Creating a Customized Access Method for Blobworld

Megan Thomas, Chad Carson and Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
{mct, carson, jmh}@cs.berkeley.edu

ABSTRACT

We present the design and analysis of a customized access method for the content-based image retrieval system, Blobworld. Using the `amdb` access method analysis tool, we analyzed three existing multidimensional access methods to support nearest neighbor search in the context of the Blobworld application. Based on this analysis, we propose several variants of the R-tree, tailored to address the problems the analysis revealed. We implemented the access methods we propose in the Generalized Search Trees (GiST) framework and analyzed them. We found that two of our access methods have better performance characteristics for the Blobworld application than any of the traditional multi-dimensional access methods we examined. Based on this experience, we draw conclusions for nearest neighbor access method design, and for the task of constructing custom access methods tailored to particular applications.

1 Introduction

Millions of images are now available to users, in proprietary databases and on the Internet. As a consequence, content-based image searching applications are multiplying, as are the number of images they are expected to handle. For example, the University of California at Berkeley Digital Library has around 470 GB of images in its on-line collection [18]. One paper estimated that there is 1 TB of images on the WWW [9]; other estimates place the amount of image data on-line several orders of magnitude higher.

Users in many domains, like medical imaging, weather prediction, and TV production, now retrieve images from their collections based upon the contents of the images. The ability to efficiently execute content-based image queries becomes more important as the image databases grow. The Blobworld system [2] addresses content-based querying by breaking the images into "blobs" of homogeneous characteristics, and searching for images by specifying the characteristics of the blobs in the desired images.[1]

A full Blobworld query, which examines the entire data set, must perform computationally complex comparisons of the high-dimensional feature vectors of all the blobs in all the images (currently there are 221321 blobs in 35000 images) in order to answer each query. This is not a scalable approach, so access methods (AMs) are required to speed up the queries. This paper focuses on the AMs we studied and developed for the Blobworld system.

We implemented our AMs in the Generalized Search Tree (GiST) framework [12]. GiST lets AM designers 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 application.

`Amdb` [20] is a tool for the visualization, profiling and debugging of AMs implemented in GiST. In addition, `amdb` incorporates an AM analysis framework [15]. `Amdb` access method analysis takes as input a GiST AM, a data set and a workload (set of queries) and outputs a set of metrics characterizing the

[1] The reader is invited to examine Blobworld at `http://elib.cs.berkeley.edu/photos/blobworld/`.

observed performance of the access method, relative to an idealized tree. This allows the AM designer to find areas for potential access method improvement.

By analyzing the performance of traditional AMs on Blobworld queries using `amdb`, we identified those characteristics of the AMs that contributed the most to AM inefficiency. The modularity of the GiST framework allowed us to easily create new AMs to address the performance problems `amdb` had pointed out. We returned to `amdb` in order to analyze the performance of the new AMs and verify that they did provide improvement over the old AMs.

Section 2 provides background information on GiST, `amdb` and Blobworld. The analysis process begins in Section 3, where our experimental framework is set up. Section 4 presents the analysis of the performance of a few standard access methods. We present our new AM designs, derived from intuitions gained analyzing the traditional AMs, in Section 5. Section 6 analyzes the performance of the new AM designs. Section 7 covers related work. In Section 8 we discuss opportunities for future work, and the conclusions of our study.

2 Background

2.1 GiST

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 [10] 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; INSERT, which adds a *(key, RID)* pair to the tree; and DELETE, which removes such a pair from the tree. It 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.

We are most interested in SEARCH, because it dictates query behavior. In order to find all leaf entries satisfying a search predicate, SEARCH recursively descends *all* subtrees for which the parent entry's predicate is consistent with the search predicate (employing the user-supplied extension method *consistent()*).

While the domain specific characteristics of the indexed data, such as size and shape, are part of the input, domain specific characteristics of the BPs are parameters of the AM design and considerably influence performance. 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.

Metric	Concept
Excess Coverage Loss	unnecessary I/Os due to queries accessing leaf nodes containing no relevant data as a result of inaccurate BPs
Utilization Loss	unnecessary I/Os due to the node storage utilization deviating from target utilization
Clustering Loss	unnecessary I/Os due to the difference between optimal and achieved leaf-level data clustering

Table 1: `Amdb` Leaf-Level Analysis Metrics

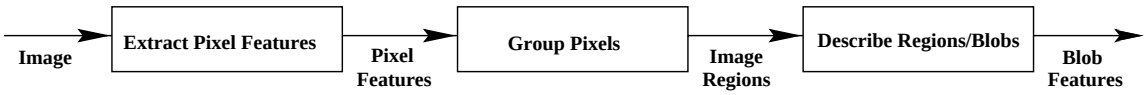

Figure 1: The Stages of Blobworld Processing: from pixels to blob descriptions

2.2 `Amdb`

`Amdb` [15, 20] 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 which 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 lost relative to the 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.

