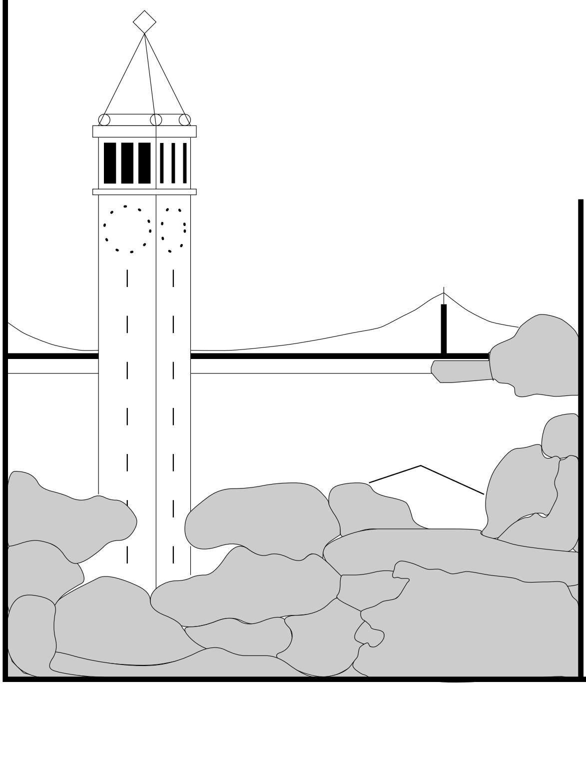
Bounding Predicates and Insertion Policies for Multidimensional Indexes

Megan Thomas, Kirby Zhang, Jimmy Shih and Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
{mct, silkworm, jshih, jmh}@cs.berkeley.edu



Report No. UCB/CSD-0-1098

May 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Bounding Predicates and Insertion Policies for Multidimensional Indexes

Megan Thomas, Kirby Zhang, Jimmy Shih and Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
{mct, silkworm, jshih, jmh}@cs.berkeley.edu

ABSTRACT

We present two new techniques for improving the performance of multidimensional indexes. For static data sets, we find that bulk loading techniques are effective at clustering data items in the index; however, traditional designs of an index's bounding predicates can lead to poor performance. We develop and implement in GiST three new bounding predicates, two of which have much better performance characteristics for our Blobworld image-search application than several traditional access methods. We then proceed to study dynamic data sets, the analysis of which lead to a focus on insertion algorithms. We develop, implement, and analyze an insertion algorithm called the *Aggressive Insertion Policy*, which uses global rather than greedy information when making insertion decisions.

1 Introduction

Multidimensional access methods (indexes) are used to speed search in a variety of new applications, and as a result have been the subject of significant study in recent years. Yet our experience with applications we are studying shows that there is still quite a bit of room for improvement. This paper presents two new techniques to improve performance of both near-neighbor and range queries in multidimensional access methods.

First, we consider indexing static data sets, which can be effectively bulk loaded into an access method. We demonstrate that the *bounding predicates* (or "keys") in an index have a major impact on search performance for static data sets. In designing an access method tailored for the Blobworld image retrieval application [3], we developed three new bounding predicates that provide significant performance improvements. Two of our new bounding predicates, the JB and the XJB, result in access methods that achieve nearly optimal performance for Blobworld's near-neighbor workload, performing about twice as well as the best of the traditional access methods we analyzed.

Second, we consider indexing dynamic data sets, for which the access method's insertion algorithms play an important role. We develop a new insertion algorithm we call the *Aggressive Insertion Policy* (AIP), which uses a technique akin to near-neighbor search to choose a leaf node for a newly inserted item. AIP's choice is independent of the access method's bounding predicates, choosing a tree-global leaf for insertion. A variety of experiments show that access methods loaded via AIP perform up to 35% fewer I/Os on various workloads than the R*-tree, over a variety of insertion-loaded data sets; this effect is most dramatic for higher-dimensional workloads.

Section 2 provides background information on GiST and `amdb`. From Section 3 to Section 5 we focus on the development of an access method for the static Blobworld data set. Section 3 provides an overview of the Blobworld access method problem and presents the analysis of the performance of a few standard access methods. We present our new access method designs, derived from intuitions gained analyzing the traditional access methods, in Section 4. Section 5 analyzes the performance of the new access method designs. Section 6 introduces our new insertion policy, and Section 7 presents the experiments we conducted to validate its utility. Section 8 covers related work. In Section 9 we discuss the conclusions of our study and opportunities for future work.

2 Background

2.1 GiST

We implemented our access methods (AMs) in the Generalized Search Tree (GiST) framework [1, 10]. The GiST frameworks enables AM designers to create new tree-structured AMs with a minimum of design and coding effort. GiST provides the tree maintenance and concurrency control infrastructure. All the AM designer needs to implement is the code specific to the particular AM.

The GiST framework generalizes the notion of a height-balanced, multi-way tree. A GiST provides "template" algorithms for navigating and modifying the tree structure through node splits and deletes. Like all other (secondary) index trees, the GiST stores *(key, RID)* pairs in the leaves; the RIDs (Record IDentifiers) point to the corresponding records on data pages. Internal nodes contain *(predicate, child page pointer)* pairs; the predicate evaluates to true for any of the keys contained in or reachable from the associated child page. This captures the essence of a tree-based index structure: a hierarchy of predicates, in which each predicate holds true for all keys stored under it in the hierarchy. An R-tree [9] is a well known example with these properties: the entries in internal nodes represent the minimum bounding rectangles of the data below the nodes in the tree. The predicates in the internal nodes of a search tree will subsequently be referred to as bounding predicates (BPs).

Apart from the structural requirements, a GiST does not impose any restrictions on the key data stored within the tree or their organization within and across nodes. In particular, the key space need not be ordered, thereby allowing multidimensional data. Moreover, the nodes of a single level need not partition or even cover the entire key space. The leaves, however, partition the set of stored RIDs, so that exactly one leaf entry points to a given data record.

A GiST supports the standard index operations: SEARCH, which takes a predicate and returns all leaf entries satisfying that predicate, possibly using a particular traversal strategy [1]; INSERT, which adds a *(key, RID)* pair to the tree; and DELETE, which removes such a pair from the tree. SEARCH recursively descends *all* subtrees for which the parent entry's BP is consistent with the search predicate (employing the user-supplied extension method *consistent()*), in order to find all leaf entries satisfying a search predicate. A BP's task is to describe, or cover, that part of the data space which is present at the *leaf* level of its associated subtree. The GiST implements these operations with the help of a set of extension methods supplied by the access method developer. The GiST can be specialized to one of a number of particular access methods (AMs) by providing a set of extension methods specific to that AM. These extension methods encapsulate the exact behavior of the search operation as well as the organization of keys within the tree.

