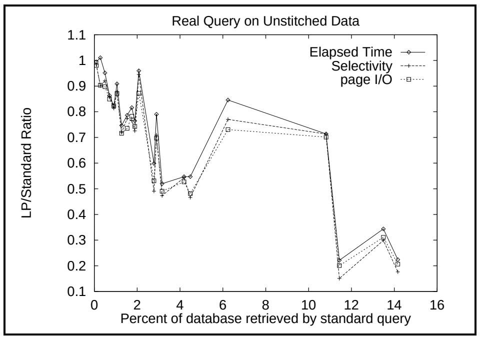

Figure 18: Realistic Queries (LP)

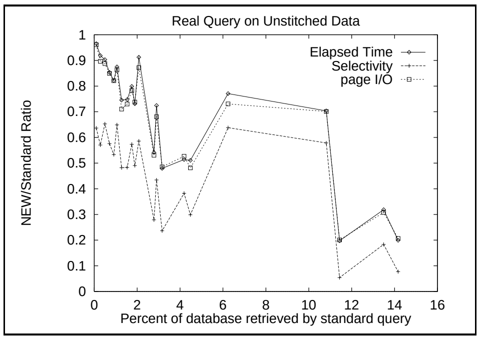

Figure 19: Realistic Queries (Clipping)

7.2 The Queries

We used two different approaches to generating queries. The first approach was designed to observe general trends over a large set of randomly generated 'realistic' queries. For example, queries like "Find all streets that intersect Highway 12" are fairly typical in a real GIS system.

To randomly generate such queries, we used stitched data objects as the query seeds (even when the experiments were run on unstitched datasets). That is, we picked a stitched spatial object (which corresponds to a feature likely to be of interest to an end-user, e.g., a street or railroad) at random and treated it as the query. The query was executed using each of the three candidate search algorithms (STD, LP and Clipping). The measurements were then grouped together into several ranges (for the parameter varied along the X axis), each of which was verified to have at least ten points. Thus, each data point summarizes the numbers due to several queries.

The results on these queries (Section 7.3) broadly confirm the results presented in Section 6, but it is difficult to see the effect of the various parameters involved since the query generation process allows several parameters to vary simultaneously. To address this shortcoming, we also chose a set of queries using the Paradise front-end to give us complete end-to-end performance numbers. [3] These are indeed

[3] Paradise [4] is a prototype object-relational DBMS for GIS applications developed at University of Wisconsin, it is built on top of the SHORE storage manager.

queries that an end-user might ask, and we hand-picked the queries to ensure that a range of situations was covered. These results are analyzed in Section 7.4, and confirm that the results of Section 6 with respect to the impact of the various parameters are indeed accurate.

7.3 Realistic Queries: General Trends

We now present aggregate results on realistic queries chosen at random as described in Section 7.2.

Figures 18 and 19 show the results for unstitched data. Several points should be noted. First, the three ratios (for elapsed time, number of objects returned, and pages read) are very closely correlated, as for synthetic queries (Section5). Second, even when the constraint searches retrieved the same number of pages, elapsed time never exceeded (by more than a marginal amount) that for the standard algorithm. On the other hand, elapsed time was reduced by 10% to 50% for most queries, and even up to 80% in some cases. Third, while the time and page I/O behavior of the version with the Clipping overlap test is essentially the same as that for the version using LP overlap test, the selectivity (number of objects returned) is always substantially less due to the tests at the leaf nodes. (The additional time cost for these checks is included in the graphs.) This can yield substantial reductions in the post-processing time, as we will show in Section 7.4.2. Finally, the reason for the fluctuation in the curves is that parameters like orientation, query shape, and number of edges in the bounding polygon all vary (in addition to selectivity, which is shown on the X axis) with randomly selected queries.