The performance metrics related to the nodes of the AM and its structure are shown in Table 1. *Excess coverage* is a measure of the effect of a BP covering more of the data space than is necessary to represent the data contained in the subtree. A BP exhibiting excess coverage causes queries to visit pages other than those in which matching data is stored. *Utilization loss* occurs when the the storage utilization of nodes is less than a given target utilization. The idea 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 in an optimal AM. This captures the performance loss that occurs when data that will be returned in the same queries are not located in the same node. The "optimal" clustering for a set of queries is found in `amdb` by using hyper graph partitioning [13]. Truly optimal clustering is NP-hard, but this heuristic works well in practice.

We use `amdb`'s metrics and visualization capabilities to gain intuitions about how to build a custom AM for Blobworld. The exact details of the metrics are beyond the scope of this paper. For a thorough description of the `amdb` analysis metrics, as well as the equations through which their values are calculated, see [15]. After implementing the AMs we design in GiST, we verify their quality using the `amdb` analysis metrics.

2.3 Blobworld

Blobworld is a system for image retrieval based on finding coherent image regions which roughly correspond to objects. Each image is automatically segmented into regions ("blobs") with associated color and texture descriptors. The "blobs" generally correspond to objects or parts of objects. The segmentation algorithm is fully automatic; there is no parameter tuning or hand pruning of regions. Querying is based

Figure 2: The Stages of a Blobworld Query: from query to answer

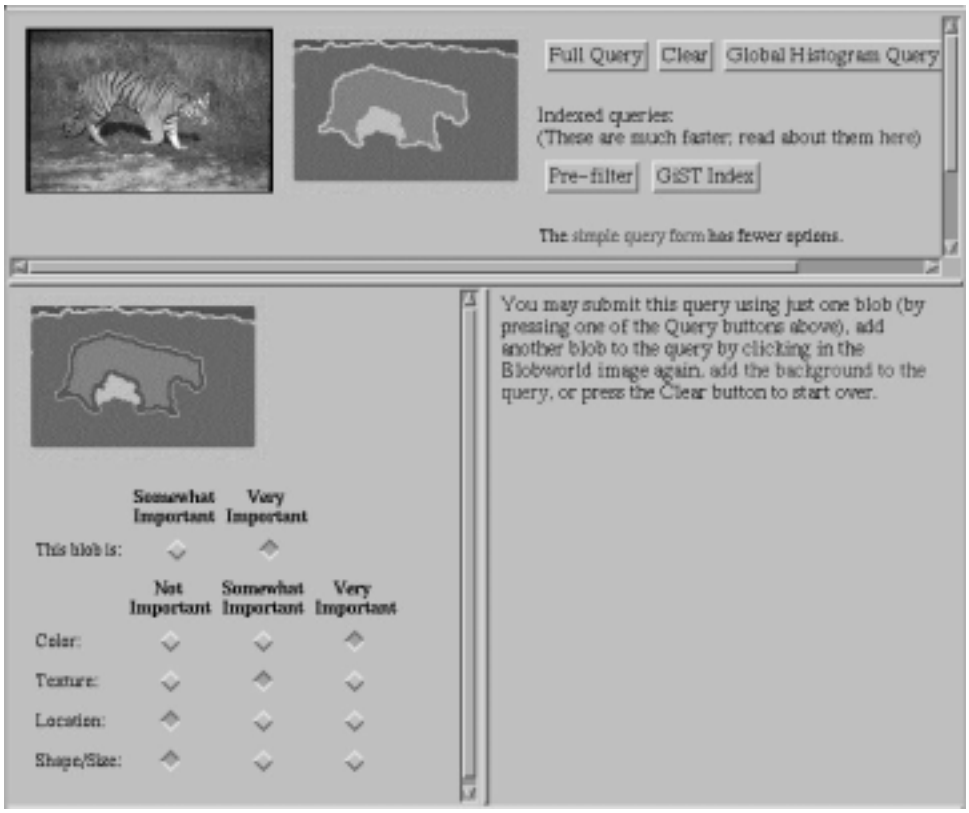

Figure 3: A Sample Blobworld Query

on the attributes of one or two regions of interest, rather than a description of the entire image. This keeps image regions unrelated to the query regions from affecting the query results.

The Blobworld representation is related to the notion of photographic or artistic scene composition. 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.

To summarize Blobworld image pre-processing (Figure 1), pixel features are extracted, the pixels are grouped into blob regions with homogeneous characteristics, and the feature vectors of the blobs are extracted. Details of the image pre-processing algorithms may be found in [2]. In this paper, we will focus on indexing the color feature vectors.

A user of the Blobworld system begins with a sample image and the interface in Figure 3. The user selects the blob she is interested in from the image and sets the weights for the various characteristics. ("Color is very important, location is not, texture is so-so...") As shown in Figure 2, this query is passed to an access method, which selects a few hundred images containing blobs that are closest to the query blob and passes these images to the Blobworld code for ranking based on the full feature vectors. Note

