Creating a Customized Access Method for Blobworld

Megan Thomas
Computer Science Division
University of California, Berkeley
mct@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 analyze three existing multidimensional access methods that 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 using `amdb`, a tool that enables visualization and performance analysis of access methods. 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. In particular, we found that our "Top X Jagged Bites" bounding predicate performed better than all the other access methods we tested.

1 Introduction

Millions of images are now available, 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 [20]. 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. 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 approach will not scale with increasing data set size, 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 [13]. 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.

`Amdb` [22] is a tool for the visualization, profiling and debugging of AMs implemented in GiST. In addition, `amdb` incorporates an AM analysis framework [17]. `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 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` 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 traditional AMs. Our new, custom-designed AMs performed slightly more than half as many I/Os as the best of the traditional AMs we analyzed.

Our new AMs are based upon observations of the nature of our query and data sets. The queries are all nearest neighbor queries returning more data than can fit on a single disk page. To reduce overall page accesses, we needed to reduce the number of AM node descriptors queries overlap. A nearest neighbor query starts from a given query point and expands outward equally in all directions to find the nearest points. Assuming node data points are bounded by, for example, minimum bounding rectangles, then nearest neighbor queries expand into these hyper-rectangles from the outside. Our AMs use variations of minimum bounding rectangles as descriptors for node contents. The new node descriptors are designed to reduce the number of hyper-spheres (i.e., nearest neighbor queries) that can overlap a minimum bounding rectangle descriptor without finding any satisfying data within the corresponding node. This is accomplished by "biting" empty space out of the corners of the minimum bounding rectangles.

Section 2 provides background information on GiST, `amdb` and Blobworld. The analysis process begins in Section 3, where our analysis framework is set up and blob descriptors are compared to whole image descriptors. Section 4 begins the analysis of AM efficiency. Section 5 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 6. Section 7 analyzes the performance of the new AM designs. 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

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. 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

Variable	Definition
Q	workload query set
u_q	average utilization seen by query
u_t	target utilization
L_q	number of leaf-level pages accessed by workload
L_q^o	number of leaf-level pages accessed by optimal clustering
L_q'	the sum over all queries in the workload of the number of leaf-level pages relevant to each query
$L_{q'}$	number of leaf-level pages accessed by a query in the workload
$L_{q'}^o$	number of leaf-level pages accessed by a query in the optimal clustering

Metric	Equation	Definition
Excess Coverage Loss	$L_q - L_q^o$	unnecessary I/Os due to queries accessing leaf nodes containing no relevant data as a result of inaccurate BPs
Utilization Loss	$L_q \times (1 - u_q/u_t)$	unnecessary I/Os due to the node storage utilization deviating from target utilization
Clustering Loss	$\sum_{q \in Q}((u_q/u_t L_{q'}) - L_{q'}^o)$	unnecessary I/Os due to the difference between optimal and achieved leaf-level data clustering

Table 1: `Amdb` Leaf-Level Analysis Metrics

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 here, 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 bounding predicate is consistent with the search predicate (employing the user-supplied extension method *consistent()*). 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.

2.2 amdb

`Amdb` [17, 22] 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.

The `amdb` AM performance metrics are shown in Table 1. Here we focus upon the key metrics, without in-depth discussions of the underlying variables. The interested reader is refered to [17] 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 is that I/Os may occur that could have been avoided if the data in the nodes were packed

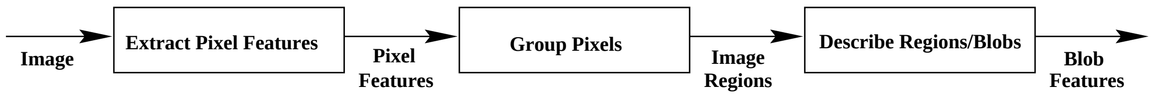

Figure 1: The Stages of Blobworld Processing: from pixels to blob descriptions

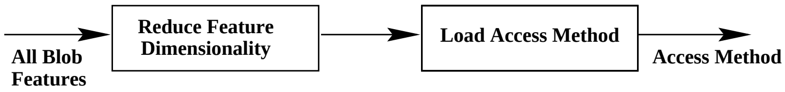

Figure 2: The Stages of Access Method Creation: from blob descriptions to an access method

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 [15]. Truly optimal clustering is NP-hard, but this heuristic works well in practice and hence serves as a good benchmark for comparison.

In this paper, we use `amdb`'s metrics and visualization capabilities to gain intuitions about how to build a custom AM for Blobworld. After implementing the AMs we design in GiST, we analyze their performance using the `amdb` analysis metrics. In Section 5 we discuss results of `amdb` analysis of the performance of well-known AMs on our query workload and data set. In Section 6 we discuss the new AMs designed to address the problems `amdb` analysis revealed in the well-known AMs' performances.

2.3 Blobworld

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

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. Querying is based 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. Blobworld differs from earlier work [7] in that its image segmentation is automatic, an approach that scales to large data sets. Blobworld also differs because it allows user control of queries at an object level, where a large portion of the image may be irrelevant.

To summarize Blobworld image pre-processing (Figure 1), pixel features are extracted from an image,

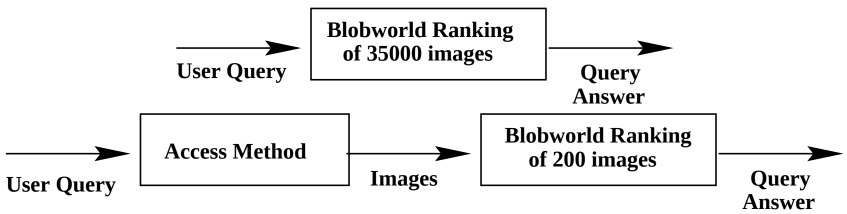

Figure 3: The Stages of a Blobworld Query: from query to answer. Note that a query without the pre-processing of an access method needs to rank all the images in the database.