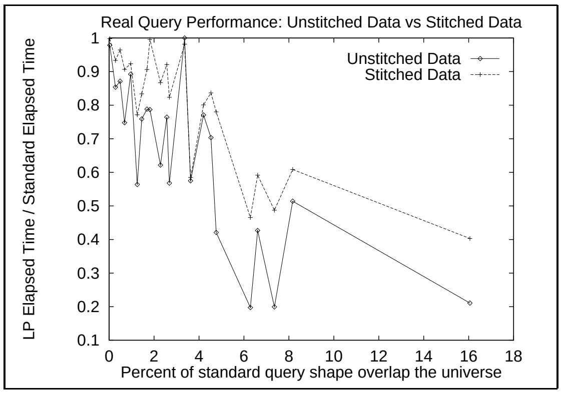

Figure 20: Realistic Queries: Stitched vs Unstitched Data

To study the effect of the stitched versus the unstitched representations, we ran the same set of queries on the stitched representation of the data as well; the results for elapsed time are shown in Figure 20. The results for the other metrics are similar, and we therefore just discuss the elapsed time results. Generally, the comparison between the constraint search algorithms and the standard algorithm is the same for the stitched case as for the unstitched case. Significant gains are often obtained, and the worst case penalty is marginal. However, the relative improvement is less when the data is stitched. There are two reasons. First, the R^*-Trees built on the stitched data has a higher degree of data overlap than the tree built on unstitched data. As observed in 6.4, this reduces the improvement due to the constraint search. Second, the height of the tree on stitched data is one level shorter, which makes queries less expensive overall and leaves less room for performance enhancements.

7.4 Realistic Queries: Selected Examples

In order to understand the behavior of the constraint search technique on typical queries in some detail, we used the Paradise GIS system to visualize the data and generate a selection of interesting queries. A careful analysis of the performance of the candidate search algorithms on these queries illustrates the effect of parameters such as query shape and selectivity, and further confirms the results in Section 6.

We generated three kinds of queries, which are of interest to GIS users in areas such as highway planning,

Query	Label	Name	Description	Orientation
Q1	L4	National Ave	major road	20 degree east/west
Q2	L6	I-43 1	segment of highway 43	near flat, east/west
Q3	L2	I-43 2	segment of highway 43	vertical, north/south
Q4		I-43 1+2	combination of I-43 1 and 2	L shape
Q5		I-43 3	segment of I-43 1	near flat, east/west
Q6		I-43 4	segment of I-43 2	vertical, south/north
Q7		I-43 3+4	combination of I-43 3 and 4	L shape
Q8	L5	Forest Home Ave 1	major road	diagonal
Q9	L1	Brown Deer Road	major road	flat east/west (sparse)
Q10	L3	I-94	segment of highway 94	flat east/west (dense)
Q11		Forest Home Ave 2	segment of Forest Home Ave 1	diagonal

Table 2: Line Query Descriptions

Region	Boundary	Shape
A1	Appleton Ave, Fond du lac Ave, Villard Ave and Walnut St	thin diagonal
A2	Highway I-94, National Ave, Highway I-41 and Highway I-43	near rectangular
A3	Brown Deer Road and two Railroads	near triangular (sparse)
A4	Lake Michigan shore line Milwaukee River and Kenwood Blvd St	complex shape by the lake
A5	Hand-picked (dense)	rectangular (dense)
A6	Silver Spring Dr, Fond du Lac Ave and Railways	triangular (small)
A7	Highway I-43 and Home Forest Ave	triangular (big)

Table 3: Region Query Descriptions

traffic control, urban zoning and real-estate:

1. *Line Queries:* Pick a single route (consisting of one or more segments of highways and local roads) and find all other roads that it intersects.

2. *Band Queries:* Pick a single route, draw a buffer zone, and find all spatial features that overlap the buffer.

3. *Region Queries:* Pick a region which is bounded by highways, local roads, or some other features like rivers or shore lines, and find all spatial objects that overlap the region.

Table 2 describes the line queries, which are shown (using a Paradise view) in Figure 21. The band queries are derived from the above line queries. We varied the buffer size from 0 (the original line) to 10000 meters. The region queries are described in Table 3, and shown in Figure 22.

7.4.1 Result Analysis

The performance results for the selected queries are shown in Tables 4, 6 and 7. All ratios are computed with respect to the standard R-Tree search algorithm.