2.2 `amdb`

`Amdb` [15, 18] is an AM visualization, profiling and debugging tool for GiST, incorporating a general analysis framework for AM performance. The `amdb` analysis process takes a GiST AM loaded with data and a workload (a set of queries) as input. It provides metrics that characterize the observed performance, in page accesses, of the AM in the context of the given workload. `Amdb` analysis compares the AM performance to the performance of an idealized AM, with metrics reflecting how much performance the GiST AM lost relative to an "optimal" AM. `Amdb` helps the AM designer find areas for potential improvement by relating the performance properties of the AM to the user-defined extension methods.

`Amdb` uses several metrics for AM performance. Here we focus upon the key metrics, without in-depth discussions of the underlying equations; the interested reader is referred to [15] for details. The first metric is *excess coverage*, a measure of the effect of BPs covering more of the data space than is necessary to represent the data contained in the subtrees. A BP exhibiting excess coverage may cause a query to visit pages covered by that BP even when those pages store no data satisfying the query. *Utilization loss* occurs when the the storage utilization of nodes is less than a given target utilization. The idea behind utilization loss is that I/Os may occur that could have been avoided if the data in the nodes were packed more tightly. *Clustering loss* expresses the difference between the organization of data into particular leaf nodes in the AM, and the organization of data into leaf nodes in an "optimal" AM. This captures the performance loss that occurs when data that will be returned in the same query are not located in the same node. `Amdb` finds the "optimal" data clustering for a workload by using the hypergraph partitioning heuristic in [11]. Truly optimal clustering is NP-hard, but this heuristic works well in practice and hence serves as a good benchmark for comparison.

3 Blobworld Access Method Design

First we discuss creating a customized AM for a particular application, the image retrieval system Blobworld.

3.1 Blobworld Problem Description

Blobworld is a system for image retrieval based on finding coherent image regions which roughly correspond to objects.

Blobworld images are treated as ensembles of a few "blobs" representing image regions which are roughly homogeneous with respect to color and texture. Each blob is described by its color distribution and mean texture descriptors. Querying is based on the attributes of one or two regions of interest, rather than a description of the entire image. Details of the Blobworld image processing and querying algorithms may be found in [3, 7]. Here we focus on indexing the color feature vectors.

We perform dimensionality reduction using Singular Value Decomposition on the color feature vectors and load the five most significant dimensions of the resulting vectors into an access method. We found [7, 19] that indexing five-dimensional vectors and using nearest neighbor queries returning 200 "blobs" to the Blobworld ranking algorithms ensured enough "good" images were in the final query results to satisfy the Blobworld designers. Therefore, for the rest of the Blobworld AM development we use five-dimensional data vectors and query workloads consisting of nearest neighbor queries that retrieve 200 images each.

3.2 Query Workload

The on-line Blobworld system is a research prototype, so no large set of actual user queries exists. Of the user queries that have been recorded, the majority have been filtered through the Blobworld welcoming page,[1] and hence are based on one of the eight sample images on that page.

We needed a wide variety of queries to properly stress test the AM performance, so we elected to use an artificially generated workload. We randomly selected 5531 blobs from the data set of 221231 blobs (35000 images) to serve as foci of the nearest neighbor queries in our query workload, each query returning 200 blobs. This is enough queries so that every blob in the data set will, on average, be retrieved by several queries. We used this workload in the `amdb` analysis of the performance of each type of AM.

3.3 Access Method Efficiency

We aim to design the most efficient AM we can, where the metric to minimize is leaf-level I/Os, since the inner nodes of the AM can reasonably be assumed to be in memory. The tradeoff between leaf level and total I/Os is examined in more detail in Section 5.

In order for the AM to be worth using, AM performance *must* be faster than simply scanning a flat file of the five-dimensional feature vectors. Because AM disk accesses are random I/Os and a flat file scan is sequential, AM I/Os can be around 15$\times$ slower than sequential scan I/Os.[2] Thus the AM must not hit more than one in fifteen of the leaf-level pages in the AM on each query.

Blobworld image processing is compute-intensive and time consuming,[3] so it is done

[1] Which the reader is invited to examine at `http://dlp.cs.berkeley.edu/photos/blobworld/`.

[2] Using Seagate Barracuda ultra-wide SCSI-2 drives, [17] measures a throughput of 9 MB/s under Windows NT. The average seek time and rotational delay for this drive are 7.1 ms and 4.17 ms, respectively. For 8 KB transfers, this results in a ratio of 14 sequential I/Os for each random I/O. In the past, raw drive throughput has increased faster than seek times and rotational delay have decreased, so the ratio between random and sequential I/Os is likely to increase in the future.

[3] Image processing takes seven to twelve minutes per image on a Sun Ultra-1 with 128MB of memory.

via off-line, batch processing. Because of this, the Blobworld data set is static, and we can ignore data insertion and deletion in our AM development. Most importantly, we can bulk-load the data, which drastically improves the clustering of data items in the leaf nodes of the AMs.

3.4 Traditional Bounding Predicate

We used the STR algorithm [16] to sort the data for bulk-loading into an R-tree [9], which is the canonical multidimensional AM, used in a variety of research and commercial systems. As shown in Table 1, we found that sorting and bulk-loading the data minimized the utilization and clustering loss over our query workload, leaving the largest performance losses for the R-tree[4] in excess coverage. Recall that excess coverage loss is a result of BPs that indicate that relevant data may be in a leaf node when it is not. Essentially, the main problem with a bulk-loaded R-tree is its sloppy BPs.

In order to examine other BPs, we bulk-loaded two other previously-proposed AMs for multidimensional data, the SS-tree [21] and the SR-tree [12]. SS-trees use spherical BPs and SR-trees use minimum bounding rectangles plus bounding spheres. The leaf levels of all of the these trees are dictated by the STR sorting, but the differing BPs cause varying fanout in the inner nodes and varying levels of excess coverage loss when executing the query workload.