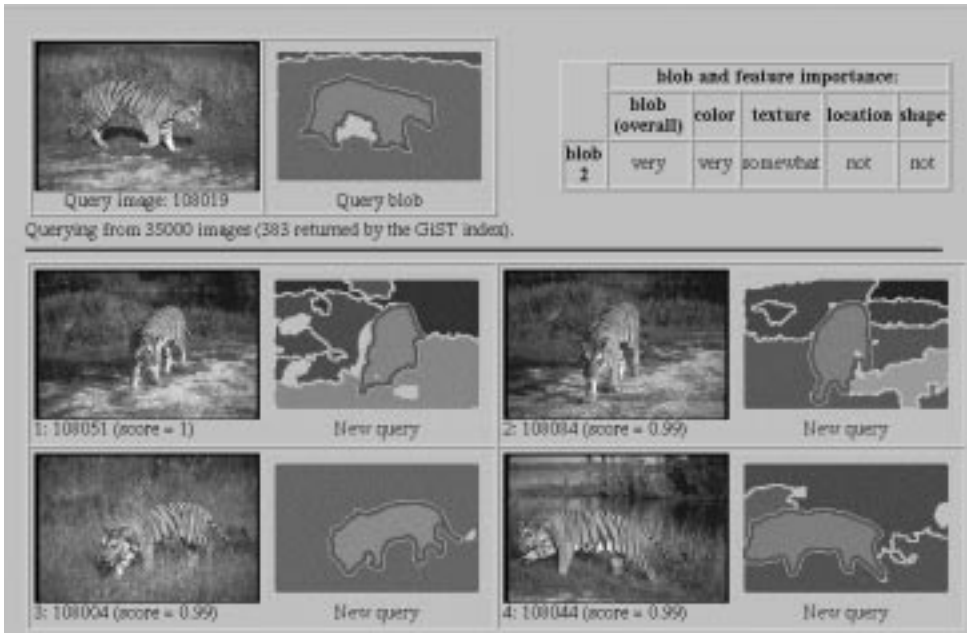

Figure 4: Sample Blobworld Query Results

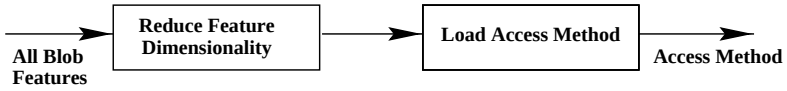

Figure 5: The Stages of Access Method Creation: from blob descriptions to an access method

that the AM does not necessarily choose the nearest few hundred neighbors that would be found by the full Blobworld ranking of all images; it is a "quick and dirty" estimate of the top few hundred, from which Blobworld chooses the top few dozen to present to the user. The goal of the AM is to get the top few dozen Blobworld would select into the top few hundred that the AM selects. Figure 4 is an example of Blobworld query results.[2]

3 Experimental Framework

The goal of our access method design is to speed up Blobworld queries as much as possible. We begin by examining the results of Blobworld queries that return varying numbers of images, and AMs that store color feature vectors with varying dimensionality. Lower data dimensionality increases tree fanout and reduces tree height. Retrieving fewer images from the AM code reduces the number of necessary AM page accesses and the costs for the final ranking that Blobworld performs. However, reducing data dimensionality or the number of images retrieved also reduces the number of images in the AM result set that match those in the full Blobworld query result set, a number we want to maximize. Therefore, we seek to simultaneously minimize the data dimensionality in the AM, and the number of images an AM query may return while still giving the user nearly the same results she would get from a full Blobworld query over all the images.

The full image feature vectors have 218 dimensions, which is typically too many dimensions to index

[2] See [5] for an end-to-end overview of the Blobworld querying system.

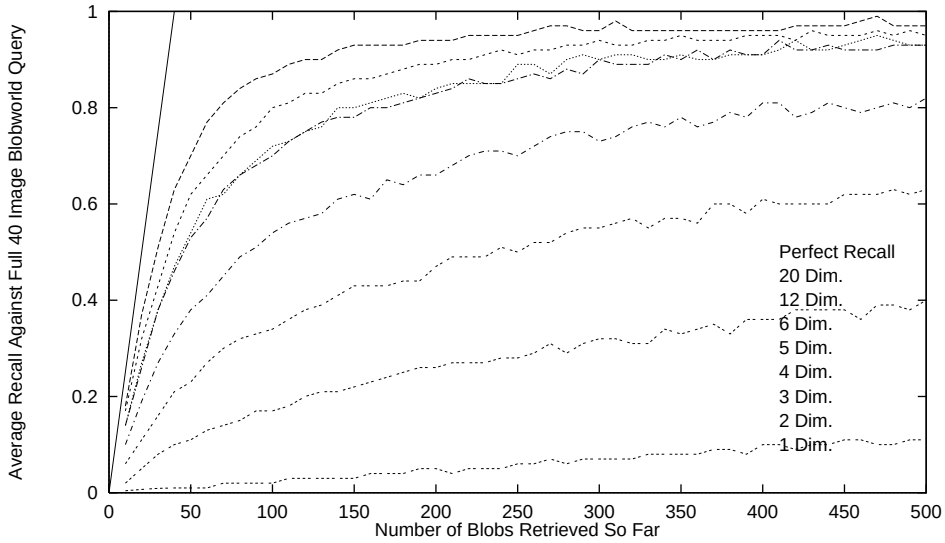

Figure 6: Comparison of Recall of Varying Data Dimensionality Queries vs. Full Blobworld Queries: Recall is computed against the top forty images returned by a full Blobworld query; there are 221231 blobs in the database. Note that the recall of the low dimensional queries strictly improves with increasing dimensionality of the data; the 20D is highest and 1D curve is lowest. The queries over 5D and 6D data have nearly identical recall; adding one more dimension beyond five produces negligible improvement in query recall.