Line	LP		Clipping		Page I/O Ratio	STD Obj Count	STD Retrieved / Total # Objs
	Time Ratio	Selectivity Ratio	Time Ratio	Selectivity Ratio			
Q1	46.18%	40.16%	44.18%	29.16%	42.07%	4715	11.84%
Q2	70.80%	72.10%	69.56%	48.00%	66.00%	1000	2.51%
Q3	85.02%	90.52%	88.90%	71.77%	82.99%	2912	7.31%
Q4	73.12%	70.80%	72.64%	66.37%	70.71%	24635	61.88%
Q5	79.51%	85.57%	80.12%	48.07%	89.47%	104	0.26%
Q6	91.23%	86.30%	86.30%	53.73%	83.63%	482	1.21%
Q7	76.34%	74.86%	75.10%	55.17%	74.00%	1205	3.17%
Q8	26.84%	23.72%	27.19%	13.14%	25.18%	8747	21.97%
Q9	95.04%	98.76%	95.04%	84.36%	95.55%	243	0.61%
Q10	73.04%	77.25%	72.24%	53.17%	68.51%	1811	4.54%
Q11	26.09%	21.83%	24.15%	7.66%	25.09%	3732	9.37%

Table 4: Line Query Performance Result

Line	LP		Clipping	
	Elapsed Time (milli seconds)	Real Object Count	Elapsed Time (milli seconds)	Real Object Count
Q2+Q3	2723	3637	2844	2570
Q4	10183	17252	10116	16351
Q5+Q6	590	505	614	309
Q7	681	947	703	698

Table 5: 'L' Shape Line Query Performance Result

Width	0 meter (Line)			100 meters			1000 meters		
	Time LP	Sel1 LP	Sel2 Clipping	Time LP	Sel1 LP	Sel2 Clipping	Time LP	Sel1 LP	Sel2 Clipping
Q1	46.18%	40.16%	29.16%	46.78%	39.48%	28.77%	49.06%	41.52%	29.86%
Q2	70.80%	72.10%	48.00%	81.97%	76.08%	47.83%	78.23%	78.01%	52.89%
Q3	85.02%	90.52%	71.77%	84.82%	90.21%	72.81%	84.48%	89.96%	73.75%
Q4	26.84%	23.72%	13.74%	27.03%	23.72%	13.95%	28.02%	25.16%	15.19%
Q5	95.04%	98.76%	84.36%	97.52%	98.80%	82.80%	90.09%	97.19%	87.22%
Q6	73.04%	77.25%	53.17%	73.29%	77.16%	54.22%	78.18%	77.96%	59.13%

Table 6: Band Query Performance Result (LP/STD Query Time)

There are several interesting points to observe in the line query results. First, the orientation of the line queries has a major impact on the performance gains, as illustrated by queries Q1, Q8 and Q10. The constraint search improved Q8, which is diagonally oriented, the most. Q10, which is flat, saw the least improvement, and Q1 was somewhere in between. Second, while the shape of the bounding polygon versus the MBB is in general a good predictor of the gains to be obtained by using constraint search, the distribution of objects is significant as well. For example, Q4 has a triangular bounding polygon, yet the relative improvement is less than that for Q2. This is because the portion of the MBB that is not in the bounding polygon for Q4 is relatively sparse. Third, sometimes there are significant speed-ups to be had by partitioning a query into smaller queries. For example, the cost of Q4 is much greater than the combined cost of Q2 and Q3, although Q4 is just the union of Q2 and Q3. The same point arises when we examine Q4, Q5 and Q6. In principle, one could look for a collection of polygons that cover the query and whose combined area is less than the area of the bounding polygon for the original query. This is, however, an issue that we have not studied in depth.

The results from the band queries also demonstrate the effectiveness of the constraint search algorithm, although the improvement decreases, in general, as the band width increases. This is consistent with the synthetic query results, which indicate that the relative improvement decreases as the overlap of the query with its MBB increases. Nonetheless, we still see performance gains between 10% and 70% for band queries.

For region queries, we see performance improvements ranging from 15% to 60% in all cases except in the MBB query, in which, obviously, no work can be avoided by using a bounding polygon. In this case, the more expensive overlap tests are an unnecessary overhead, but the cost is seen to be negligible (in fact, within the 'noise' level of a lightly loaded machine). Note that similar shapes tend to have performance gains in the same range, which is quite predictable. We see this in queries A6 and A7. Query A4 is an interesting case, similar to the line query Q4. Given the triangular shape, it would seem that performance gains should be well beyond what we got (18%). However, the potential gain was diluted [4] by the fact that most of the region excluded by the use of a polygon (instead of a MBB) falls in Lake Michigan.

In order to illustrate the merits of the Clipping overlap method, we included results for the constraint search algorithm using both the LP and Clipping overlap methods. In almost all the cases, the selectivity ratio is less (fewer retrieved objects from R-Tree) when we use the Clipping overlap method, although the time ratio is roughly the same. The reduction in selectivity is in the range of 10% to 20%. This could potentially have a huge impact on post-processing.