Figure 1 shows that the majority of the losses for all three trees are excess coverage losses, with SS-tree performance being the worst of the three AMs. It also shows that the SS-tree performs more unnecessary I/Os for the query workload than the R-tree or SR-tree perform in total. R-tree and SR-tree performance is comparable, with the spheres in the SR-tree BPs saving a small amount of excess coverage loss relative to the R-tree. For both R-trees and SR-trees, about 30 percent of the I/Os are due to excess coverage loss.[5]

[4] While R*-trees [2] are considered better than R-trees over dynamic data sets, bulk-loading the data eliminates any difference between the two AMs.

[5] The relatively poor performance of the SS trees may be aggravated by the STR sorting algorithm, which attempts to organize the data into hyper-rectangular tiles, rather than hyper-spherical regions.

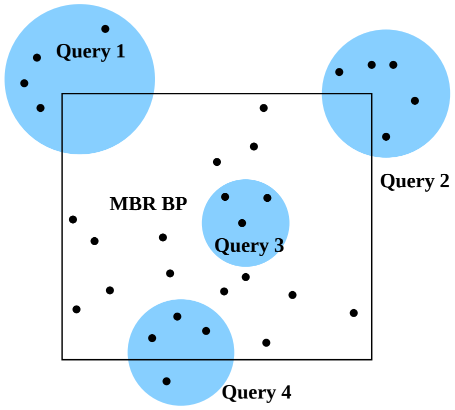


Figure 2: Nearest Neighbor Queries And Rectangular BPs

In the next section we focus on better BPs, which strike a balance between BP size and the precision of data description, for nearest neighbor queries.

4 Towards a Better Blobworld Access Method

An understanding of how nearest neighbor queries interact with BPs provides useful intuitions for designing better AM BPs. Nearest neighbor queries [4] work by finding points of increasing distance from the query point, in essence asking expanding sphere queries. Figure 2 depicts a rectangular BP and the circles of nearest neighbor queries. If the query point is in the middle of a BP, as in Query 3, this node will be accessed. However, nearest neighbor queries starting from points just outside a particular BP may or may not intersect that BP. For good performance on nearest neighbor query workloads, the intersection of BPs with non-matching query spheres, like Queries 1 and 2 in Figure 2, should be minimized.

Ideal BPs cause no excess coverage loss at all, but this goal is difficult to map to geometric properties of potential BPs, so we seek approximations to this goal.

Rather than attempting to visualize five dimensions here, Figure 3 shows a two dimensional example of a source of our intuition regarding BP improvement. The visualization in Figure 3 is of the contents of a few nodes in a two dimensional R*-tree constructed from the two most significant

Losses (in number of I/Os)	Bulk Loaded	Insertion Loaded
Excess Coverage Loss	62683	6027000
Utilization Loss	2768	67562
Clustering Loss	6435	120875

Table 1: Performance Losses in R-trees

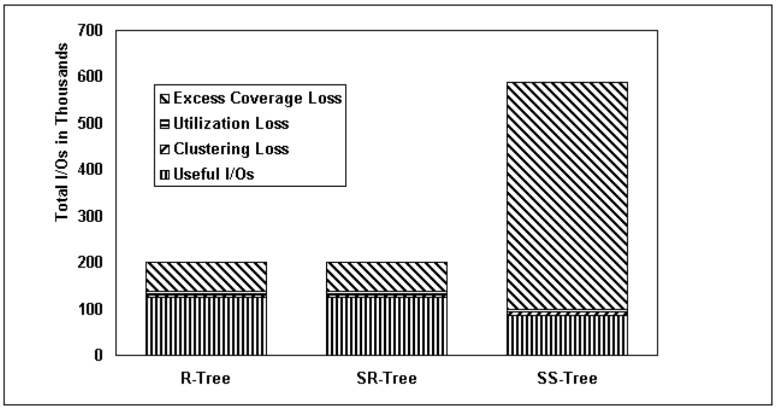


Figure 1: Access Method Performance Losses in Number of I/Os: This shows the number of I/Os that were due to excess coverage, utilization and clustering loss. The remaining I/Os were necessary to answer the queries.

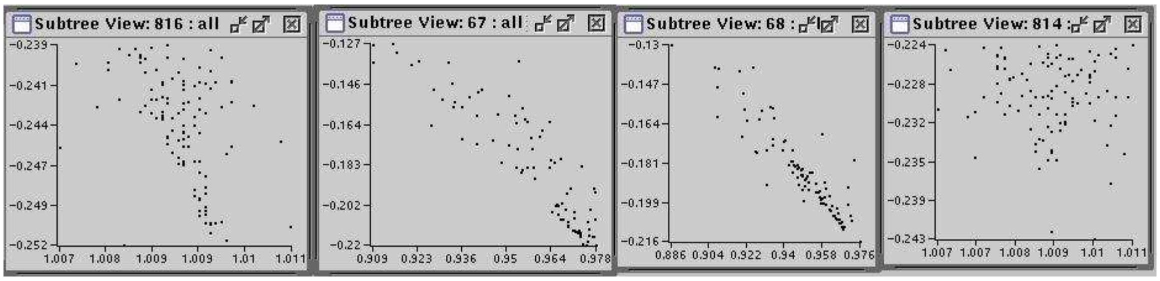


Figure 3: `amdb` Visualization of 2D R-tree Leaf Nodes

dimensions of the Blobworld color vectors. Figure 3 depicts the minimum bounding rectangles (MBRs) of a few leaf nodes and the points they contain. The data points of some leaf nodes do not fill their MBRs, instead leaving noticeable gaps at corners of the MBRs. This suggests that queries may incur I/Os while checking the contents of the empty corners of the MBRs, as the two top queries in Figure 2 do. Combined with the intuition about the importance of reducing spherical intersections with BP regions near the BP edges, this leads us to focus on attempting to remove empty areas from the corners of an MBR BP, by "biting" into the volume of the BP from the corners.

4.1 MAP

To minimize the ability of outside spheres to impinge upon a BP, two hyper-rectangles could be stored as the BP. The first alternate BP we consider, called the Minimum Area Predicate (MAP) BP does just that. Examination of the R-tree revealed that there were 24 children of the root node, and space for about 80. Therefore, the size of the inner node BPs could become larger without incurring the penalty of increasing the height of the AM tree.