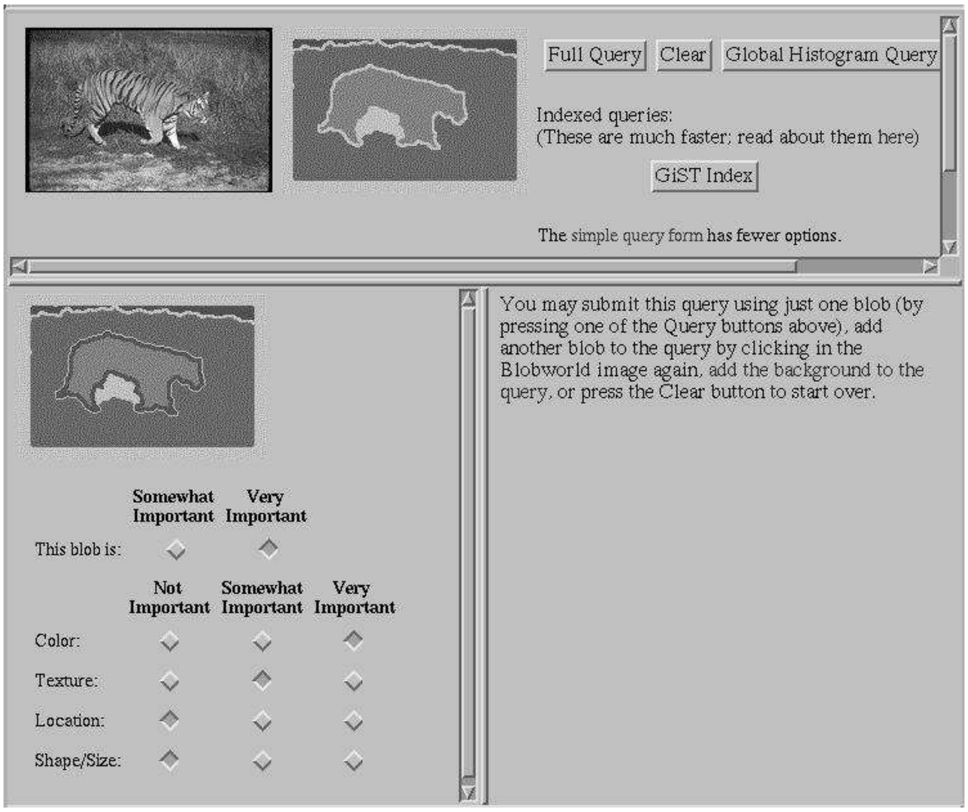

Figure 4: A Sample Blobworld Query

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. The access methods are constructed by performing dimensionality reduction on the color feature vectors and loading the resulting vectors into an access method, as depicted in Figure 2.

A user of the Blobworld system begins with a sample image and the interface in Figure 4. The user selects the blob he 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 3, 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 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 5 is an example of Blobworld query results.[1]

In addition to designing and analyzing AMs for blob data, we would like to find out if color feature vectors for blobs produce more indexable data than whole image feature vectors. Intuitively, this seems likely, since blobs are specifically selected to be regions of uniform coloring, unlike an entire image. So the color signatures of the blobs should be "sharper" than color signatures of the whole images, and should produce data that clusters better than the intuitively more diffuse color signatures of whole images. This

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

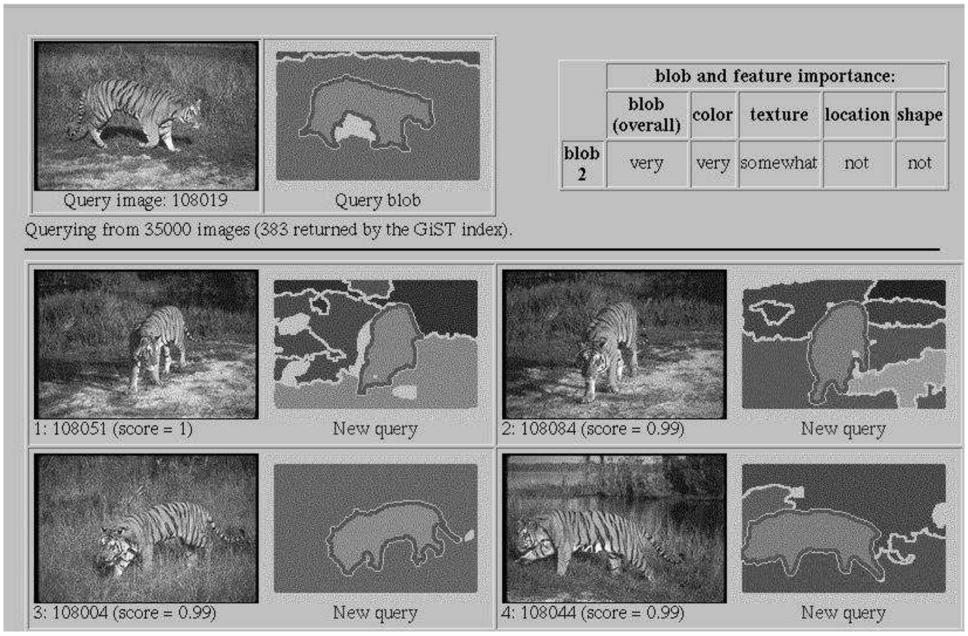

Figure 5: Sample Blobworld Query Results

should lead to workloads with better optimal clustering, hence the opportunity for higher-performance AMs.

3 Framework

The Blobworld AM must achieve two goals.

- high quality query results, acceptably close to the results of full Blobworld queries
- the AM must speed the overall query process

These goals intertwine, and changes made to improve one will affect the other, usually adversely. Therefore, the process of designing an AM for Blobworld must focus on analyzing the trade-offs between the goals and making intelligent decisions to maximize AM performance with regard to both goals.