Region	LP		Clipping		Page I/O Ratio	STD Obj Count	STD Retrieved / Total # Objs
	Time Ratio	Selectivity Ratio	Time Ratio	Selectivity Ratio			
A1	37.30%	34.60%	34.89%	23.43%	35.34%	5360	13.46%
A2	84.89%	93.89%	84.76%	79.16%	79.06%	835	2.09%
A3	51.53%	50.85%	52.03%	40.76%	50.64%	3270	8.21%
A4	85.43%	83.49%	84.36%	72.85%	79.52%	1654	4.15%
A5	1.009%	100.0%	99.57%	100.0%	100.0%	2303	5.78%
A6	66.74%	65.14%	64.79%	50.70%	65.44%	1836	4.61%
A7	61.15%	56.07%	55.90%	46.77%	56.14%	5375	13.50%

Table 7: Region Query Performance Result

[4] Pun intended.

7.4.2 Post-Processing

Overall, we have shown that for the selected 'realistic' queries, constraint search runs significantly faster than the standard search algorithm in most cases, and is never more than marginally worse. This is true whether we use LP or the Clipping algorithm for overlap tests. Using the Clipping test, we are additionally able to reduce the number of returned objects substantially.

In a spatial query processing environment, R-Tree search only serves as a filter. The retrieved objects are typically processed further. This post-processing can be very expensive, e.g., retrieving the spatial object—note that the R-tree search returns object ids, typically—and verifying whether the object overlaps the query. Obviously, the fewer the candidates returned by the R-Tree, the fewer the objects to be processed further.

To measure the impact of this factor on the overall reduction in costs due to the constraint search algorithms (which return fewer objects than the standard search algorithm), we modified the Paradise system to use the constraint search code in its R-Tree code, and measured the total elapsed time and I/O for the region queries. The performance numbers can be seen in Table 8. Note that the constraint search algorithm (Clipping) always wins over the standard R-Tree search. In some cases, the speedup is as high as a factor of 3. In most cases, the query time reduction closely matches the selectivity reduction.

Region	STD Time (sec)	Time Ratio Clipping/STD	Selectivity Clipping/STD	Index Page I/O Clipping/STD	Total Page I/O Clipping/STD
A1	12.53	34.31%	23.43%	35.43%	32.61%
A2	3.45	76.23%	79.16%	79.06%	83.62%
A3	8.72	51.72%	40.76%	50.64%	48.95%
A4	5.36	81.90%	71.85%	79.52%	77.92%
A5	6.32	100.3%	100.0%	100.0%	100.0%
A6	4.40	96.59%	50.70%	65.44%	94.27%
A7	12.27	98.20%	46.77%	56.14%	97.01%

Table 8: End-to-End Region Query in Paradise

However, there are a few cases (A6 and A7) in which such correlated speedup is not observed. For example, A7 got less than 2% reduction in time, while the selectivity ratio is in the range of 40% to 50%. A careful study of the total I/O ratio provides us with some insights on this seemingly odd behavior. Reduction in the number of objects retrieved is not the only factor in determining post-processing performance. The object size also matters. For both A6 and A7, fewer entries are extracted from the R-Tree, but they are all big objects. The majority of the time was spent in accessing those large objects.

Even though there were a couple of cases where object size worked against us in the selected queries, it could equally well have worked for us. In other words, there could have been a query in which one very complex (large) object was eliminated by using a linear constraint filter.

Note that even with these perturbations, selectivity is still highly correlated with overall time. This is a consequence of the fact that post-processing in Paradise retrieves all objects whose id is returned by the R-Tree search. Our performance results suggest a refinement which could greatly reduce post-processing cost if the standard R-Tree search algorithm is used: *Approximate the query by a polygon and test overlap with the bounding box of each object returned by the R-Tree search algorithm. Only retrieve those objects that pass this test.* For overlap, the Clipping overlap test or LP could be used. By incorporating our overlap test into the R-Tree search algorithm, the need for this extra check during post-processing is eliminated.

