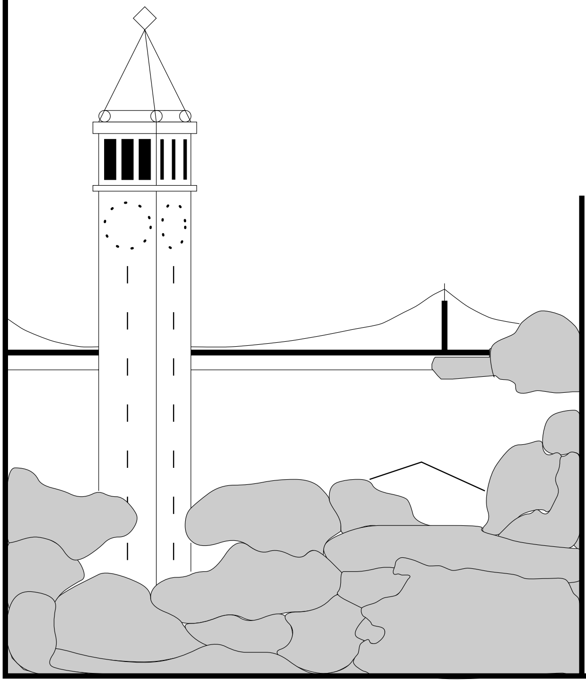
Boolean Bounding Predicates for Spatial Access Methods

Megan Thomas and Joseph M. Hellerstein



Report No. UCB/CSD-2-1174

March 2002

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

Boolean Bounding Predicates for Spatial Access Methods

Megan Thomas and Joseph M. Hellerstein

ABSTRACT

Tree-based multidimensional indexes are integral to efficient querying in multimedia and GIS applications. These indexes frequently use shapes in internal tree nodes to describe the data stored in a subtree below. We show that the standard Minimum Bounding Rectangle descriptor can lead to significant inefficiency during tree traversal, due to false positives. We also observe that there is often space in internal nodes for richer, more accurate descriptors than rectangles. We propose exploiting this free space to form subtree predicates based on simple boolean combinations of standard descriptors such as rectangles. Since the problem of choosing these *boolean bounding predicates* is NP-complete, we implemented and tested several heuristics for tuning the bounding predicates on an index node, and several heuristics for deciding which nodes in the index to improve when available tuning time is limited. We present experiments over a variety of real and synthetic data sets, examining the performance benefit of the various tuning heuristics. Our experiments show that up to 30% or more of the total I/Os in a query workload can be eliminated using the boolean bounding predicates chosen by our algorithms.

1 Introduction

Spatial and multimedia databases often make heavy use of search-tree indexes like R*-trees [3] to provide efficient query processing. As an example, the Blobworld image search system [5] supports nearest neighbor queries over an index built on color vectors, to answer queries of the form "find me images like this one". A wide variety of multimedia and GIS applications can be made more efficient by reducing the number of I/Os performed in index search.

The internal nodes of tree-based indexes typically contain a sequence of pairs (p, ptr), where ptr is a pointer to a subtree, and p is a descriptor – or "Bounding Predicate" (BP) – for the subtree below, such that each datum found in the leaves below satisfies p. The popular R*-trees, for example, use Minimum Bounding Rectangles (MBRs) as their BPs.

We will show that in many spatial and multimedia index scenarios, most of the I/O overhead is caused by imprecise BPs misdirecting the tree traversal algorithm. In this paper we present a simple but powerful BP representation that improves the accuracy of bounding predicates without changing the tree structure – in particular, without expanding the height – of the index.

We achieve our performance benefits by exploiting unused space in the inner nodes of search-tree indexes. Indexes often have some space on each node empty in order to accommodate future insertions, or because of the structural effects of node splits [13]. We use this free space to store more accurate versions of the inaccurate bounding predicates on the node.