In a traditional AM, like a B-tree, the quality of results is not an issue. A database query has a precisely defined answer set and the AM is expected to return all and only the data items in that answer set. The Blobworld AM can not meet this goal because indexing the full 218 dimensional color vectors would be prohibitively inefficient. Moreover, Blobworld queries do not have a precisely defined answer set. The desired answer set is "whatever pictures the user wants." Even the full Blobworld querying code is a sophisticated set of heuristics, because the problem of full object recognition is currently intractable. A query issued through the AM may not return the exact set of "correct" images that a full Blobworld query would. However, it should return most of the "correct" images that a full Blobworld query would. It may even return a few "correct" images that a full Blobworld query missed. From a user standpoint, this is entirely acceptable system behavior.

Measuring the quality of AM results against the perfect query results (i.e., the set of all tiger images in the database) would be prohibitively time-consuming. Because even the full Blobworld queries miss correct images, finding the perfect answer set would require a human looking through all 35000 images to flag all the correct answer images, for each query in our test workload. This is not feasible. Therefore,

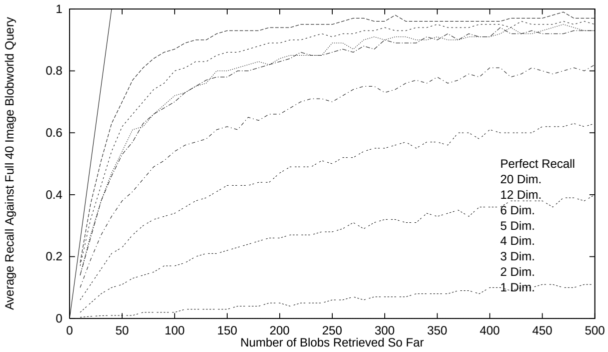

Figure 6: Comparison of Recall of Queries using Blob Color Histogram AMs of Varying Data Dimensionality vs. Full Blobworld Queries: Recall is computed against the top forty images returned by a full Blobworld query; there are 221231 blobs in the database. Note that the recall of the low dimensional queries 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.

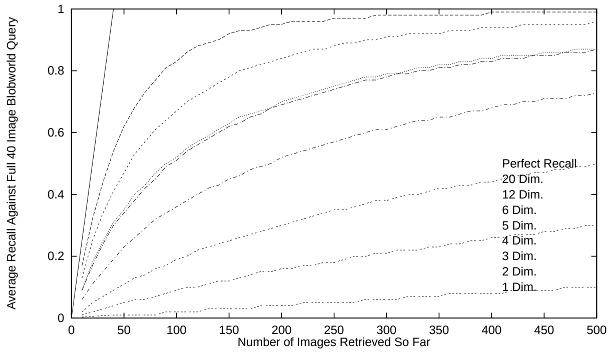

Figure 7: Comparison of Recall of Queries Using Whole Image Color Histogram AMs of Varying Dimensionality vs. Full Blobworld Queries: Recall is computed against the top forty images returned by a query that uses Blobworld ranking and comparison algorithms, but on only whole image descriptors; there are 35000 images 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. Note that an AM query returning 500 images returns about 1.4% of the whole image database. An AM query returning 500 blobs returns only .22% of the blob database.

we shall measure the quality of AM results by comparing the AM query results to the results of full Blobworld queries. This measure has the advantage that, as the full Blobworld querying results improve, the Blobworld AM result quality should parallel that improvement.

We will trade off the quality of AM results to improve AM speed, and vice versa. For example, indexing fewer dimensions leads to higher fanout inside the AM, hence fewer I/Os and faster queries. It also reduces the quality of the AM results.

3.1 Setting up the Access Methods

To find the best query return set size, we examine the results of Blobworld queries that return varying numbers of images. To find the best data dimensionality we examine AMs that store color feature vectors with varying dimensionality. This discussion is independent of the particular AM used; all AMs will the same answers with varying efficiency. We will begin the discussion of efficiency in Section 4.

Lower data dimensionality increases tree fanout and reduces tree height, which is good for I/O performance. Retrieving fewer images from the AM reduces the number of necessary AM page accesses and the costs for the final ranking that Blobworld performs, which also improves performance. 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 so as to make the results of an AM Blobworld query agree with the results of a full Blobworld query as much as possible. Therefore, we seek to simultaneously minimize the data dimensionality in the AM and the number of images an AM query returns, while still giving the user nearly the same results he would get from a full Blobworld query over all the images.

The full color feature vectors have 218 dimensions, which is typically too many dimensions to index effectively for nearest neighbor queries [6]. Therefore, we perform Singular Value Decomposition on the color feature vectors, as suggested, for example, in [6] and [11]. The resulting vectors are truncated to include only the most significant dimensions. Figure 6 shows the average query recall[2] of 5531 nearest neighbor queries over relatively low dimensional blob feature vectors, calculated relative to the results of the same queries run using Blobworld query processing of the full 218 dimensional vectors. The dimensionality of the truncated 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 AM data curve. Adding further dimensions to the data set does not significantly improve the quality of the data results.

To compare the indexability of color feature vectors based on blobs to that for color feature vectors based upon the entire images, Figure 7 shows the results of analogous queries run using feature vectors for the whole images. The same set of queries were run over the same set of images, which were described using color histogram vectors, one per image, whose dimensionality had been reduced using SVD. The images returned by the AM were calculated relative to the top forty images returned by a full Blobworld query over the data set of 35000 whole image feature vectors, not blob feature vectors. Figure 7 shows that the recall of the results of the low dimensional whole image descriptor queries does not rise as swiftly with the increase in number of dimensions as the low dimensional blob query results. Note that Figure 7 paints the rosiest possible picture for whole image AM use, since the whole image AM returns a far greater percentage of its data set for each query. The x axis of Figure 7 is in images, and the x axis of Figure 6 is in blobs. However, the blob AM is about six times larger than the image AM (since there are six blobs per image on average). The low dimensional blob descriptor queries are achieving better recall than the whole image descriptor queries, so the benefit of breaking images into blobs seems to outweigh the cost of storage.