8 Query by Linear Constraint for High Dimensional Queries

Linear constraints are a very powerful tool for formulating queries over multidimensional point datasets, and can be used in a variety of applications, including business and scientific analysis. In this section, we present examples of such queries and experimental results concerning their performance. Note that the region covered by such queries may be 'open', i.e., the only possible bounding box would be based on the MBB for the root of the R-Tree, which is of no use for restricting search. Thus, the constraint search techniques that we have presented are essential for supporting the queries discussed in this section; the standard R-Tree search algorithms, and even techniques that view queries as polyhedra (in terms of vertex representations), are of limited use.

r	#rounds	%CPU	matches	page reads
0.1	1	15	16	2396
0.1	2	19	16	2396
0.1	3	19	16	2396
0.2	1	16	51	4149
0.2	2	20	51	4149
0.2	3	20	51	4149
0.3	1	16	107	5604
0.3	2	21	107	5604
0.3	3	21	107	5604
0.4	1	17	207	6942
0.4	2	21	207	6942
0.4	3	21	207	6942
0.5	1	16	313	8114
0.5	2	20	313	8114
0.5	3	20	313	8114

Table 8: P/E query results.

For our data set, we use a synthetically extended version of the Compustat stock and business data set. The portion of the data set we are concerned with includes 5 fields, forming a five dimensional point data set. The five fields we are concerned with are *year number, earnings per share, stock low, stock high,* and *stock value at year close*. The data set originally contained 20 years worth of data on approximately 4000 companies. Unfortunately, approximately half the tuples were null values (the company didn't exist yet), [5] so we synthetically generated enough data to account for 100 years of information. The last 20 years contain the information directly from the database (minus null values). The data was generated using a uniform distribution between the high and low values for a field for each company. In addition, all integrity constraints such as $stock_{low} < stock_{high}$ were maintained by selecting the appropriate range in random number generation. The resulting R^*-tree contained 37466 pages.

For this data set, we explore variations of two queries. The first query is: *Retrieve all tuples which have a price-earnings (P/E) ratio less than some constant.* Investors frequently use the P/E ratio as a measure of how undervalued or overvalued a company is. In linear constraint terms, the above query can be expressed as: $price_{year-end} = r * earnings \leq 0$ (where r is the ratio value). The second query that we consider is: *Retrieve all tuples that correspond to companies whose value didn't vary by more than some percent of the price at year's end.* In terms of linear constraints: $price_{high} - price_{low} - p * price_{year-end} \leq 0$ (where p is the percentage above). This measure is indicative of a stock's *volatility*.

8.1 Query 1: A Single Linear Constraint

For this query, we used the P/E constraint dicussed above. The ratio r was varied to generate different selectivities. In addition, we examined the effect of the maximum number of clipping rounds on both page reads and CPU time.

Note from the results in Table 8 that additional rounds of clipping had no effect on the number of page reads or matches. In addition, observe the jump in CPU cost from one round of clipping to two, but none after. These results are explained by the need for only one round of clipping. In theory, one round of clipping may not be enough to determine overlap. In practice, we have yet to observe a situation in which

[5] More extensive versions of the Compustat dataset are available, for a price!

more than one round of clipping is necessary. The CPU costs of the query rose from one to two rounds of maximum clipping since one extra round of clipping must be done to determine algorithm completion.

8.2 Query 2: A Combination of Linear Constraints

This query consists of selecting all tuples such that the P/E ratio is less than .5, the value of the company hasn't changed by more than some percent p and the tuple belongs to a measurement from the last 50 years. In terms of linear constraints:

$$\begin{cases} price_{year-end} - 0.5 * earnings \leq 0 \\ price_{high} - price_{low} - p * price_{year-end} \leq 0 \\ year \geq 50 \end{cases}$$

p	%CPU	matches	page reads
0.100000	0.256087	8	8109
0.200000	0.257674	13	8113
0.300000	0.257023	16	8113
0.400000	0.257004	20	8114
0.500000	0.249743	26	8114

Table 9: Complex Query.

Table 9 shows the performance of the above query while varying p. Since one round of clipping was sufficient for all experiments (as in Query 1), only the results for one round of clipping are displayed in Table 9. Note that the slightly higher CPU cost is a result of adding more constraints.

8.2.1 The Degree of Overlap In High-Dimensional R*-Trees

We observe that there is a surprising amount of I/O in the above queries given the overall selectivity. This is due to a large amount of overlap in the R*-tree generated by our data set. To better illustrate this, the results in the following table give the average, low, and high number of page reads generated by 100 point queries. These point queries correspond to 100 points from the Compustat data set. Thus, the selectivity of each data set is 1 tuple.