effectively for nearest neighbor queries [6]. Therefore, Singular Value Decomposition is performed to reduce the dimensionality of the blob feature vectors, as suggested, for example, in [11] and [6]. The resulting vectors are truncated to include only the most significant dimensions. Figure 6 shows the average query recall of 5531 nearest neighbor queries over relatively low dimensional feature vectors, compared to the results of the same queries run using Blobworld query processing of the full 218 dimensional vectors. The dimensionality of the lower dimensional feature vectors ranges from one to twenty. Not surprisingly, the more images the low dimensional query returns, the closer the results of the low dimensional query match the results of a full Blobworld query.

The recall curves rise sharply up to the five dimensional data curve. Adding further dimensions to the data set does not significantly improve the quality of the final results. We have found that the query answer quality resulting from queries over five-dimensional SVD vectors returning 200 nearest neighbors is sufficient to satisfy the Blobworld designers. Therefore, for the rest of the paper we will use five-dimensional data vectors and query workloads consisting of nearest neighbor queries that retrieve 200 images each.

3.1 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,[3] and hence are typically based on one of the eight sample images in that page.

Thus, we felt an artificial workload would better stress the access method design. Moreover, we needed a broad query workload, since the efficacy of the `amdb` analysis rests on the premise that the query workload covers the data set. If a data item is never accessed by a query, `amdb` will have no means to determine how to properly place it in the optimal clustering, which will reduce the validity of the optimal clustering `amdb` uses as a basis for calculating performance losses.

[3] `http://elib.cs.berkeley.edu/photos/blobworld/`

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

Table 2: Performance Losses in R-Trees

We randomly selected around 5531 blobs from the data set of 221231 blobs (35000 images) to serve as foci of the nearest neighbor queries in our query workload. This is enough queries so that every blob in the data set should, on average, be retrieved by several queries. We used `amdb` to analyze the performance of this query workload for each type of tree we analyzed.

3.2 Beginning the Design

Having decided upon our data set and query workload, we aimed to design the most efficient index we could, where the metric to minimize is leaf-level I/Os, since the inner nodes of the index can reasonably be assumed to be in memory. The tradeoff between leaf level and total I/Os is examined in more detail in Section 6.

To be worthwhile, 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, access method I/Os can be around 15× slower than sequential scan I/Os.[4] So the AM must not hit more than one fifteenth of the leaf-level pages in the AM.

Blobworld image processing is compute-intensive and time consuming, so it is done via off-line, batch processing. Therefore, 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 index trees.

4 Standard Index Alternatives

We used the STR algorithm [16] to sort the data for bulk-loading into an R-tree [10]. Using `amdb`, we found that using STR minimized the utilization and clustering loss over our query workload, leaving the largest performance losses for the R-tree in excess coverage, as shown in Table 2. Essentially, the only problem with a bulk-loaded R-tree is its sloppy BPs.[5] We also tried bulk-loading two other traditional AMs for multi-dimensional data, the SR-tree [14] and the SS-tree [21], to see if different bounding predicates helped performance. SS-trees use spherical BPs and SR-trees use minimum bounding rectangles plus bounding spheres.

Figure 7 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. As Figure 8 shows, the SS-tree performs more unnecessary leaf level 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 leaf level excess coverage loss relative to the R-tree.[6] However, for both R-trees and SR-trees, about 30

[4] Using Seagate Barracuda ultra-wide SCSI-2 drives, [19] measures a throughput of 9MB/s under Windows NT. The average seek time and rotational delay for this drive are 7.1 ms and 4.17 ms, respectively. For 8KB 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.

[5] While R*-trees [1] are considered better than R-trees, bulk-loading the data eliminates any difference between the two AMs.

[6] When performance losses for inner and leafs nodes of the index trees are considered, the SR-tree has higher excess coverage loss than the R-tree.

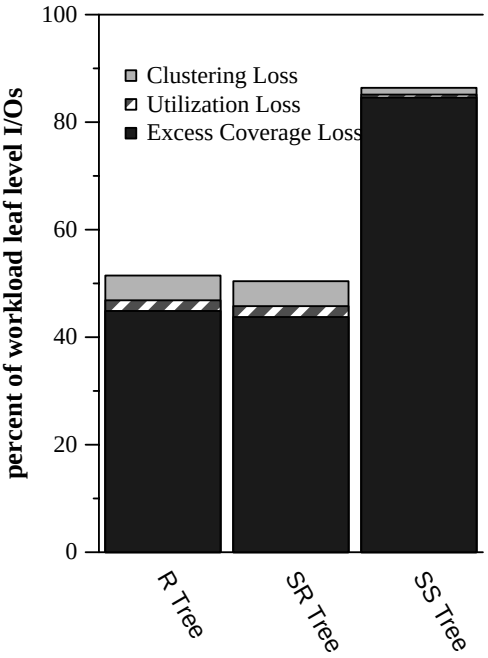

Figure 7: Access Method Performance Losses Relative to Total Workload Leaf Level I/Os: This shows the percent of the leaf level I/Os that were due to excess coverage, utilization and clustering loss.