Optimal clustering numbers derived using the same clustering algorithms that `amdb` uses indicate that the blob data could be clustered by an ideal AM in such a way as to allow each query to run with slightly more than half the number of necessary leaf level I/Os (3.9 average I/Os per test workload query) as would be necessary for whole image queries (6.7 average I/Os per test workload query).[3] Unfortunately, using clusterability analysis tools to compare whole image and blob AM performance produces dubiously useful information, because the whole image and blob data sets not only contain different feature vectors, but the sets themselves are of different sizes. So we can not be positive how much of the differences in AM recall with changing AM dimensionality are due to the fact that the whole image set allocates only one data item to each image, while the blob set gives each image several data items (blobs).

Analyses of AMs based upon the whole image color feature vector data were run in parallel with the blob feature vector data AM analyses, but they all bear out the first cut intuitions that whole image feature vector AMs do not perform as well as blob feature vector AMs, so we will not present them.

We 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. Two hundred images is a point lying just beyond the "knee" of the recall curves in Figure 6. 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.

[2] $\frac{relevant\ image\ retrieved}{(relevant\ images\ retrieved)+(relevant\ images\ not\ retrieved)}$

[3] These numbers were derived by running around 20000 queries over the full Blobworld algorithms, to retrieve the top forty images the full Blobworld algorithms would retrieve for a blob AM with perfect recall, and the top forty images the full Blobworld algorithms would retrieve for a whole image AM with perfect recall. With perfect recall, of course, there would be no need to retrieve more than the top forty images the user is actually going to look at from the AM. This perfect workload was then clustered using the algorithms `amdb` uses, to learn the clusterability of perfect blob and whole image AMs.

4 Beginning the Design

Having covered the issues of data dimensionality and query set size, we will now address AM performance efficiency.

4.1 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,[4] and hence are based on one of the eight sample images in that page.

Thus, we felt an artificial workload would better stress our access methods. 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 [17].

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. 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.

4.2 Access Method Efficiency

Having decided upon our data set and query workload, we aimed to design the most efficient AM we could, 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 7.

AM performance *must* be faster than simply scanning a flat file of the five-dimensional feature vectors, in order for the AM to be worth using. 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.[5] So 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,[6] 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 AMs.

5 Standard Index Alternatives

We used the STR algorithm [18] to sort the data for bulk-loading into an R-tree [10], which is the canonical multidimensional AM, used in a variety of research and commercial systems. As shown in Table 2, 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 [7] in excess coverage. Essentially, the only problem with a bulk-loaded R-tree is its sloppy BPs. We also bulk-loaded two other previously-proposed

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

[5] Using Seagate Barracuda ultra-wide SCSI-2 drives, [21] 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.

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

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

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

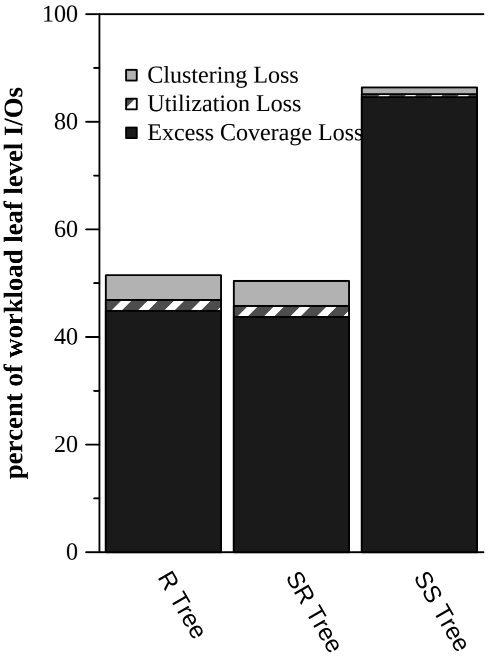

Figure 8: Access Method Performance Losses Relative to Total Workload Leaf Level I/Os: This shows the percent of the leaf level I/Os that were due to excess coverage, utilization and clustering loss. The remaining percentage represents I/Os necessary to answer the queries.

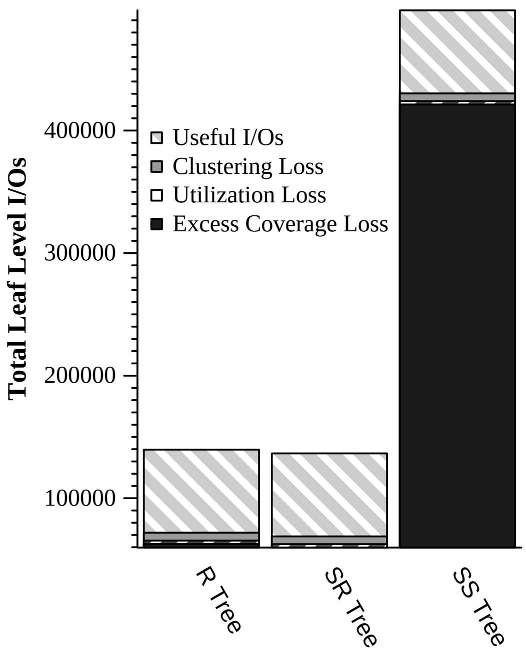

Figure 9: Access Method Performance Losses in Number of Leaf Level I/Os: This shows the number of the leaf level I/Os that were due to excess coverage, utilization and clustering loss. The remaining I/Os were necessary to answer the queries.

AMs for multi-dimensional data, the SR-tree [16] and the SS-tree [24], to see if the different BPs helped performance. SS-trees use spherical BPs and SR-trees use minimum bounding rectangles plus bounding spheres.