Multidimensional bounding predicates can in principle be arbitrarily complex geometric shapes; however, such complexity can consume much of the remaining free space on a node, and can complicate basic search primitives like spatial overlap. Inspired by Constructive Solid Geometry [8], we propose retaining the simplicity of basic shapes like rectangles, but combining them in boolean expressions. For instance, we can replace a simple MBR with the *union* of two smaller rectangles that, together, more tightly bound the same set of points. Alternately, we could describe a set of data points with the original MBR *minus* some smaller rectangle, where the smaller rectangle describes unpopulated space within the MBR. Generally, we refer to these combined descriptors as *boolean bounding predicates.*

Function Name	Purpose
Consistent	Given a query predicate and an index node, returns true if node satisfies predicate
Union	Constructs a new bounding predicate out of a given set of points and/or bounding predicates
Penalty	Given a point and an index node, return penalty (a number) for inserting point into that node
PickSplit	Split a set of node entries into two; used to handle page overflow during insertion

Table 1: Basic GiST API: The logic of the boolean bounding predicates functionality we added to GiST mainly affects the performance of the Consistent function.

For a given BP, there are a variety of possible ways to tune it, resulting in different boolean BPs. For a given node, there are a variety of choices in allocating the free space to tune the various BPs on the node. For a given index tree, there may be nodes that greatly benefit from tuning their BPs, and nodes that benefit less. These options lead to the set of design problems that we study here. First, we provide a simple language for refining BPs via boolean connectors. Second, we examine how to allocate free space to BPs on an individual internal node to minimize false positives during index tree traversal, a process we call *tuning* a node. Finally, we investigate how to prioritize nodes to tune, so that the tuning process need not visit the entire tree in cases where doing so is prohibitively expensive.

Our experimental results show that up to 30% or more of the total I/Os for a set of queries can be eliminated using boolean bounding predicates chosen with our tuning algorithms.

Section 2 provides background and motivation for the work presented here, and Section 3 outlines our approach. We elaborate on our boolean bounding predicates in Section 4. Sections 5 and 6 cover the tuning and prioritization algorithms we implemented and experimental results. We cover related and future work in Sections 7 and 8 and conclude in Section 9.

2 Background and Motivation

We implemented boolean BPs in the libgist Generalized Search Tree (GiST) package [11], which provides a convenient infrastructure for experimenting with new indexing schemes. We extended the GiST framework (Table 1) with a small set of interfaces (covered in more detail in Section 4) for generalized boolean BPs.

To get a feel for typical performance problems in multimedia search, we used GiST and its companion profiling tool `amdb` [15, 19] to analyze sample queries for the Blobworld [5] image retrieval application. We provide an intuitive overview of `amdb`'s profiling here. The only I/Os that are necessary in traversing a search-tree index are those that lead the traversal algorithm to data that must be returned to the user; other I/Os are performance *losses* due to index inefficiency. The `amdb` analysis assigns each I/O to one of four causes: “retrieving useful data”, “utilization loss” (under full index nodes), “clustering loss” (poor assignment of data items to leaves), or “excess coverage loss” (imprecise bounding predicates). As Figure 1 shows, most of the losses in our Blobworld experiments were attributed to imprecise BPs. Imprecise BPs lead to unnecessary I/Os because they result in “false positives”, guiding the tree-traversal algorithm to leaf nodes that do not contain answers to the query. Excess coverage loss I/Os constituted from 10% to over 50% of the total I/Os performed by query workloads over R*-trees built on the various data sets we used for experiments.

We note that Blobworld – like many GIS and multimedia applications – has a read-mostly

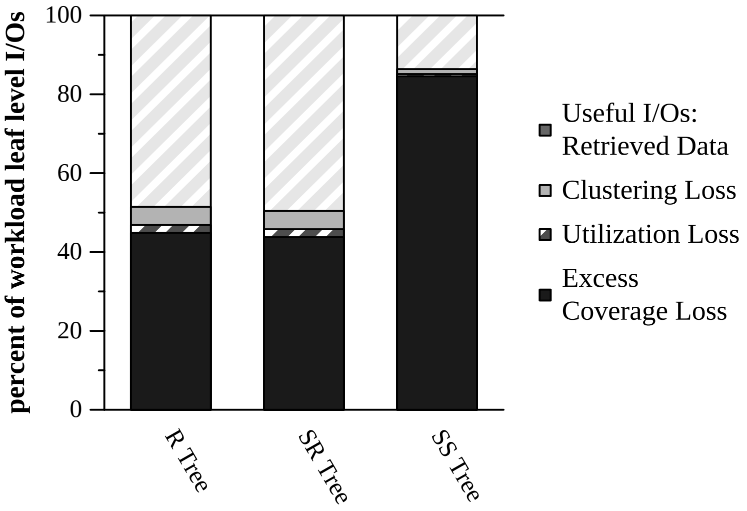


Figure 1: I/Os performed during a query workload, broken down by underlying cause. Tests run on R-tree [10], SR-tree [14] and SS-tree [20] spatial indexes built over sorted, bulkloaded Blobworld data set.