The Minimum Area Predicate (MAP) BP, shown in Figure 4, is a variant of a standard R-tree that stores two hyper-rectangles instead of one for each BP. MAP constructs two rectangles such that the total hyper-volume they enclose is minimal. Volume minimization is a heuristic used in a number of earlier AMs [2, 9] when attempting to reduce excess coverage loss. Our conjecture was that the MAP rectangles would form L, T or + shapes that did not include the empty corners, and would thereby reduce excess coverage loss.

The first algorithm that springs to mind to address the problem of finding two rectangles to bound a set of data points may be the R-tree node splitting heuristics. However, those heuristics are aimed at finding two rectangles that overlap as little as possible to bound the data set. Because the rectangles in the MAP BP will be part of the same BP, overlap between them is not necessarily harmful. Using the heuristics developed for R-tree node splitting would optimize the wrong metric for this situation.

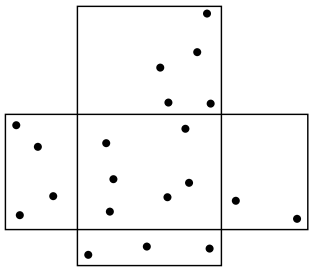


Figure 4: A MAP BP

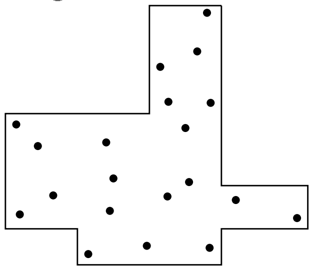


Figure 5: A Jagged Bites BP

For each leaf node, the ideal MAP BP construction algorithm cycles through every possible splitting of the data points into two sets and bounds each set with a MBR. The pair of MBRs with the smallest total volume (counting overlapped regions only once) becomes the BP for that node. However, the time necessary to try every possible partition of the data points is prohibitive. Approximating MAP became necessary. Instead of trying every possible pair of MBRs, approximate MAP (aMAP) tries 1024 randomly selected pairs of sets and picks the MBR pair with the minimum total volume out of those it examines to be the node BP.

4.2 JB

Consider the points in the rectangle in Figure 2. The MAP BP in Figure 4 bounds those same data points more tightly than the MBR BP. However, the MAP BP still leaves empty areas at some corners of each rectangle of the BP in Figure 4. This would be less problematic in a BP that stored the MBR and the largest possible rectangular "bite" out of each corner. This type of BP, which we call a "Jagged Bites" (JB) BP and which is pictured in Figure 5 for the same set of data points, stores the MBR of a set of data, as well as a set of points identifying the bites. Just as the MBR of a data set can be represented by storing two points, one for the highest values in each dimension, and one for the lowest, a corner bite can be represented by the associated MBR corner point and another point at the one "internal" corner of the bite rectangle, which might not intersect any MBR hyper edge.

We use a heuristic algorithm for creating JB BPs because the problem of creating a perfectly optimal JB BP is NP-complete [19]. Figure 6 shows an algorithm for creating a JB BP out of a combination of projections of the data points to the dimensional axes. This heuristic tries every possible value of dimension 0, the first dimension of the STR-created vectors. For each dimension not equal to 0, it finds the minimal value of that dimension among the data points whose dimension 0 values fall between the current value of dimension 0 and the MBR boundary. For the point so constructed, the heuristic calculates the hyper-volume between that point and the corner of the MBR and, if the hyper-volume is greater than the largest hyper-volume seen so far, it keeps that point as the bite apex point.

4.3 XJB

JB BPs are large, leading to low tree fanout. An alternative, called XJB for "Top X Jagged Bites," is to simply store the X largest bites, leaving the remaining MBR corners "unbitten."

Storing the X largest bites for each BP adds $(D+1) \times Xb$ numbers, where b is the number of bytes per number, to the MBR storage space, D numbers to specify the internal bite point, and one number to identify the corner with which the bite is associated.

All three of the BPs we propose involve more complex code for in-memory comparisons than MBR BPs. The functions which are used in the nearest neighbor query algorithms [4] to calculate the distances between predicates and points, and to determine if a predicate contains a given point have more lines of code in the new BPs. However, the new BPs are all rectangle-based. Like ordinary MBR BPs, their distance functions are based around simple rectangle geometry and should not add significantly to query execution time. The BP

1. Find the minimum bounding rectangle, MBR
2. For each corner of the MBR
    (a) set biggest_volume to zero and max_bites[corner] = this MBR corner
        i. For each data point
            A. Construct a sample bite apex point by setting dimension 0 of the bite point to the value of dimension 0 of the current data point.
            B. Find all the points whose dimension 0 values are between this bite apex dimension 0 and the MBR value of dimension 0.
            C. Set the other dimensions of the bite apex point to each equal the lowest value in that dimension of the points between the current bite apex and the MBR in dimension 0.
            D. Find the hyper-volume of the space between the current bite apex point and the MBR corner point
            E. if the hyper-volume > biggest_volume, set biggest_volume = the hyper-volume, and max_bites[corner] = current bite apex point
3. Store the MBR and the set of max_bites as the JB BP

Figure 6: A heuristic for constructing a JB BP from a set of points. For simplicity, this pseudo-code does not deal with the issues raised by corners being high and low in varying dimensions.

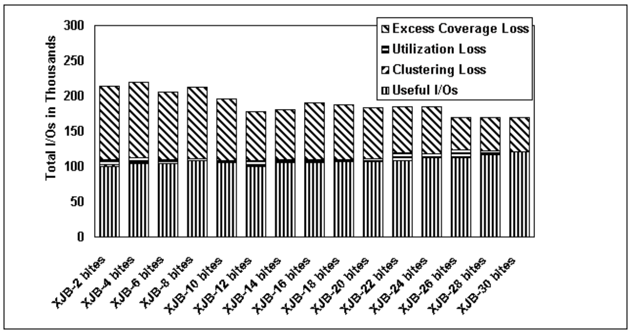


Figure 7: XJB Access Method Performance Losses in Number of Total Workload I/Os for various values of X

Bounding Predicate	BP Size
MBR	$2D$
MAP (2 MBRs)	$4D$
JB	$2D + D * 2^D$
XJB	$2D + D * X + X$

Table 2: Size of the array necessary to store the BP for each of our proposed R-tree variants (D = dimensionality of the data)

creation algorithms like that in Figure 6 are, however, currently too complex for use in dynamic AMs where BPs are created and modified frequently during insertion and deletion. In Section 6 we present a BP-independent algorithm for improving the performance of any AM in use on a dynamic data set.