Figure 8 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 9 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. When performance losses for inner nodes plus leaf nodes of the AMs are considered, the SR-tree has higher excess coverage loss than the R-tree. However, for both R-trees and SR-trees, about 30 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 AM needs more restrictive BPs. The ideal BP would 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

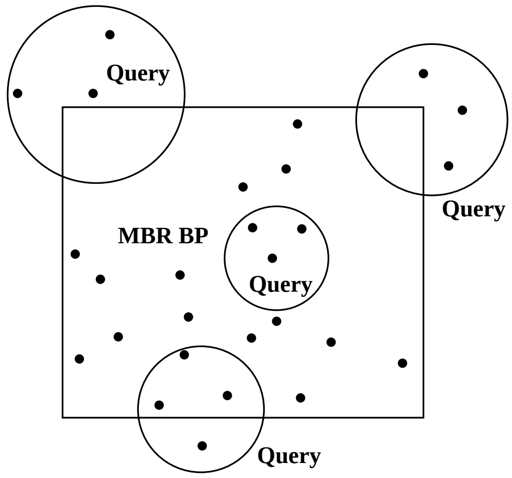

Figure 10: Nearest Neighbor Queries And Rectangular BPs

criteria, at a very high cost in terms of the second. In the next section we focus on better BPs for nearest neighbor queries which strike a balance between BP size and the precision of data description.

6 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 [3] work by finding points within a given distance of the query point, in essence asking expanding sphere queries. Figure 10 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 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 the two topmost queries in Figure 10, should 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 were 24 children, 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.

Figure 11 shows the `amdb` visualization of some individual 2D R-tree nodes (because 5D data can not be easily visualized), showing the points the minimum bounding rectangles (MBRs) contain and the MBR BPs. The data points of some leaf nodes do not fill their MBRs, instead leaving 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 10 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.

6.1 MAP

The first alternate BP we consider is the Minimum Area Predicate (MAP) BP, 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 is not precisely what we want to minimize,

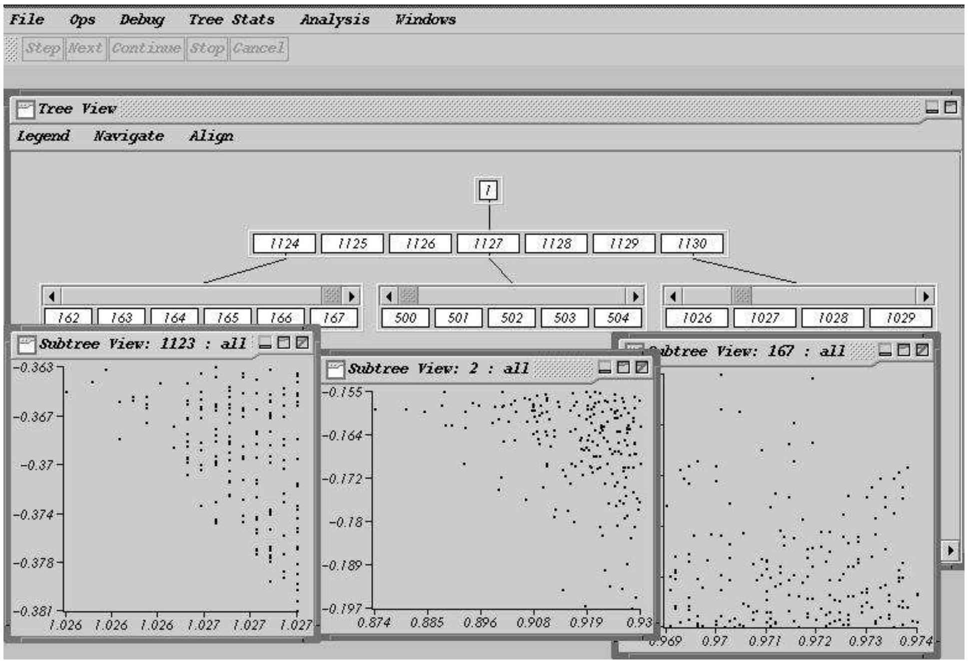

Figure 11: `amdb` Visualization of 2D R-tree Leaf Nodes

but it is the heuristic used in a number of earlier AMs [1, 10]. 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 is likely to be beneficial. So using the heuristics developed for R-tree node splitting would be counterproductive in this situation.

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 the minimum total volume out of those it examines to be the node BP.

6.2 JB

Consider the points in the rectangle in Figure 10. The MAP BP in Figure 12 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 12. 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, would describe the data in a node more precisely than a MAP BP. The JB BP, pictured in Figure 13 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

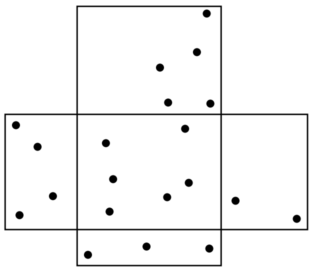

Figure 12: A MAP BP

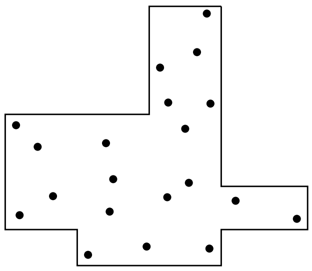

Figure 13: A Jagged Bites BP

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 14: 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.

associated MBR corner point and another point at the one "internal" corner of the bite rectangle, which might not intersect any MBR hyper edge.

An algorithm for creating a JB BP out of a combination of projections of the data points to the dimensional axes is in Figure 14. This heuristic tries every possible value of dimension 0. 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 constructed thusly, it 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. We use a heuristic algorithm because the problem of creating a perfectly optimal JB BP is NP-complete, the proof of which is in Appendix A.