Min I/O	Max I/O	Avg I/O
2	15770	4089

The large number of page reads performed to retrieve a single point seems to be caused by the inability of the split algorithm (used in R*-Tree insertion) to function well in high numbers of dimensions. This is, however, a deficiency of the R*-Tree and is not caused by or related to query by linear constraint. (On the particular data set that we studied, the degree of overlap could possibly have been reduced by applying a good bulk-loading algorithm; this would have greatly reduced the I/O for the example queries.) It is interesting to note that query by linear constraint can be applied with equal efficiency to many multidimensional indexing structures. Some of these include grid files, buddy trees, P-trees, and TV trees (using an L1 distance metric). Thus the value of query by linear constraint is in not tied to the use of the R-Tree as a high dimensional indexing structure.

9 Related Work

We've made two main contributions: an algorithm to test overlap of an MBB and a conjunction of constraints in an arbitrary number of dimensions, and a search algorithm in tree-structured multidimensional index that bounds queries by linear constraints.

The overlap algorithm is an important part of the search algorithm. There are Computational Geometry algorithms for detecting polyhedron—box overlap or polyhedron—polyhedron overlap (e.g. see [22]); why can't these be used instead? The input to these algorithms is usually the vertices of the polyhedra and not the tight representation of the conjunction of linear constraints. Since the number of vertices of a polyhedron is exponential in the number of constraints (for arbitrary dimensionality), converting the input to fit those algorithms will be prohibitively expensive.

Several variants of the R-Tree [10] have been proposed, including the R^+-Tree [25] and the R^*-Tree [1]. Other closely related structures are the k-d-B Tree [24] and the Buddy-Tree [15, 16]; they all use boxes (MBBs) to bound the space represented by a tree node. While they have different insertion and deletion algorithms, they use essentially the same search algorithm (although, for point queries, the Buddy-Tree, k-d-B Tree and R^+-Tree have the property that only a single path from root to a leaf is searched).

Popular access methods for point data include the Grid File [19]. The performance study in [1] suggests that the R^*-Tree is superior to the other variants of the R-Tree, and indeed even the Grid File for point data. Consequently, the R^*-Tree was implemented as the spatial access method for the Paradise GIS on top of the Shore data manager. For the same reasons, we chose to evaluate the performance of our refinement with respect to the R^*-Tree variant.

The other proposals (that we are aware of) that do not use MBBs to bound objects and/or queries are the Sphere tree [28], the Cell tree [8, 9] and the P-tree [12].

The Sphere tree uses spheres instead of boxes to bound both queries and data in the tree. Storing spheres has the advantage that the orientation of the axes does not affect performance. The major disadvantage is that "narrow" objects (such as lines) are very poorly approximated by spheres.

The Cell tree uses convex polyhedra (which can be thought of as a special case of a conjunction of linear constraints) to store data in the tree as well as to query it. The space overhead for describing each polyhedron is considerable even if the tightest description possible (i.e., linear constraints) is used. The methods used by Günther and Wong [9] for polyhedron–polyhedron overlap detection for the cell-tree do not seem feasible in more than three dimensions. (A 4-dimensional test of two polyhedra with 16 vertices in each takes about 10^7 terabytes of space and about 10^{20} operations for preprocessing!) Even if we consider using algorithms such as Simplex or Karmarkar's algorithm [6] instead of the overlap checks proposed for the Cell tree, the CPU overhead is likely to be too high.

The P-tree [12] has a structure close to the R-tree and uses high dimensional boxes as representations for lower dimensional polyhedra. If we take, say, a 2-D object and bound it with a 2-D MBB as in an R-Tree, the approximation may be poor if the object is not oriented nicely w.r.t the two axes. If we consider k additional axes and use $k+2$ dimensional MBBs to bound objects and queries, we get a better approximation. This is the essential idea behind the P-Tree, but there are some important drawbacks. First, R-Trees are very sensitive to increases in dimensionality of the data (see [21]). If it is intended that point data be stored in the R-Tree, the only advantage gained from use of the P-Tree is a more precise query form (at the cost of added dimensions in the tree itself). In addition, any advantage derived from more precise querying is limited by the accuracy with which the query can be expressed in terms of the (predetermined) set of axes. Finally, determination of conservative (slightly overlarge) polyhedral bounds of arbitrary polyhedra is not discussed in the paper. This is a very difficult problem and is central to the use of the P-Tree as a general purpose structure.