5 Analysis of New Bounding Predicates

Before we begin with the results of the `amdb` analysis of the new access methods, we mention a couple of more traditional, domain-dependent analyses of the performance of the new BPs, to provide a more intuitive feel for their performance. Table 2 shows the size of all of the proposed BPs as a function of the data dimensionality. Also, our proposed BPs have been designed to reduce the hyper-volumes they contain; Table 3 shows that the new BPs have succeeded in reducing the total data volume enclosed by all the BPs in the AMs.[6] Note that comparing the perimeter of the various BPs would be uninteresting because the transformations MAP, JB and XJB perform upon MBRs do not change the perimeter of the resulting bounding polygons.

	5D Blob AM
R-tree	0.1038
SS-tree	1.1152
aMAP	0.0085
JB	0.0097
XJB	0.0097

Table 3: Sums of the hyper-volumes of all the leaf-level bounding predicates in the access methods.

We expected that the aMAP tree would be better than the R-tree, the XJB tree better than the aMAP tree, and the JB tree best of all when it came to performance losses, and worst in number of I/Os. Figures 8 and 9 show that our expectations were not quite accurate.

As shown in Figure 8 and 9 the aMAP tree performance is on par with the R-tree for this application. The aMAP tree performance metrics are better than R-trees at the leaf level, but worse at the inner nodes. This occurs because the aMAP BPs at the root level provide little benefit over the R-trees MBR; both effectively cover the data space. However, because the aMAP BPs are larger in bytes than a single MBR, there is greater fanout, hence more inner nodes for each query to check. Our workload accesses more total nodes, inner plus leaf, in the aMAP tree than the R-tree. Ironically, while the aMAP tree failed the goal of minimizing overall workload I/Os, it did meet the goal of fewer leaf level I/Os than the R-tree. This suggests that if the inner nodes were all in memory, the aMAP tree or a hybrid scheme that mixes single hyper-rectangle BPs and multiple hyper-rectangle BPs in the inner nodes may be a better choice than the R-tree. However, we shall see that there are BPs that are better than the aMAP BP.

Figure 8 shows, as we had hoped, that the leaf level excess coverage loss for JB was greatly reduced. The large size of the JB BPs increased the height of the tree from the R-tree height of 3 to a height of 6. Unexpectedly, the total workload

[6] Interestingly, the volumes of unit-radius hyper-spheres fit on a bell curve, with the five dimensional unit radial hyper-sphere having the maximum volume. [20]

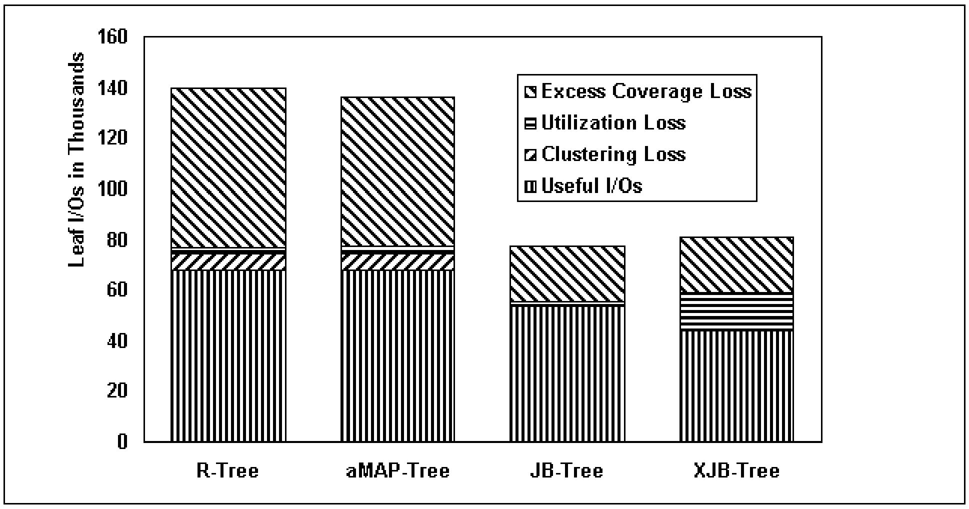


Figure 8: Access Method Performance Losses in Number of Leaf Level Workload I/Os: $X = 26$ for XJB

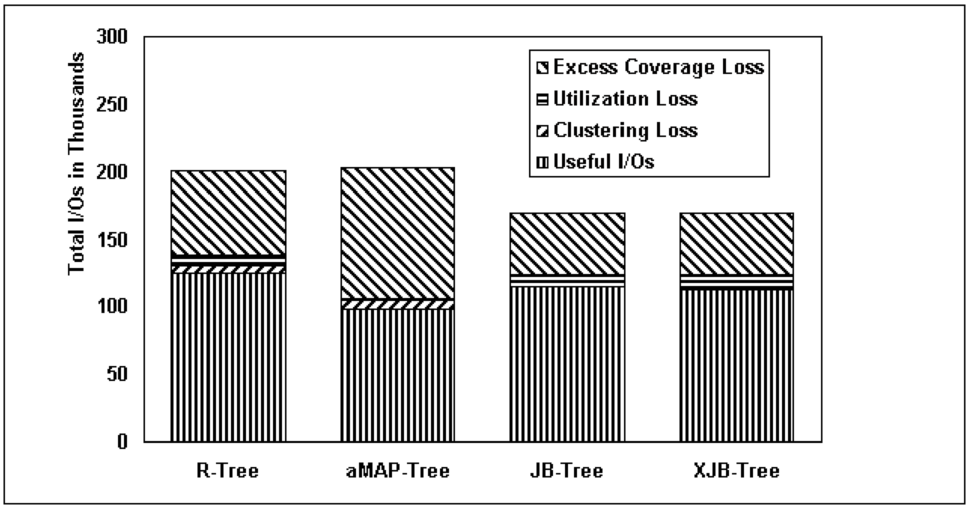


Figure 9: Access Method Performance Losses in Number of Total Workload I/Os: $X = 26$ for XJB

I/Os for the JB tree is less than that for the R-tree or the MAP tree. This justifies our intuition that MBR corners are the key problems for R-trees in nearest neighbor search.