workload with very infrequent, batch updates. Because the data is largely static, it can be sorted and the index intelligently bulk-loaded [16]; this explains the low utilization and clustering losses in our experiments (Sections 5 and 6). In these scenarios, the key to good performance is to improve the precision of BPs to minimize excess coverage loss – the focus of this paper. We defer discussion of dynamic updates to future work (Section 8). However, we wish to stress here that an understanding of our static BP tuning problem is both of practical importance in many applications, and a prerequisite to addressing more dynamic environments.

3 Bounding Predicate Imprecision Problem

The crux of the boolean bounding predicate approach is to begin with a basic R*-tree[1], select a poorly performing inner node and use the free space on that node to store extra rectangular components for some of the BPs on the node. We call this process *tuning*. Simple BPs are transformed into combinations of basic rectangles, united using the boolean operators *union* and *minus*.[2]

We have three problems to address in designing boolean bounding predicates:

1. **Boolean BP Creation:** Given a simple MBR, how should it be broken up into multiple rectangles combined with boolean operators? (Section 4.)

2. **Node Tuning:** Given a fixed-size internal node containing $<\mathrm{BP}, \mathsf{ptr}>$ pairs and some free space, reallocate space to the BPs to generate boolean BPs that minimize false positives during index traversal. (Section 5.)

3. **Node Prioritization**: In some environments it may not be possible or worthwhile to tune every node in the index tree. So, given a search tree with k internal nodes and a budget $B < k$ of nodes to tune, select the best *B-subset* of the internal nodes for node tuning. (Section 6.)

[1]The R*-tree was selected for general familiarity; our ideas are applicable to many tree-based indexes.

[2]A third boolean operator would be *intersection*. However, the intersection of two overlapping rectangles is simply one smaller rectangle so, for simplicity, we have not implemented *intersection*. As we shall see in Sections 5 and 6, we do well even without it.

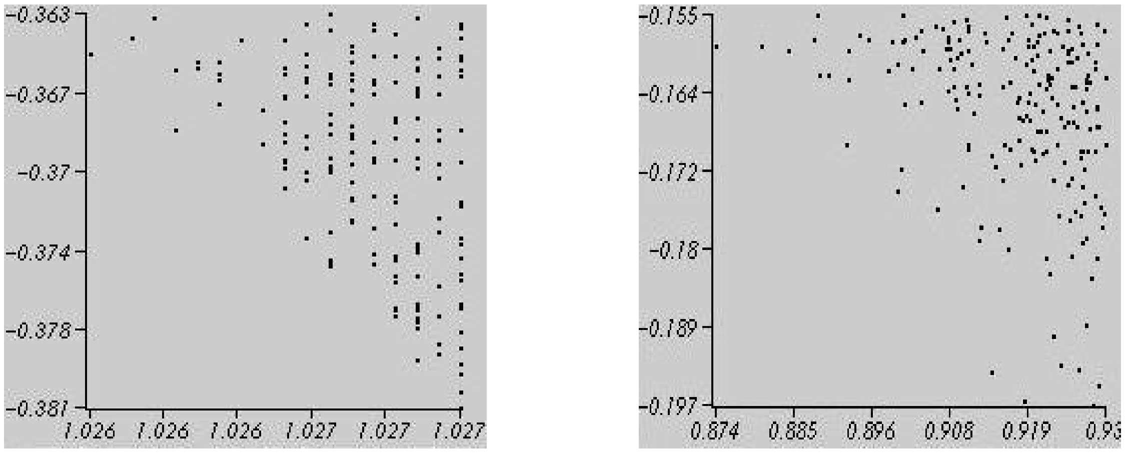


Figure 2: Screen shots from the `amdb` tool, graphing the points stored on sample leaves of a 2D R*-tree index.