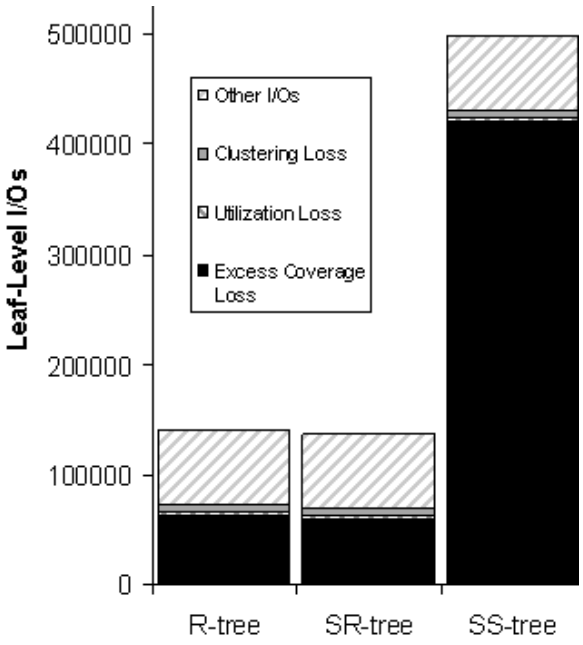

Figure 8: Access Method Performance Losses in Number of Leaf Level I/Os: How many leaf level I/Os during workload execution were due to the excess coverage, utilization or clustering loss?

percent of the I/Os are due to excess coverage loss. We hypothesize that the relatively poor performance of the SR and SS trees is an artifact of the STR sorting algorithm, which attempts to organize the data into hyper-rectangular tiles, rather than hyper spherical regions.

Excess coverage loss is a result of bounding predicates that indicate that relevant data may be in a leaf node when it is not. To reduce excess coverage loss, the index needs more restrictive BPs. Ideally, we would like bounding predicates that completely minimize false leaf node hits, while also minimizing the total number of I/Os for the query workload. A BP that simply lists all of the leaf node contents would meet the first of these criteria, at a very high cost in terms of BP size. However, we can not use a BP that is prohibitively large, since larger BPs lead to lower tree fanout, taller trees, and more I/Os. In the next section we focus on better BPs for nearest neighbor queries which strike a balance between size and precision of data description.

5 Towards a Better Blobworld AM

An understanding of how nearest neighbor queries interact with BPs is necessary in order to design a better access method bounding predicate. Nearest neighbor queries [3] work by finding points within a given distance of the query point, in essence asking expanding sphere queries. Figure 9 depicts a rectangular BP and the circles of nearest neighbor queries. If the query point is in the middle of a BP, it does not matter how small the BP volume is; this node will be hit. 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 the two topmost queries in Figure 9, must be minimized.

To minimize the ability of outside spheres to impinge upon a BP, two hyper-rectangles, instead of one, could be stored as the BP. Examination of the R-tree revealed that there was enough room in the

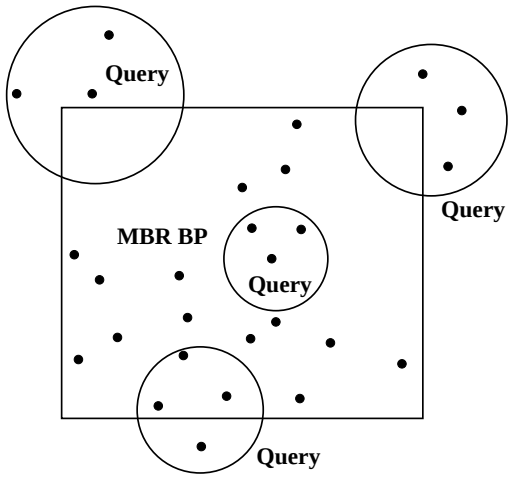

Figure 9: Nearest Neighbor Queries And Rectangular BPs

root node to accommodate more inner nodes at the second level, without expanding the height of the tree. The root node had only 24 children, and space for about 80. Therefore, the size of the inner node BPs could become larger without incurring the heavy penalty of increasing the number of levels in the index tree.

Figure 10 visualizes some individual 2D R-tree nodes (because 5D data can not be visualized) in `amdb`, showing the points they contain and the minimum bounding rectangle (MBR) BPs. The data points of some leaf nodes do not fill their MBRs, but leave noticeable gaps at corners of the MBRs. This 2D visualization suggests that queries may incur I/Os while checking the contents of the empty corners of the MBR, as the two top queries in Figure 9 do. Combined with our intuition about the importance of reducing spherical intersections with BP regions near the 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.

5.1 MAP

The first alternate BP we consider is the MAP (Minimum Area Predicate), 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 volume they enclose is minimal. Volume is not precisely what we want to minimize, but it is the heuristic used in a number of earlier AMs [10, 1]. 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 problem of finding two rectangles to bound a set of data points has been addressed in 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 may actually be beneficial. Using the heuristics developed for R-Tree node splitting would be counterproductive in this situation.