6.3 XJB

An alternative, called XJB for "Top X Jagged Bites," is to simply store the X largest bites, leaving the remaining MBR corners "unbitten." Note that picking the bites with the largest volumes is only a

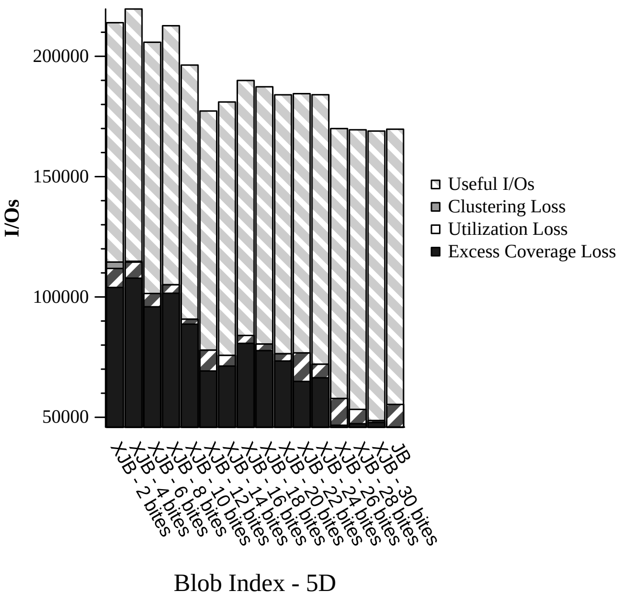

Figure 15: XJB Access Method Performance Losses in Number of Total Workload I/Os for various values of X

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

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.

All three of the BPs we propose involve more complex code for AM searching than MBR BPs. The 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.

7 Analysis of Alternatives

Before we begin with the results of the `amdb` analysis of the new access methods, we note that there are several other ways to compare the AMs. For example, Table 3 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 4 shows that the new BPs have succeeded in reducing the total

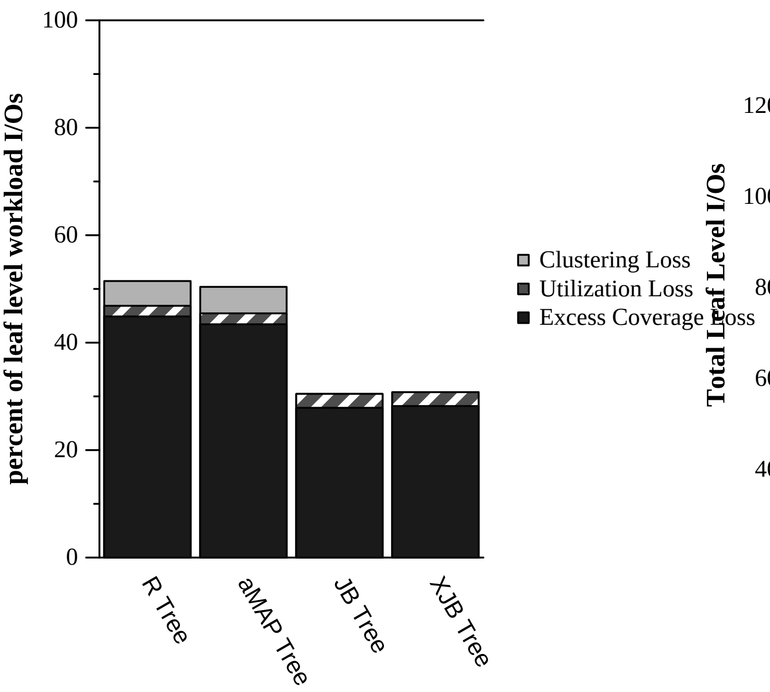

Figure 16: Access Method Performance Losses Relative to Workload Leaf Level I/Os: $X = 26$ for XJB

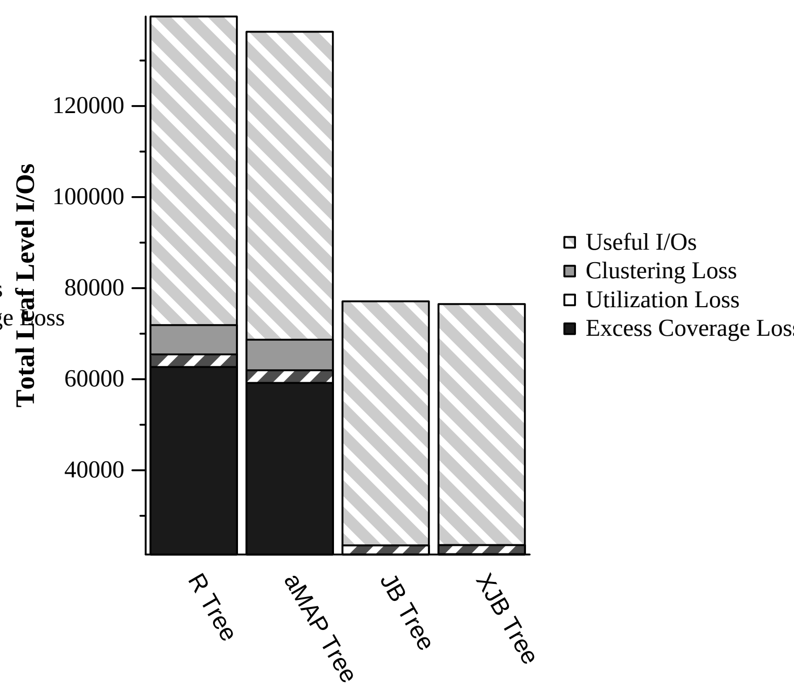

Figure 17: Access Method Performance Losses in Number of Leaf Level I/Os: $X = 26$ for XJB

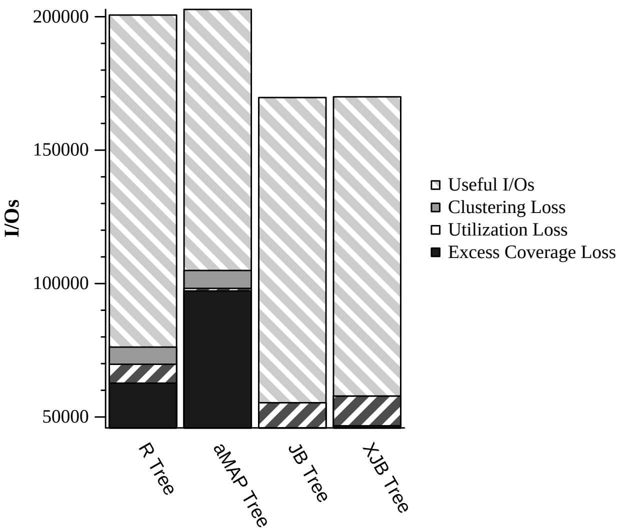

Figure 18: Access Method Performance Losses in Number of Total Workload I/Os: $X = 26$ for XJB

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

Table 4: Sums of the hyper-volumes of all the leaf-level bounding predicates in the access methods.

data volume enclosed by all the BPs in the AMs.[8] 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.