Vital to the first two of these problems is the issue of judging the "badness" of a particular BP (BP_badness), in order to determine if we have available a "better" BP to replace it with. The third problem is based on judging the "badness" of the set of BPs on a node (node_badness), in order to determine how to prioritize nodes for tuning. Recall that our overall badness metric is excess coverage loss: the number of tree traversal I/Os that lead to irrelevant leaves. Given two different BPs and no other information, it is expensive to accurately determine which BP is going to contribute more excess coverage loss: the only way to be sure is to run the relevant query workload against both BPs. This measurement process is too expensive to perform in the inner loops of our tuning algorithms. Hence for Node Tuning and Boolean BP creation, we compare potential BPs using hyper-volume as our BP_badness metric. This mirrors the heuristics used in many other spatial data structure algorithms – e.g. the R-tree split algorithm. For Node Prioritization, the node_badness metric we use is `amdb`'s excess coverage loss metric, as gathered by running a single characteristic workload over the untuned tree.

3.1 Aside: Why Rectangles?

Instead of constructing more precise bounding predicates out of simple rectangles, we did consider using shapes, like the convex hull, designed to precisely bound point sets. Unfortunately, the convex hull has a number of undesirable properties. For example, we would have little control over the size of the hull description and, consequently, over the fanout of the index tree; many algorithms exist to approximate convex hulls around data points in more than two dimensions, but even sophisticated tools like QHull [2] allow users to control the precision of the resulting hull, not the number of vertices. A Qhull user can manually merge hull surface facets to create hulls with a user-selected number of vertices and facets, but that approach is impractical for our purpose.

In addition, calculating the containment of a point or another convex hull within a given convex hull is a more complex process than tolerable for an operation performed many times per query.

Another important bar to the use of convex hulls is illustrated in Figure 2, which shows sets of points from the leaves of a R*-tree [3] constructed over two dimensional color vector data. The gaps in the lower left of the two different point sets suggest that concavity would be desirable in a BP; convex hulls can not accomodate this desire.

Function Name	Purpose	Called By
TuneIndex	Given an index, sort the nodes using the specified node ordering algorithm – SS, SSC or CR – and call TuneNode on each node in turn.	User
TuneNode	Given a node, tune it using the specified node-level algorithm: NR, NG or NSA.	TuneIndex
TuneMinus	Given a BP or BP component, split it into two components to be combined with minus; return new BP components and benefit of change.	TuneNode
TuneUnion	Given a BP or BP component, split into two BP components to be combined using union; return new BP components and benefit of change.	TuneNode
min_space_needed	Used in conjunction with the node management function returning node free space, to calculate whether there is space for another BP component on the node.	TuneMinus and TuneUnion
bp_badness	Return the value of some metric representing the "badness" of the given BP; we used hyper-volume.	TuneMinus and TuneUnion

Table 2: Boolean Bounding Predicates Functions: These functions were added to the GiST API in order to provide a means to experiment with index tuning.

4 Creating Boolean Bounding Predicates

We begin by addressing how we transform simple MBRs into boolean BPs. Our basic operations are to replace a rectangle with either the union of two rectangles, or the difference of two rectangles.

TuneUnion essentially splits the rectangle into two; this is a common operation in most multidimensional search trees during node splitting. However, in our case we are not splitting up the set of points in order to separate them onto different nodes in the tree; we just want to more tightly circumscribe their extents, as in Figure 3. Hence our union split algorithm is a variant of the R*-tree node splitting algorithm [3], with a different optimization goal. In particular, the R*-tree splitting algorithm works to minimize overlap between the resulting two bounding rectangles, which are intended to separate two subsets of points. In our case, overlap is perfectly acceptable, because we are characterizing a single set of points; we simply wish to minimize hyper-volume. So we take the R*-tree split algorithm and swap the calls to minimum "overlap-value" with minimum "area-value" (hyper-volume) in the subroutine ChooseSplitIndex, as shown in Figure 5. We also set the split imbalance parameter m, which controls the ratio of the number of data items in one resulting rectangle to the number of data items in the other, to 5% of the number of data items on the page, instead of the 40% recommended in [3], because it is perfectly acceptable in our scenario for the proportion of points to bounding rectangles to be skewed.