For each leaf node of data points, the idealized MAP tree 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 complexity of trying every possible partition of the data points is prohibitive. Approximating MAP became necessary. Instead of trying every possible pair of MBRs, aMAP (approximate MAP) tries

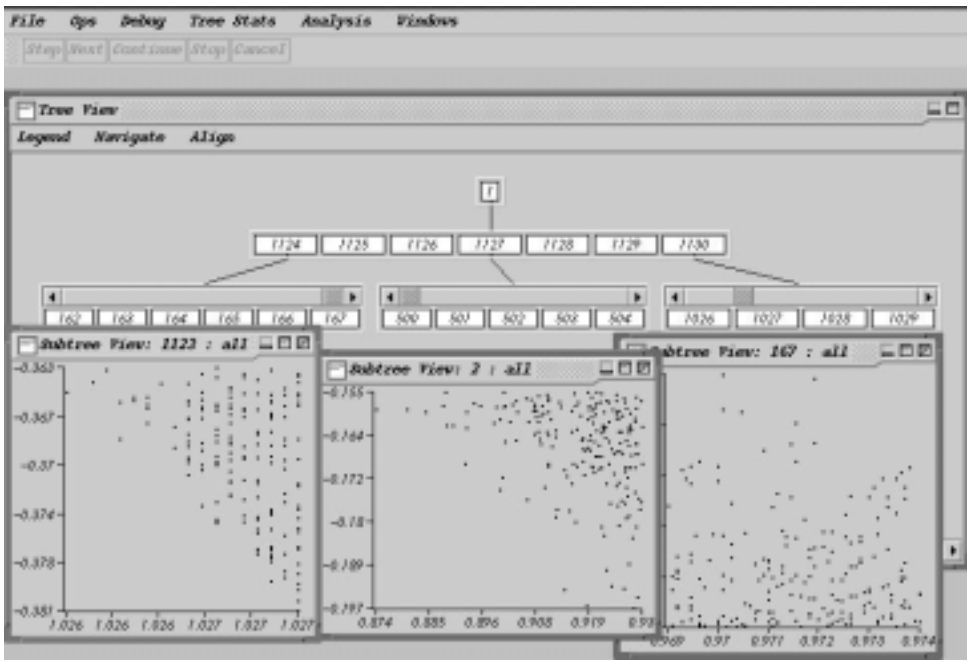

Figure 10: `amdb` Visualization of 2D R-Tree Leaf Nodes

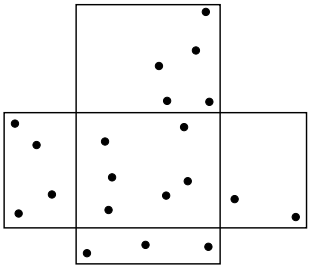

Figure 11: A MAP BP

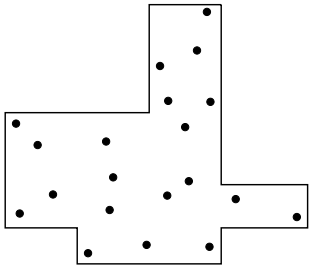

Figure 12: A Jagged Bites BP

1024 randomly selected pairs of sets and picks the minimum total volume MBR pair out of those it examines to be the node BP.

5.2 JB

Consider the points in the rectangle in Figure 9. The MAP BP in Figure 11 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 this figure. This would be less problematic in a BP that stored the MBR and the largest possible rectangular "bite" taken out of each corner. This type of BP, which we call a "Jagged Bites" BP (JB), would describe the data in a node more precisely than a MAP BP. The JB BP, pictured in Figure 12 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 does not intersect any MBR hyper edge. An algorithm for creating a JB BP out of a combination

```
1. Given the MBR
2. For each dimension d, create sorted[d], a list of the data points ordered along d
3. For each corner of the MBR
    (a) Set bite point to current corner point of the MBR
    (b) Set stopped_in_dim to equal zero
    (c) Set how_far_nibbled[dimension] to all zeros
    (d) Set done[dimension] to all false
         i. While stopped_in_dim is less than the number of dimensions
        ii. For each dimension d
            A. If not done[d]
            B. Increment how_far_nibbled[d]
            C. Construct bite by setting bite[d2] = sorted[how_far_nibbled[d2]] for each
               dimension d2
            D. If any points are between bite and this corner of the MBR, decrement
               how_far_nibbled[d], increment stopped_in_dim and set done[d] = true
            E. Else set max_bite[this corner] equal to bite
4. Store the MBR and the set of max_bites as the JB BP
```

Figure 13: 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.

of projections of the data points to the dimensional axes is in Figure 13.[7] This heuristic checks squarish bites by simultaneously "nibbling" off the next projected value in each dimension until the presence of a data point has stopped the nibbling in every dimension.

Unfortunately, the total number of corners in a hyper-rectangle is 2^D, so storing a point for each corner in a hyper-rectangular BP adds $D \times 2^D$ numbers to the size of the stored BP. For data in even a modest number of dimensions, this should make the JB BP too large.