Our technique of bounding the query by constraints can be used to improve the search performance of any of the proposed tree index structures (e.g., R-Tree variants, K-d-B Trees, P-Trees) that use MBBs. Even if the P-Tree is used, our technique can provide better approximations for the query since we are not limited by a predetermined set of axes.

A summary of several kinds of approximations that can be used to represent a spatial object is presented in [14]. One of the suggested techniques is decomposition of a complex spatial object into a collection of smaller objects. Each of the smaller objects is then approximated by a MBB. The decomposition idea is discussed in the context of spatial join queries to filter out pairs of objects that do not overlap, but not in the context of search queries. The idea of decomposing a large query into smaller queries can certainly provide improvements in search performance (e.g., consider a diagonal query object, w.r.t. the axes). However, even if the decomposition produces non-overlapping objects, the queries corresponding to these objects may involved repeated traversal of parts of the tree. Clearly, a similar idea could be used in conjunction with our use of constraints to bound queries: we could look for a *collection* of convex polygons that together cover the query. As indicated in Section 7, such an approach holds the potential for dramatic speed-ups in some cases.

Several other spatial access methods have been proposed, e.g., [18, 20], but these are not directly related to our work. Other query types that have been proposed are polygon queries [14] and spline and Bézier curve queries [13]. Both methods look promising. However, it is not clear how the polygon queries scale to higher dimensions and the only applications that use splines or Bézier curves seem to be in CAD/CAM. In both cases, the evaluation methods are not directly related to our work because our input for the queries is different.

10 Conclusions and Future Work

We have proposed a simple refinement to the R^*-Tree search algorithm and demonstrated that it offers the potential for significantly enhanced performance over a wide range of queries, with little or no penalty even in the worst case. It is straightforward to add such a refinement to an existing R^*-Tree implementation. An important contribution of this work is the efficient and safe polygon–box overlap test presented in Section 3.

As a next step in the application of constraint techniques to R^*-Trees, we intend to explore the use of MBBs that are *rotated* with respect to the axes to store data in the tree. Our search algorithm would of course work without change in such a structure. However, new insertion and deletion algorithms are required.

References

[1] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. "The R^*—Tree: An Efficient and Robust Access Method for Points and Rectangles". Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp. 322-331.

[2] Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C., Tsatalos,O., White, S., Zwilling, M. "Shoring up Persistent Objects". Proc. ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 383-394.

[3] Dantzig, G.B. "Linear Programming and Extensions". Princeton University Press, Princeton, N.J. 1963.

[4] DeWitt, D., Kabra, N., Luo, J., Patel, J., Yu, J. "Client-Server Paradise". Proc. 20th Int. Conf. on VLDB, 1994. pp.558-569.

[5] Faloutsos, C., Roseman, S. "Fractals for Secondary Key Retrieval," Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on PODS, 1989, pp. 247-252.

[6] Fang S. C., Puthenpura S. "Linear Optimization and Extensions: Theory and Algorithms". Prentice Hall 1993.

[7] Goldstein J., Ramakrishnan R, Shaft U., Yu J. B. "Using Constraints to Query R^*-Trees". Technical Report No. 1301 , Comp. Sci. Dept., Univ. of Wisconsin-Madison, 1996. Full Manuscript.

[8] Günther, O. "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases". Proc. 5th Int. Conf. on Data Engineering, 1989, pp. 508-605.

[9] Günther O., Wong E. "A Dual Approach to Detect Polyhedral Intersection in Arbitrary Dimensions". Proc. 25th Annual Allerton Conf. on Comm., Control and Comp., Oct 1987, pp. 859-868.

[10] Guttman, A. "R—Trees: A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47-57.

[11] Jagadish, H.V. "Linear Clustering of Objects with Multiple Attributes," Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp.332-342.

[12] Jagadish, H. V. "Spatial Search with Polyhedra". Proc. IEEE 6th Int. Conf. on Data Engineering 1990, pp. 311-319.

[13] Kriegel, H.P., Heep S., Fahldiek A., Mysliwitz N. "Query Processing of Geometric Objects with Free Form Boundaries in Spatial Databases". Proc. 4th Int. Conf. DEXA 1993, pp. 349-360.