Our minus split algorithm, TuneMinus, does not have an obvious analogy in prior work. It is NP-complete to choose the best rectangle to subtract; indeed, it is NP-complete even under the constraint that the subtrahend is a corner of the original rectangle [12]. Intuitively, this corner constraint is attractive because MBR corners "stick out", and are likely parts of the MBR to unnecessarily overlap a large number of queries. Since even this constrained problem is NP-complete, we use a greedy heuristic, presented in Figure 6, that subtracts the largest corner rectangle the heuristic can find from the MBR. An example is pictured in Figure 4.

Both operations can be applied recursively using the logic laid out in Figure 7. The structure of a boolean BP is stored in a header byte which every BP in an index implementing tuning must

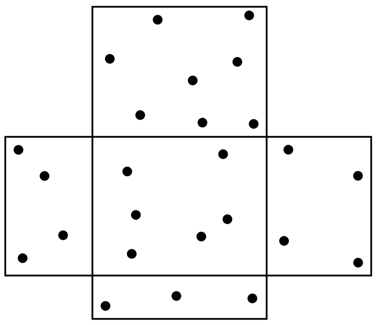


Figure 3: Point Set Bounded by Two Unioned Rectangles

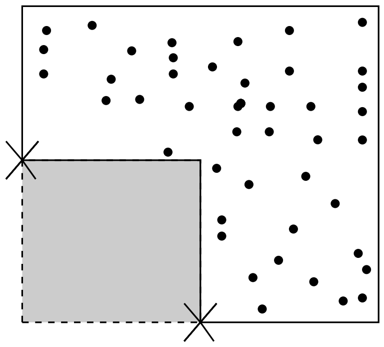


Figure 4: Point Set Bounded by a Rectangle Minus a Second Rectangle

Descriptor	Num. Tuples	Dimensions	Height	Nature	Comment
Blobworld	223105	5	3	real	color histogram data
Colorado [7]	750000	2	3	real	road coordinates in Colorado, USA
Forest Coverage [1]	100000	8	4	real	elevation, slope, shading, etc
Clustered	300000	2, 5	3	synthetic	10000 clusters
Clustered	100000	8	4	synthetic	3333 clusters
Fractal	300000	2	3	synthetic	Sierpinski IFS fractal
Uniform	300000	2, 5	3	synthetic	
Uniform	100000	8	4	synthetic	

Table 3: Experimental Data Sets

have. The addition of a boolean BP component to a BP requires setting two bits in the header, one to indicate that there is another BPC and one to indicate whether the new BP component's relationship with its "partner" BP component is union or minus.

Note that boolean combinations of other shapes could be used analogously if they were more appropriate for a particular application. For example, boolean BPs are a generalization of the idea behind SR-trees [14]. From our perspective, every SR-tree BP is simply a rectangle and a sphere combined by an intersection operator.

5 Node Tuning

Given a particular index node to tune, we must decide which BPs in the node are most deserving of extra bytes, i.e., to which BPs we should allocate some free space to store an extra rectangle. We experimented with three different heuristics to handle node level tuning, which we call *NR* (Node Random), *NG* (Node Greedy) and *NSA* (Node Simulated Annealing).

NR selects a BP to tune by simply picking one BP at random from the set of BPs on the node and tuning it. If the BP has already been tuned into more than one rectangle, randomly select one of the two BP components to tune further, obeying the rules laid out in Figure 7. Once a BP (or BP component rectangle) to tune has been selected, randomly select the operation, TuneMinus or TuneUnion, to attempt. The return value from the TuneMinus or TuneUnion operation will be the change in BP_badness between the new BP and the old. If the change is zero or negative, the new BP is discarded. In either case, the process of selecting and attempting to improve a BP is repeated until we run out of sufficient free space to store another BP component.

TuneUnion

Input: set of data items to describe

Definitions

- bb = the bounding box of the set of data items
- area-value = area[bb(first set)] + area[bb(second set)]
- margin-value = margin[bb(first set)] + margin[bb(second set)]
- overlap-value = area[bb(first set) $\cap$ bb(second set)]

Algorithm TuneUnion

- Invoke Algorithm Split to split the data items into two sets.
- Calculate the bounding box of each new set.
- Calculate the difference between the hyper-volume of bb(input set) and the hyper-volume of [bb(first set) $\cup$ bb(second set)] (overlapping regions only count once towards total hyper-volume).
- Return two new bounding boxes and the calculated difference.