For XJB, we performed experiments to find the smallest number of corner bites for which the workload I/Os were effectively as low as for the JB tree. Figure 7 shows this happens when $X = 26$. We found that the number of XJB leaf level I/Os was about equal to the number of leaf level I/Os JB performed, and both performed only slightly more than half the number of leaf level I/Os of the aMAP tree or R-tree.

We mentioned in Section 3 that we had to be certain that the AMs hit less than one fifteenth of the leaf level pages in order to compensate for the difference between random and sequential I/O costs for disk accesses. The `amdb` analysis revealed that, even without assuming that the inner nodes are in memory, none of our new AMs hit more than one in 50 of the AM total pages during query execution.

In spite of its height, for our query workload and static data set, the JB tree had the best performance characteristics, including total I/Os, of our AM designs. However, this analysis does not take into account memory buffer effects. XJB is likely to be more effective in the Blobworld system because its tree height can be lower than the JB tree height. Thus, the XJB inner nodes are more likely to fit in memory. Both JB and XJB exhibit performance characteristics on our data set and workload that are superior to R-trees, SR-trees and SS-trees.

6 Aggressive Insertion Policy

XJB and JB address the problem of excess coverage loss in an AM. We now turn to a BP independent algorithm that addresses utilization and clustering losses in dynamic AMs, which can not rely on sorting and bulk loading to reduce those losses as Blobworld did. We developed a new algorithm for AM insertion which we call the *Aggressive Insertion Policy* (AIP).

Height-balanced AMs differ in their BPs and in the associated methods (PICKSPLIT, PENALTY, UNION, CONSISTENT), but they share a greedy algorithm for data item insertion. Data insertion

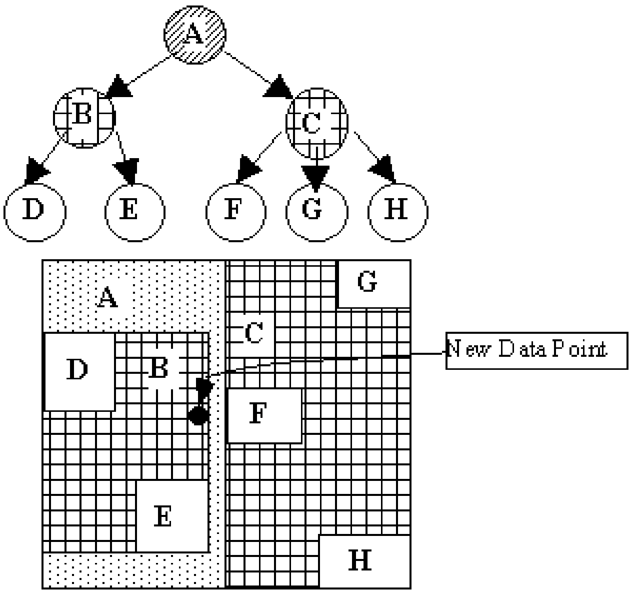


Figure 10: AIP Motivation: Illustrating the problem with the traditional "greedy" insertion. The new data point should be inserted into leaf F for the optimal data clustering. But if the root node only looks at its immediate children's bounding predicates, it will insert the new data point into branch B instead of branch C and result in a sub-optimal data clustering, with either D or E expanded to include the new point.

begins at the root node and at each level of the tree the insertion algorithm decides which of the clusters at that level the new data item belongs to based solely upon the PENALTY method from that level's BPs. Once that cluster is chosen, the algorithm proceeds down to the next level and decides on a sub-cluster, until it reaches the leaf level.

A potential problem with this approach is that leaf level clusters, the least likely to be stored in memory and hence the most important to retrieval performance, have little influence on the decision of which data items are placed into them. The insertion algorithm is "greedy", consulting only the current level for information at each step. This can be a problem because BPs become increasingly less precise descriptions of enclosed data at higher levels in the tree. Figure 10 illustrates this with an example. The new data point is inserted into B based only on a vague notion of how the BPs are lower down in the tree. However, to achieve the best choice of node at the leaf level (node F), node C should get the new point.

AIP focuses upon the BPs at the leaf level in making insertion decisions, essentially performing a nearest neighbor search to locate the appropriate leaf level node to insert a new data item into. AIP looks for a leaf level node with a globally minimum penalty metric[7] and inserts the new data item there. This keeps BP overlap at higher levels of the AM from affecting the data item placement, leading to tighter leaf-level BPs and reduced excess coverage loss. Like the R* AM, the AIP also performs reinsertion, which mitigates the impact of poor insertion order on the AM.

7 Aggressive Insertion Policy Experiments

We used both real data sets and synthetic data sets to validate AIP. For the real data sets, we use the Blobworld data set, two-dimensional Tiger geographic data of road maps of LongBeach and Montgomery Counties[8], and two-dimensional computational fluid dynamics data[9]. For the synthetic data sets, we generated both clustered and uniform 100K point data sets in a range of dimensions. For the query workloads, we generated synthetic range and nearest neighbor query workloads because traces of real query workloads were not available. For the nearest neighbor queries, we randomly chose 5% of the points in the data sets as the centers of queries returning 200 nearest neighbors. For the range queries, we similarly selected random sample points as the range centers and then randomly generated the width in each dimension for each query. Maximum widths were set so that range queries would average returning on the order of a couple hundred data items per query, largely so that the results would be comparable with the nearest neighbor query workloads.

Figures 11 and 12 show that AIP reduces total workload I/Os relative to the R* tree, most particularly at the higher dimensions, by up to 35% for nearest neighbor and range query workloads. AIP does particularly well on the larger data sets, like Blobworld, where it achieves a 35% improvement, and those with higher dimensionality. Experiments performed on a variety of two-dimensional data sets were inconclusive; the R* tree and the AIP AM each performed better on some of the two-dimensional sets. We hypothesize that at very low dimensions R* does well enough that AIP can not regularly improve over it. It is as the AM structure of a traditional AM deteriorates over many insertions that AIP becomes more and more attractive, and AM structure deteriorates most noticeably at high dimensions and data set sizes. The R* AM has a chance of choosing a suboptimal branch at each level of the AM. As AM height increases because of lower fanout at higher dimensions, it becomes more likely that the R* AM chooses a suboptimal branch at some level. AIP is indifferent to AM height. AIP is also BP independent; as an insertion policy it can be used in an AM with any type of BP to improve the AM performance.