[14] Kriegel, H.P., Horn, H., Schiwietz, M. "The Performance of Object Decomposition Techniques for Spatial Query Processing". Proc. 2nd Symposium on Large Spatial Databases, Lecture Notes in Computer Science, Vol 525, Springer, 1991, pp. 257-276.

[15] Kriegel, H.P., Schiwietz M., Schneider R., Seeger, B. "Performance Comparison of Point and Spatial Access Methods". SSD 1989, pp. 89-113.

[16] Kriegel, H.P., Schiwietz M., Schneider R., Seeger, B. "The Buddy-Tree: An Efficient and Robust Method for Spatial Data Base Systems". Proc. 16th VLDB Conf. 1990, pp. 590-601.

[17] Kuenzi, H.P., G.B., Tzschach, H.G., Zehnder., C.A. "Numerical Methods of Mathematical Programming". New York Academic Press, 1971.

[18] Lomet, D. and Salzberg, B. "The hB-Tree: A Multi-attribute Access Method with Good Guaranteed Performance," ACM TODS Vol. 15, No. 4, Dec. 1990.

[19] Nievergelt, J., Hinterberger, H., Sevcik, S.C. "The Grid File: An Adaptable, Symmetric Multikey File Structure," Readings in Database Systems, Morgan Kaufmann, 1988.

[20] Orenstein J.A. and Merrett, T. "A Class of Data Structures for Associative Searching," Proc. 3rd ACM SIGACT-SIGMOD-SIGART Symposium on PODS, 1984, pp. 181-190.

[21] Otterman M., "Approximate Matching with High Dimensionality R-trees". M.Sc. Scholarly paper, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1992. Supervised by Faloutsos C.

[22] Preparata F. P., Shamos M. I. "Computational Geometry: An Introduction". Springer-Verlag 1990 (3rd printing).

[23] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T. "Numerical Recipes in C," Cambridge University Press, Cambridge. 1988.

[24] Robinson, J. "The k-d-B Tree: A Search Structure for Large Multi-dimensional Dynamic Indexes," Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981, pp. 10-18.

[25] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R^+—Tree: A Dynamic Index for Multi—Dimensional Objects," Proc. 13th Inf. Conf. on VLDB, 1987, pp. 507-518.

[26] Seeger, B., Kriegel, H.P. "Techniques for Design and Implementation of Efficient Spatial Access Methods," Proc. 14th VLDB Conf., 1988.

[27] Stonebraker, M., Frew, J., Gardels, K., Meredith, J. "The Sequoia 2000 Storage Benchmark". Proc. ACM SIGMOD Int. Conf. on Management of Data, 1993, pp. 2-11.

[28] Van Oosterom, P., Claasen, E. "Orientation Insensitive Index Methods for Geometric Objects," Proc. 4th International Symposium on Spatial Data Handling, 1990,

A The Clipping Algorithm

We denote this predetermined number of iterations as I. The general form of the algorithm is:

1. Repeat I times the following:
   (a) For $i := 1$ to m do
      i. Find smallest possible B' s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$.
      ii. If $B' = \emptyset$ stop with output "No".
      iii. Set $B := B'$.
   (b) If B has not changed then stop with output "Yes".
2. Return "Yes"

A step (clipping by constraint i and all dimensions) can be done as follows:

1. Define $\vec{z} = (z_1, .., z_n)$ as follows: If $A_{ij} > 0$ then $z_j = h_j$ else $z_j = l_j$.
2. Define $d = c_i - \vec{A}_i \cdot \vec{z}$.
3. For each $1 \leq j \leq n$ define l'_j, h'_j as follows:

   If $A_{ij} = 0$ then $l'_j = l_j$ and $h'_j = h_j$

   If $A_{ij} > 0$ then $h'_j = h_j$ and $l'_j = \max\{l_j, h_j + \frac{d}{A_{ij}}\}$

   If $A_{ij} < 0$ then $l'_j = l_j$ and $h'_j = \min\{h_j, l_j + \frac{d}{A_{ij}}\}$
4. Set $B' = [l'_1, h'_1] \times \cdots \times [l'_n, h'_n]$

Note that this calculation of B' may yield an empty box. This happens iff $d > 0$. To save some of the cost of the algorithm in this case we can insert a step "If $d > 0$ return "No intersection"" between steps 2 and 3 of the algorithm.

B Paradise Screens for the Real Queries