Algorithm Split

1. Invoke ChooseSplitAxis to determine the axis perpendicular to which the split is performed.
2. Invoke ChooseSplitIndex to determine the best distribution into two sets along that axis.
3. Distribute the data items into two sets.

Algorithm ChooseSplitAxis

1. For each axis, sort the data items by the lower then by the upper value of their rectangles and determine all distributions as follows:
   (a) Along each axis, sort the M data items by the lowest dimension, then by the next higher, and so on.
   (b) For each sort $M-2m+2$ distributions of the M data items into two groups are determined, where the k-th distribution ($k = 1, \dots, (M-2m+2)$) has the first $(m-1)+k$) entries in the first group and the remaining entries in the second group.
2. Compute S, the sum of all margin-values of the different distributions.
3. Choose the axis with the minimum S as split axis.

Algorithm ChooseSplitIndex

1. Along the chosen split axis, choose the distribution with the minimum *area-value*. Resolve ties by choosing the distribution with the minimum *overlap-value*.

Figure 5: TuneUnion Algorithm. Most of the logic is copied from [3]; logic altered for TuneUnion is in *italics*.

```
                                  TuneMinus

Input:  set of data items to describe

  1. If the input set is of BPs rather than points, create a set of data points

       • add the corner points of the rectangles in each BP

       • remove the outer corner point of the subtrahend rectangles in minus
         pairings (i.e., the corner point shared with the minuend)

       • replace removed points with corner points on the subtrahend rectangle
         which are on the bounding surfaces of the minuend but are not the removed
         point.  For examples, see the x-marked points in Figure 4.

  2. Find the minimum bounding rectangle of the data points, MBR

  3. Set biggest_volume to zero and max_minus_rectangle = null

  4. For each corner of the MBR

       • For each data point

         (a) Start constructing a sample subtrahend-rectangle apex point, try_point,
             by setting dimension 0 of try_point to the value of dimension 0 of the
             current data point.
         (b) Find all the points, the sub_set, whose dimension 0 values are between
             try_point dimension 0 and the MBR value of dimension 0; if this corner
             is low in dimension 0, this means points lower than try_point, if the
             corner is high, this means points higher than try_point.
         (c) Set the other dimensions of try_point to each equal the lowest (if the
             current corner is high in this dimension, the highest) value in each
             dimension of the points in sub_set.
         (d) Find the hyper-volume of the space between try_point and the current
             MBR corner point.
         (e) If the hyper-volume > biggest_volume, set biggest_volume = the
             hyper-volume, and max_minus_rectangle = a subtrahend-rectangle
             stretching between try_point and the current MBR corner.

  5. Calculate the difference between the hyper-volume of the MBR and the
     hyper-volume of [MBR − max_minus_rectangle].

  6. Return the MBR and max_minus_rectangle as the new BP, and the calculated
     hyper-volume difference.
```

Figure 6: The TuneMinus heuristic for constructing a pair of rectangles, the first minus the second, from a set of points.

$$\begin{array}{lll}
S & \Rightarrow & <\!fullrect\!> \;|\; (<\!union\!>) \;|\; (<\!minus\!>) \\
<\!union\!> & \Rightarrow & S \;\cup\; S \\
<\!minus\!> & \Rightarrow & (<\!fullrect\!> - <\!emptyrect\!>) \;| \\
 & & (<\!minus\!> - <\!emptyrect\!>) \;|\; (<\!union\!>) - <\!emptyrect\!> \\
<\!fullrect\!> & \Rightarrow & \texttt{MBR of some set of data items} \\
<\!emptyrect\!> & \Rightarrow & \texttt{rectangle bounding empty space}
\end{array}$$

Figure 7: Context-Free Grammar for Boolean Bounding Predicate Rectangle Combinations. Note that the $<emptyrect>$ is always a simple rectangle, because there is no benefit in refining rectangles that enclose empty space. The initial S is the minimum bounding rectangle.

```
A → (B ∪ C), where B, C are contained in A
A → (A − B), where B is an empty corner of A
A ∪ B → C, where C is the MBR bounding A and B
A − B → A
```

Figure 8: State Transitions Used for A Single Step of Bounding Predicate Refinement In the Node Simulated Annealing (NSA) Algorithm: A can be either a single rectangle or a combination thereof.