The benefits of AIP in our experiments are relatively modest, at most about 35%. However, as shown in Figures 11 and 12, most of the performance loss in these experiments is due to excess coverage loss – the effect of the old-fashioned rectangular BPs we used. Note that AIP's nearest-

[7] In our experiments, the penalty metric is distance to BP center.

[8] Extracted from the US Bureau of Census TIGER database

[9] http://www.cs.du.edu/ leut/MultiDimData.html

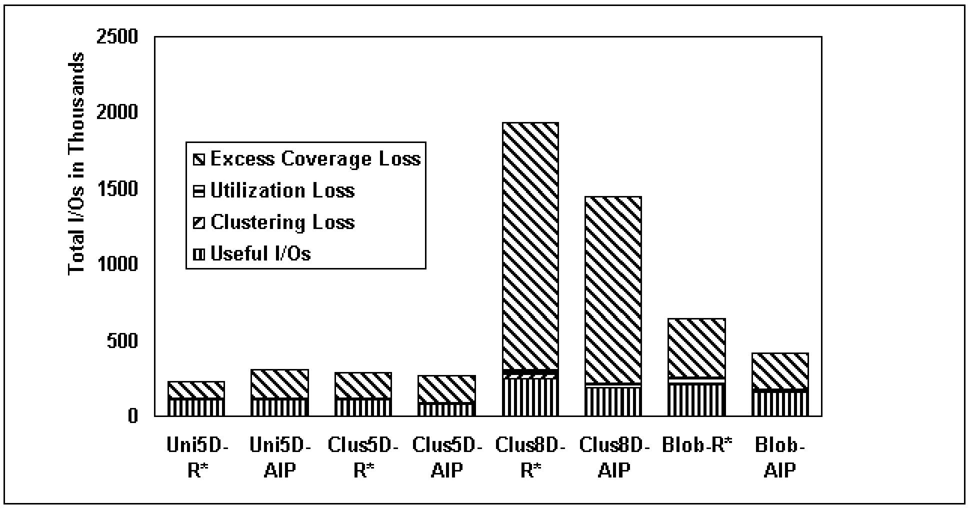


Figure 11: Total Nearest Neighbor Query Workload I/Os, AIP vs. R* Insertion-Loaded AMs

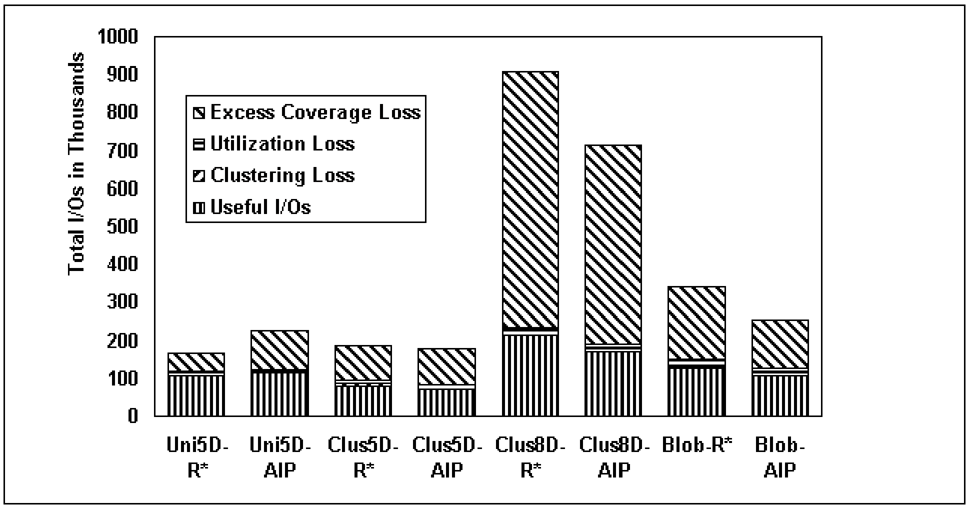


Figure 12: Total Range Query Workload I/Os, AIP vs. R* Insertion-Loaded AMs

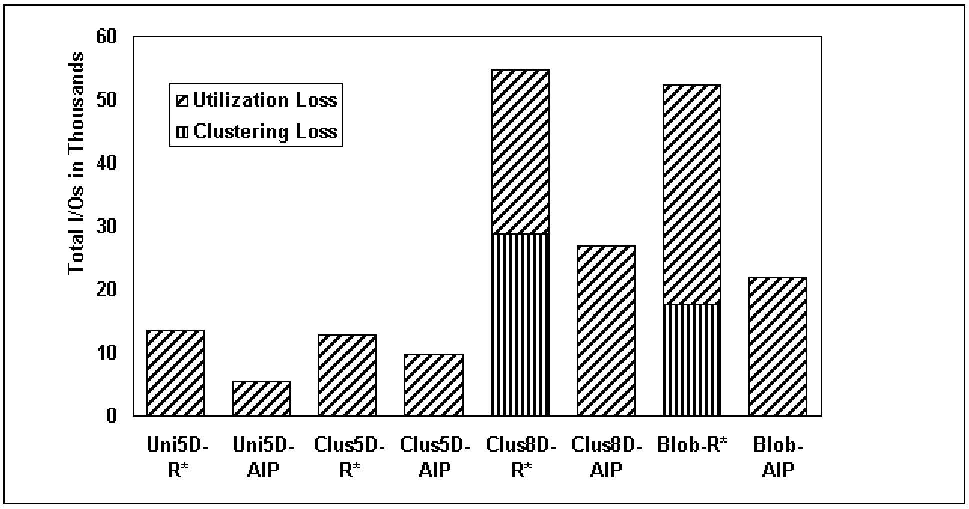


Figure 13: Total Nearest Neighbor Query Workload I/Os, AIP vs. R* Insertion-Loaded AMs, Clustering and Utilization Losses Only

neighbor insertion algorithm is not affected by the choice of BPs – poor bounding predicates may slow it down, but it always finds the nearest neighbor to the point to be inserted, and locates the new point on the same leaf. As a result, AIP's effects on clustering and utilization losses are captured in this experiment, and all that remains is the effect of the clustering on more sophisticated BPs. We believe that AIP combined with XJB would result in performance very close to optimal even for dynamic data. Unfortunately, we were unable to validate this result directly in time for submission. In the absence of that experimental result, we present Figure 13, which zoom in on just the clustering and utilization loss from Figure 11. Note that AIP is extremely effective at minimizing clustering and utilization loss (the two factors under its control): it does between 30% to 70% better than R*-trees on these metrics. The remaining performance factor – excess coverage loss – is controlled by the bounding predicates. Our Blobworld experiments suggest that XJB would remove most of the remaining clustering loss from these experiments, complementing the benefits of AIP. We will validate this experimentally in the final version of this paper.