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 16, 17 and 18 show that our expectations were not quite accurate.

As shown in Figures 16 and 17 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 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 17 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 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 AM 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 $X = 26$ because, as Figure 15 shows, that is the smallest number of corner bites for which the number of I/Os is effectively as low as for the JB tree. 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 4 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 for disk accesses. 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.[9]

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.

8 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 variably sized inner nodes. It is designed to minimize the overlap of BPs, which we are able to accomplish more effectively

[8]Note that the volumes of unit radial hyper-spheres fit on a curve, with the five dimensional unit radial hyper-sphere having the maximum volume. [23]

[9]The MAP AM hit about 1 in 52 pages; the other new AMs did considerably better.

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 [19], 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.

High dimensional indexing work to date focused upon creating indices for any high dimensional data. To our knowledge, the problem of taking a single application and its data set, then tailoring an access method just for them has not been previously addressed.

9 Conclusions and Future Work

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 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.

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 could 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
- 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`. We thank Kris Hildrum for co-creating the JB Problem NP Construction. We thank Chad Carson for all the work he did to make our experiments possible. We thank Joe Hellerstein for being a patient and thoughtful advisor. This research was supported by fellowship stipend support from the National Physical Sciences Consortium and Lawrence Livermore National Laboratory.

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, E. Koutsoupias, and C. Papadimitriou. On the analysis of indexing schemes. In *Proc. of the ACM-SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 249–256, Tucson, AZ, May 1997.

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

[14] K. Hildrum and M. Thomas. Jagged bite problem np-complete construction. Technical Report UCB//CSD-99-1060, University of California at Berkeley, 1999.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

A JB Problem NP-Complete Construction

This proof was first presented in [14].

Definitions

Without loss of generality, we assume that all points are positive, and that the MBR corner we bite into will be located at $\vec{0}$. If this is not the case, the problem can be adjusted by shifting and reflection so that it holds.

We also assume that all the p_i in all the points $\vec{p}$ are finite, i.e., $\forall p_i$ in all $\vec{p}, p_i < C$ for some finite number C.

JB problem

JB problem: Given a set P of n-dimensional points, maximize

$$\prod_{i=0}^{n-1} v_i = v_0 v_1 \dots v_{n-1}$$

subject to $\forall \vec{p} \in P, \exists i$ such that $p_i \geq v_i$.

JBP problem

To argue NP-hardness, we need to turn the above problem into a problem that has a yes/no answer. Call this problem the JBP (Jagged-Bite-Predicate).

JBP problem: Given a number k, and a set of points P, does there exist a vector $\vec{v}$ such that:

- $\prod v_i \geq k$
- $\forall \vec{p} \in P, \exists i$ such that $p_i \geq v_i$?

Notice that these two problems are polynomial time equivalent. It is clear that solving jagged-bite problem means one can solve the predicate version. To go the other way, use JBP as an oracle in a binary search to find the maximum k, which gives an answer to the JB problem. For example, start with $k = 100$, and if the answer is yes, double k. (Otherwise, halve k.) If the answer is no, then set $k = 1/2(100 + 200)$, then if the answer is yes, set $k = 175$ and so on until k is found. This will take a logarithmic number of uses of the JBP oracle.

3-SAT

Given variables $x_0 \ldots x_{n-1}$ and a formula $\phi = c_1 \wedge c_2 \wedge \ldots c_m$, where c_i is the disjunction of three literals (for example $c = x_3 \vee \neg x_2 \vee x_5$), determine whether ϕ is satisfiable.

JBP is NP-hard

We show that a polynomial-time solution of the JBP problem provides a polynomial-time solution to the 3-SAT problem. Therefore, JBP is NP-hard.

The Reduction

Given the formula ϕ, we create the following JBP problem by creating points of dimension $2n$ and setting k.

Intuitively, our reduction will relate $\vec{v}$ to a satisfying assignment. The vector $\vec{v}$ will have a location in it for each literal (a literal is a variable or its negation). Loosely, $\vec{v} = (x_0, \neg x_0, x_1, \neg x_1, \ldots, \neg x_{n-1})$, where $x_i = 1$ if x_i is true, and $x_i = 2$ if x_i is false. Likewise, $\neg x_i$ is 2 if x_1 is true, and $\neg x_1$ is 1 if x_i is false. In the discussion that follows, v_{2i} corresponds to the literal x_i, and v_{2i+1} corresponds to $\neg x_i$.

In order for the above description of $\vec{v}$ to make sense, certain values of $\vec{v}$ must be prevented. For example $\vec{v} = (1, 1, 1, 2, 1, 2)$ would imply that both x_0 and $\neg x_0$ are true, which would not correctly model the satisfiability problem. Some of the points created in the reduction exist only to force $\vec{v}$ to be well-formed. Other points are created from the clauses of ϕ such that satisfying a clause is equivalent to finding an i such that $p_i \geq v_i$ for the corresponding point.