5.3 XJB

An alternative, called XJB for "Top X Jagged Bites," is to simply store the X largest bites, leaving the remaining MBR corners "unbitten". X should be set to be as large as possible without causing the index to add another level. Note that picking the bites with the largest volumes is only a heuristic approximation of how bites should be selected. Ideal bites depend on the query workload and would be the bites that minimize the number of queries incorrectly impinging into the BP from outside of it; i.e., the ideal bites would minimize excess coverage loss.

Storing the X largest bites for each BP adds $(D + 1) \times X$ numbers 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.) Table 3 shows the size of all of the proposed bounding predicates as a function of the data dimensionality.

All three of the BPs we propose involve somewhat more complex code for index searching. The

[7] There is an efficient algorithm for constructing a better JB BP. However, we did not have time to implement and analyze it, so we do not present it here. The performance of the JB BP presented here is a lower bound on the better algorithm, and is still quite effective (Section 6). In the final version of the paper we will implement the improved algorithm.

Bounding Predicate	BP Size
MBR	$2D$
MAP (2 MBRs)	$4D$
JB	$(2 + 2^D)D$
XJB	$2D + (D + 1)X$

Table 3: Size of the array necessary to store the BP for each of our proposed R-tree variants (D = dimensionality of the data)

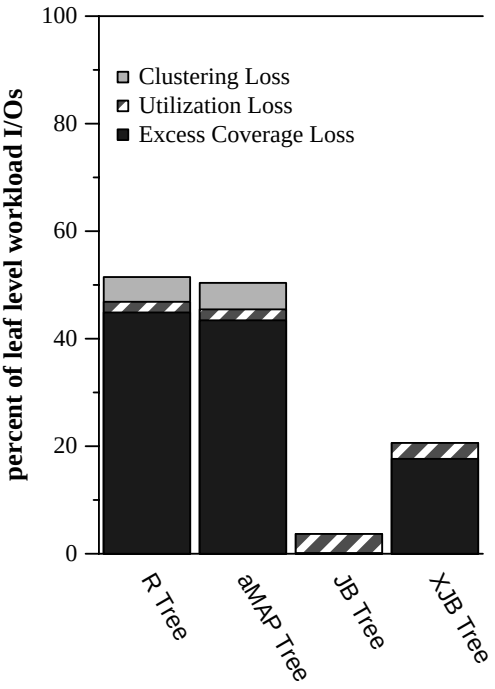

Figure 14: Access Method Performance Losses Relative to Workload Leaf Level I/Os: $X = 10$ for XJB

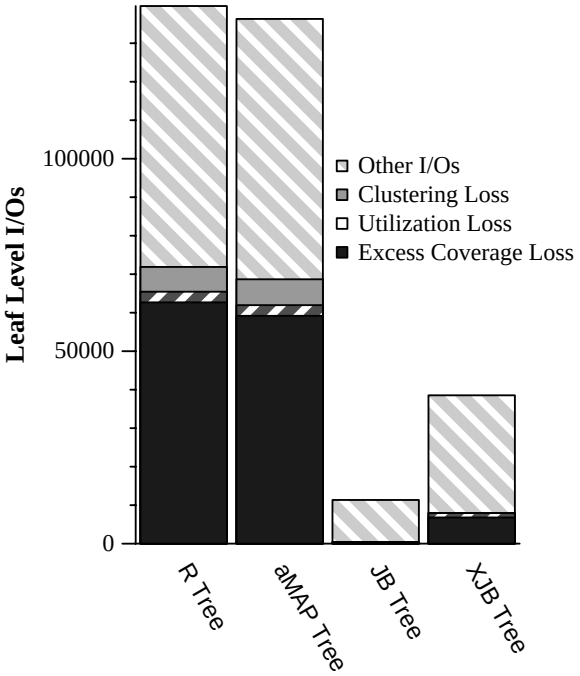

Figure 15: Access Method Performance Losses in Number of Leaf Level I/Os: $X = 10$ for XJB

distance and containment functions that form the backplane of the R-tree nearest neighbor algorithms are more complicated for 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.

6 Analysis of Alternatives

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 14, 15 and 16 show that our expectations were not quite accurate.

As shown in Figures 14 and 15 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

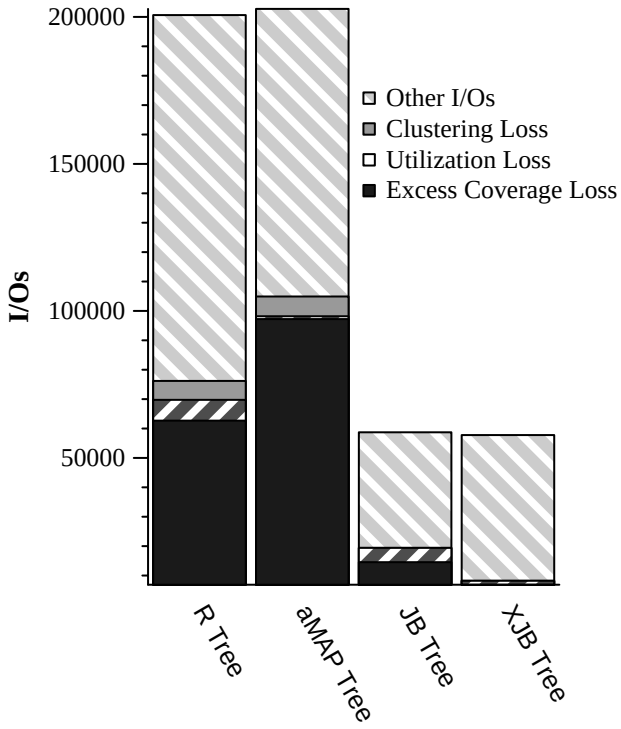

Figure 16: Access Method Performance Losses in Number of Total Workload I/Os: $X = 10$ for XJB