8 Related Work

Much research has gone into tree-based multidimensional access methods; [8] surveys over 50 data structures for the two-dimensional case alone, and there has been a slew of more recent work as well ([21, 6, 5], etc). While [8] attempts to provide a partial ordering of many access methods based on their performance, the final comparison is inconclusive due to the wide variance in experiments performed in the earlier work. There are no widely-accepted benchmarks for multidimensional access method comparison. Therefore it can be difficult to put new access methods into context with related work, and we do not attempt to compare our techniques to all the indexes proposed in the literature. However, comparing our techniques to the approximately optimal indexes modeled by `amdb`, we see that we are doing very well, with our experimental results varying from .9 to 7× `amdb`'s optimum on both real-world and synthetic workloads, where the R* tree varies from .9 to 10×. We are not aware of such absolute metrics of success in prior work. Moreover, our focus on GiST-based designs has potential for practical impact, since concurrency and recovery schemes have been developed for GiST [14], and GiSTs have been implemented in at least one commercial DBMS [13].

9 Conclusions and Future Work

We began with optimizing an access method for a static real-world data set, that of the image retrieval system Blobworld. Using the tools provided by GiST and `amdb`, we analyzed the performance of several traditional access methods for nearest neighbor search in Blobworld. Finding that their bounding predicates led to unnecessary I/Os in the course of query execution, we designed three new bounding predicates and then analyzed the resulting new access methods. The crux of our new bounding predicates is that they remove volume from the corners of the bounding rectangles, where spherical queries are likely to intersect. We found that two of the three new access methods, JB and XJB, demonstrated significantly improved performance for our application.

We then considered indexing dynamic datasets, for which we present a new, bounding predicate independent insertion algorithm which considers "global" information when making insertion decisions. We compared the *Aggressive Insertion Policy* to the R* tree over a variety of data sets, and found that by using AIP, we can improve the query performance by up to 35%.

There are a number of future directions we could take this work.

- exploring the use of hybrid bounding predicates, more and less precise bounding predicates within the same access method
- finding a mathematical way to express, and a computationally efficient algorithm to compute, the "rectangle(s) that intersect with a minimal number of spheres whose centroids are outside the rectangle(s)" for use in building BPs optimized for nearest neighbor queries
- creating heuristics for rapid XJB creation, to make it possible to use XJBs in dynamic AMs
- designing algorithms to reorganize internal node pointers and bounding predicates in AIP AMs using more "global" criteria
- comparing AIP performance to traditional insertion algorithm performance over workloads that include varying mixes of queries and insertions, to quantify the trade-offs between query and insertion I/Os
- comparing AIP performance over AMs with different types of BPs

Acknowledgments

We thank Paul Aoki, Marcel Kornacker and Mehul Shah for support and implementing GiST and `amdb`. We thank Chad Carson for all the work he did to make our Blobworld experiments possible. We thank Kris Hildrum for co-creating the JB Problem NP Construction. Megan Thomas was supported by fellowship stipend support from the National Physical Sciences Consortium and Lawrence Livermore National Laboratory.

References

[1] P. M. Aoki. Generalizing "search" in generalized search trees. In *ICDE*, pages 380–389, Orlando, FL, February 1998.

[2] N. Beckmann, Kriegel H.-P., R. Schneider, and B. Seeger. The R∗-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[3] S. Belongie, C. Carson, H. Greenspan, and J. Malik. Color- and texture-based image segmentation using em and its application to content-based image retrieval. In *Proc. of the Sixth International Conference on Computer Vision*, January 1998. http://www.cs.berkeley.edu/~carson/papers/ICCV98.html.

[4] S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high dimensional data space. In *Proc. 16th ACM Symposium on Princ. of Database Systems (PODS)*, pages 78–86, 1997.

[5] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid technique: Towards breaking the

curse of dimensionality. In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, pages 142–153, Philadelphia, PA, June 1999.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. of the 22nd VLDB Conference*, pages 29–39, Mumbai (Bombay), India, 1996.

[7] C. Carson, M. Thomas, S. Belongie, J. M. Hellerstein, and J. Malik. Blobworld: A system for region-based image indexing and retrieval. In *Proc. of the Third Annual Conference on Visual Information Systems*, pages 509–516, Amsterdam, 1999.

[8] V. Gaede and O. Guenther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):381–399, 1998.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, 1984.

[10] J. M. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in the vlsi domain. In *Proc. of the ACM/IEEE 34th Design Automation Conference*, 1997.

[12] N. Katayama and S. Satoh. The SR-tree: An index structure for high dimensional nearest neighbor queries. In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, pages 369–380, Tucson, AZ, May 1997.

[13] M. Kornacker. High-performance generalized search trees. In *Proc. Proc. 25th VLDB Conference*, pages 699–708, Edinburgh, Scotland, September 1999.

[14] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search tree. In *Proc. of the ACM-SIGMOD International Conference on Management of Data*, pages 62–72, Tucson, AZ, May 1997.

[15] M. Kornacker, M. Shah, and J. M. Hellerstein. An analysis framework for access methods. Technical Report UCB//CSD-99-1051, University of California at Berkeley, 1999.

[16] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: A simple and efficient algorithm for r-tree packing. In *Proc. of the 12th International Conference on Data Engineering*, pages 497–506, New Orleans, LA, April 1997.

[17] E. Riedel. A performance study of sequential i/o on windows nt 4. In *Proc. of the 2nd USENIX Windows NT Symposium*, Seattle, WA, 1998.

[18] M. Shah, M. Kornacker, and J. M. Hellerstein. `amdb`: A visual access method development tool. In *Proc of User Interfaces for Data Intensive Systems (UIDIS)*, Edinburgh, 1999.

[19] M. Thomas, C. Carson, and J. M. Hellerstein. Creating a customized access method for blobworld. Technical Report UCB//CSD-99-1081, University of California at Berkeley, 1999.

[20] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics*. CRC Press, November 1998.

[21] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of the 12th IEEE Int'l Conference on Data Engineering*, pages 516–523, New Orleans, LA, February 1996.