NG tries both TuneMinus and TuneUnion on every existing BP (or component rectangle) on the node and selects the new BP which provides the greatest improvement in BP_badness over its corresponding old BP to replace. NG then repeats this procedure until it runs out of free space.

NSA uses a simulated annealing algorithm [18] to explore the space of BP possibilities. Figure 8 defines the state transitions each BP is allowed. Each step in the simulated annealing process consists of selecting a BP at random from those on the node and attempting a state transition. If the BP has already been tuned into multiple components, one of the subcomponent rectangles is randomly selected, excluding the *emptyrect* rectangles in a minus operation. Once a BP or combination of BP components has been selected, a state transition from Figure 8 is selected and applied. If the transition results in a reduction of BP hyper-volume, it is saved. If it does not, the simulated annealing oracle is consulted, and the new BP is saved with probability $e^{(-\Delta V/Temperature)}$.

The difficulty with NSA is tuning it. If the initial temperature is too high and the cooling rate too slow, it becomes, effectively, an inefficient exhaustive search. Because the number fed to the simulated annealing oracle is the change in hyper-volume between an existing and a proposed BP, the proper initial temperature is dependent upon the data set. We chose to set the initial temperatures to be $.00096 \times \prod_{i=1}^{d}(max(i) - min(i))$ where d is the dimensionality of the data set, $max(i)$ is the maximum value in that dimension and $min(i)$ is the minimum value. Examination of the return values from the simulated annealing oracle showed that these temperatures resulted in reasonable cooling; the oracle started by returning *true* frequently and ended by returning *false* frequently.

For each combination of tuning algorithm and node prioritization algorithm, we ran experiments over all of the data sets listed in Table 3. Each data set was sorted using the Sort-Tile-Recursive algorithm from [16] and bulk-loaded into an R*-tree. For each data set we ran and analyzed nearest neighbor queries centered around ten percent of the total data points, selected at random from the data set, before we altered anything in the R*-trees, and analyzed the same query workload again after tuning nodes.

Figures 9 and 10 show the results of these experiments comparing the performance of the node level algorithms over various data sets; graphs generated from experiments over the other data sets (shown in Appendix A) show similar results. These graphs reflect the extent to which our techniques brought R*-tree performance toward ideal performance by eliminating wasted excess coverage loss I/Os. Figure 9 directly examines the effects of tuning on excess coverage loss I/Os; Figure 10 shows how that translates to effects on total I/Os performed while running the test queries. As you can see, 30% or more of total I/Os are eliminated for both of the eight-dimensional data sets (Forest Coverage and Clustered 8D).

NSA outperforms NG and NR for this data set and for most of the other data sets, with the differences growing more pronounced as the dimensionality of the data grows. NSA can take more than three times as long to tune a single node as NR (on a lightly loaded Sun Sparc Ultra 10 station, 256 MB of memory); NG is slower than NSA. The choice between NSA and NR is one of quality vs. tuning time. We will see in the next section that doing a good job tuning only a few nodes in the tree is often sufficient. Hence the quality benefits of NSA are probably affordable in practice.