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 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 15 shows, as expected, that the leaf level excess coverage loss for JB was negligible. The large size of the JB BPs increased the height of the index tree from the R-tree height of 3 to a height of 6. Unexpectedly, the total number of workload I/Os for the JB tree is less than that for the R-tree or the MAP tree. The JB BPs in the inner nodes are so effective at filtering index searches to the correct child nodes that queries average barely more than two leaf level I/Os each. Each leaf node stores between 100 and 200 data points; with each query asking for 200 points, they can not do better than two leaf level I/Os. This justifies our intuition that MBR corners are the key problems for R-trees in nearest neighbor search.

For XJB, we chose $X = 10$, because the XJB tree grew another level in height for larger values of X. Lower values of X demonstrated worse workload performance. We found that XJB leaf level I/Os, while higher than those for JB, were still less than half the number of leaf level I/Os of the aMAP tree or R-tree. The larger size of the XJB BP caused the tree height to grow to 4. However, the XJB BP proved to be more effective than an MBR or MAP BP at reducing unnecessary leaf level accesses. While not as effective as the JB BP, the XJB BPs were small enough to keep the XJB tree two levels shorter than the JB tree, thereby keeping the XJB tree inner node accesses low enough that XJB tree performance was on par with JB tree performance.

We mentioned in Section 3.2 that we had to be sure 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 on modern disks. The `amdb` analysis revealed that, even without assuming that the inner nodes are in memory, none

of our AMs hit more than one in 50 of the AM total pages during query execution.[8]

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 is 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.

7 Related Work

Much research has gone into dimensionality reduction [11] and new index trees [8] to cope with high dimensional data. Most of the differences between the many AMs that have been proposed lie in their insertion and deletion algorithms, which do not affect our static, bulk-loaded data set.

The X-tree [4] is an index structure for high dimensional data using MBR BPs and variable sized inner nodes. It is designed to minimize the overlap of BPs, which we are able to accomplish more effectively by using STR to tile the data space. Moreover, the variable length pages in X-trees present significant difficulties for disk layout and buffer management in system implementation. Therefore, we did not include the X-tree in our experiments.

Most work on image indexing to date [7] has focused on indexing the entire image or user-defined subregions, not on indexing automatically created image regions. One exception to this trend is [17], which uses wavelet-based image region descriptions. However, the focus of that work is on the wavelet descriptions; they index the image regions using a simple R$*$ tree.

The high dimensional indexing work to date has focused upon creating indices for high dimensional data in general. The problem of taking a single application and multidimensional data set, then tailoring an access method just for them has not been addressed to our knowledge.

8 Future Work and Conclusions

Beginning with the Blobworld system, and the tools provided by GiST and `amdb`, we analyzed the performance of several traditional access methods for nearest neighbor search in image retrieval. 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 three 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.

With the access method implementation and analysis capabilities provided by GiST and `amdb`, and based upon our experience with Blobworld, we believe it is now practical for access method designers to create access methods customized to match the specific needs of their data sets and query workloads. Because of the ever-proliferating amounts of data and applications now available, we believe that customized access methods will be both feasible and increasingly important to applications.

There are a number of directions we would like to take this work.

- designing and implementing insertion and splitting algorithms for XJB and JB
- testing aMAP, JB and XJB on other data sets, and workloads both static and dynamic
- 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

[8] The MAP AM hit about 1 in 52 pages.

- designing a means for the best X to be automatically selected by the XJB algorithm

Acknowledgments

We thank Paul Aoki, Marcel Kornacker and Mehul Shah for support, implementing GiST and `amdb`, and thoughtful conversations.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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*, Mumbai (Bombay), India, 1996.

[5] 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.

[6] C. Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Publishers, Boston/London/Dordrecht, 1996.

[7] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective query by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

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

[9] S. Gribble and E. A. Brewer. System design issues for internet middleware services: Deductions from a large client trace. In *Proc. of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[10] 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.

[11] J. Hafner, H. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:729–736, July 1995.

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

[13] 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.

[14] 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.

[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] A. Natsev, R. Rastogi, and K. Shim. WALRUS: A similarity retrieval algorithm for image databases. In *SIGMOD*, Philadelphia, PA, 1999.

[18] G. Ogle, June 1999. http://elib.cs.berkeley.edu/arch/data_stats.html.

[19] 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.

[20] M. Shah, M. Kornacker, and J. M. Hellerstein. `amdb`: A visual access method development tool. To appear, User Interfaces for Data Intensive Systems (UIDIS), 1999.

[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.