As a running example to clarify the procedure, we will use: $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$

clause points Convert a clause c of ϕ to a point $\vec{p}$ of dimension $2n$ in the following way.
$p_{2i} = 1$ if x_i appears in clause c, $p_{2i+1} = 1$ if $\neg x_i$ appears in clause c, and both are 0 otherwise. Create one such point for each clause of ϕ. The purpose of the clause points is to capture the particular formula ϕ.The purpose of the clause points is to capture the particular formula ϕ. The clauses in our running example become the points $(1, 0, 1, 0, 1, 0)$ and $(0, 1, 0, 1, 0, 1)$.

two-points For every $i \in [0, 2n-1]$ we add the point $p_j = \begin{cases} 2 & j = i \\ 0 & otherwise \end{cases}$.

The purpose of the two-points is to ensure that all the v_i will be ≤ 2. In other words, to make the JB hyper-rectangle into a hyper-square with all sides of length 2. This means, for example, that a vector $\vec{v} = (1, 1, 1, 2^4)$ will not be possible. The example formula generates $(2, 0, 0, 0, 0, 0)$, $(0, 2, 0, 0, 0, 0)$, $(0, 0, 2, 0, 0, 0)$, $(0, 0, 0, 2, 0, 0)$, $(0, 0, 0, 0, 2, 0)$, and $(0, 0, 0, 0, 0, 2)$.

one-points For every $i \in [0, n-1]$ we add the point $p_j = \begin{cases} 1 & j \in [2i, 2i+1] \\ 0 & otherwise \end{cases}$ This means the points $(1, 1, 0 \ldots 0), (0, 0, 1, 1, 0, \ldots, 0)$ and so on. These points prevent a variable and its negation from being false at the same time. Indirectly, this prevents both a variable and its negation from being true. In terms of points, one-points prevent $\vec{v}$ from being $(2, 2, 1, 2, \ldots)$

The example formula generates $(1,1,0,0,0,0)$, $(0,0,1,1,0,0)$, and $(0,0,0,0,1,1)$.

Choose $k = 2^n$. Though listed last, this condition is vital. Combined with one-points and two-points, this is the condition that forces $\vec{v}$ to have the desired structure of 1's and 2's.

The example formula $k = 2^3 = 8$.

In the next two sections, we will prove that our reduction is correct. That is, we must show that ϕ is satisfiable if and only if JBP says yes.

ϕ satisfiable $\Rightarrow$ JBP says yes

We want to show that if the 3-SAT formula is satisfiable, then the JBP algorithm will say yes. To prove this this, we will find a $\vec{v}$ such that:

- $\prod v_i \geq 2^n$
- For all points $\vec{p}$, $\exists i$ s.t. $p_i \geq v_i$

For the ϕ given above, one satisfing assignment is $x_1 = true$, $x_2 = false$, and $x_3 = false$. From this assignment, the proof below would construct the $\vec{v} = (1,2,2,1,2,1)$.

Proof

Choose $v_{2i} = 1$ if x_i is true, 2 otherwise. Similarly, $v_{2i+1} = 2$ if x_i is true, 1 otherwise. Clearly, $\prod v_i = 2^n \geq k = 2^n$, so the first condition is satisfied.

Now, we must show that for all points $\vec{p}$, $\exists j$ such that $p_j \geq v_j$. There are three cases:

- $\vec{p}$ is a clause point. We need to show that in one of the variables with a 1, $\vec{v}$ also has a one. Fortunately, we know that at least one literal in the clause corresponding to that point is true. Suppose that literal is x_i (or $\neg x_i$). Then $v_{2i} = 1$, (or $v_{2i+1} = 1$) since we constructed $\vec{v}$ to ensure that this would be true. Therefore, we have that $p_{2i} \geq v_{2i}$ (or $p_{2i+1} \geq v_{2i+1}$).
- $\vec{p}$ is of the form $(\ldots 0,\ 2,\ \ldots 0)$ We selected $\vec{v}$ such that every element of $\vec{v}$ is less than or equal to 2, so this holds.
- $\vec{p}$ is of the form $(\ldots, 1,\ 1,\ 0,\ \ldots 0)$ We constructed $\vec{v}$ such that one of every pair is 1, so this is satisfied.

JBP says yes $\Rightarrow$ ϕ satisfiable

Suppose that the JBP says yes. From the vector $\vec{v}$, we will construct a satisfying assignment to ϕ.

First, recall that we know that all v_i are less than 2 (because of the two-points). Further, for every pair v_{2i}, v_{2i+1}, we know that at least one of those pairs is less than one (because of the one-points). So, we know that $v_{2i} \times v_{2i+1} \leq 2$. Finally, since we know that $\prod v_i \geq 2^n$, we can conclude that $v_{2i} \times v_{2i+1} = 2$. Since each of them has a maximum of two, we can conclude that of the pair v_{2i}, v_{2i+1}, exactly one must be 2, and the other must be 1.

So, we construct a satisfying assignment in the following way: x_i is true if $v_{2i} = 1$, and x_i is false if $v_{2i+1} = 1$.

Now we show that for this assignment, each clause has at least one true literal, and so the formula is satisfied. Fix a clause c. This clause corresponds to a point $\vec{p}$, and we know that for some i, $p_i \geq v_i$. The only way $p_i \geq v_i$ is if $p_i = v_i = 1$. (Since p_i can be 0 or 1, and v_i can only be 1 or 2.) If v_i is 1, then that means the corresponding literal is true, and if p_i is 1, then that means that the corresponding literal is in the the clause c, which means that c contains at least one true literal, so c is satisfied. Since this holds for every c, the formula as a whole is satisfied.

NP-completeness

A problem is NP-complete if it is is NP-hard and in NP. We have shown that the JBP problem is NP-hard, so to show it is NP-complete, we need only show that it is in NP.

A problem is in NP if it has a polynomial size witness that can be verified in polynomial time. In this case, $\vec{v}$ is a witness. It is easy to check, given $\vec{v}$, that $\prod v_i \geq k$ and that $\forall \vec{p}, \exists i$ such that $p_i \geq v_i$. This means that JBP is NP-complete.