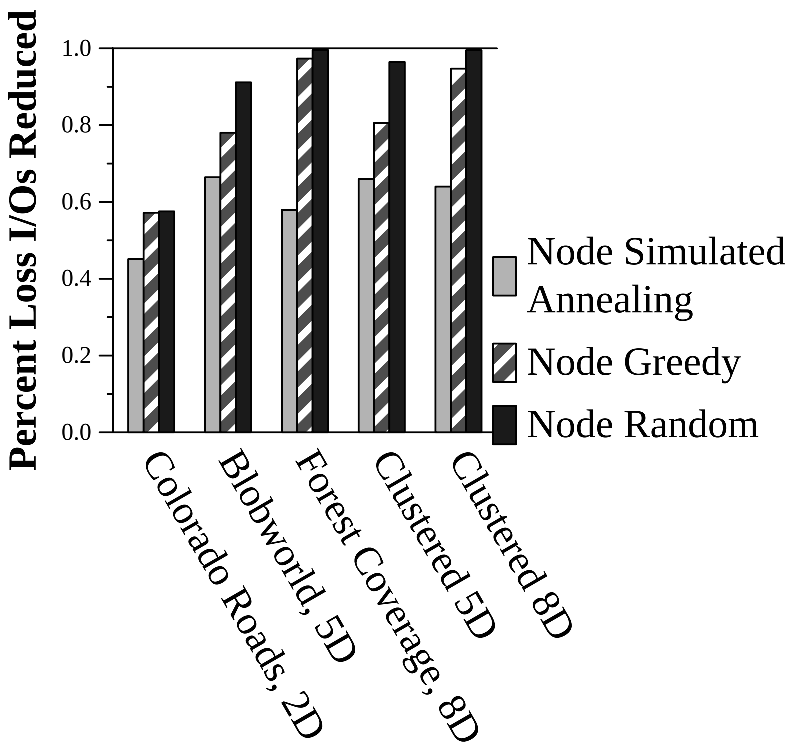


Figure 9: Performance of Node Tuning: Percent change in *excess coverage loss* I/Os. Results after all nodes of indices over various data sets have been tuned. As you can see, NSA consistently outperforms the other algorithms at reducing the number of query workload I/Os. The differences in percentage improvement between NG and NSA range from 12 to 39%; between NR and NSA the differences range from 12 to 41%. In both cases, the greatest difference was in experiments with the 8-dimensional, real-world Forest Coverage data set; the least difference was in the 2-dimensional, real-world Colorado road data.

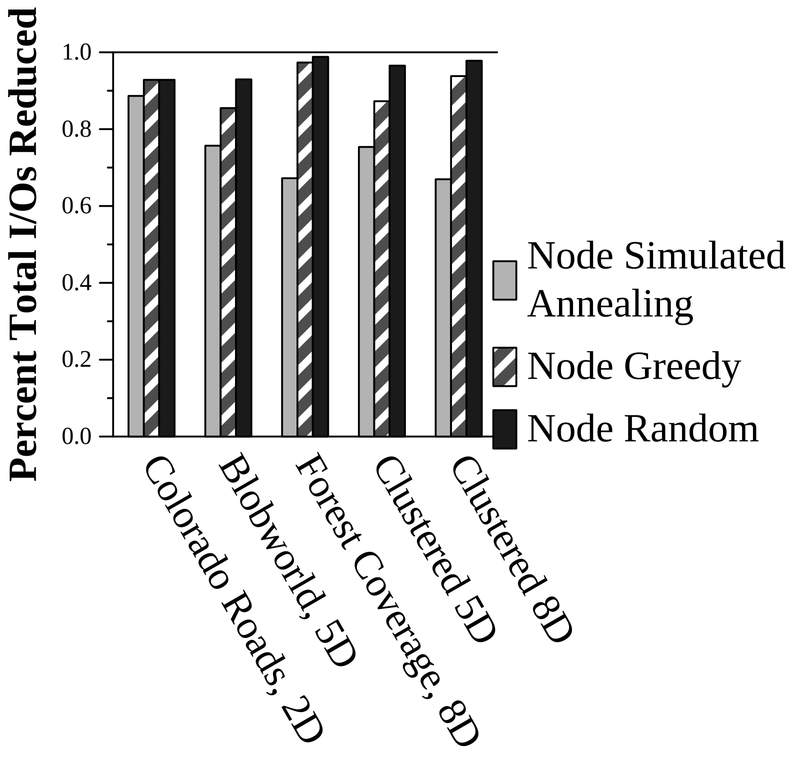


Figure 10: Performance of Node Tuning: Percent change in *total* I/Os. Results after all nodes of indices over various data sets have been tuned. As you can see, NSA consistently outperforms the other algorithms at reducing the number of query workload I/Os. The differences in percentage improvement between NG and NSA range from 4 to 30%; between NR and NSA the differences range from 4 to 31%. In both cases, the greatest difference was in experiments with the 8-dimensional, real-world Forest Coverage data set; the least difference was in the 2-dimensional, real-world Colorado road data.

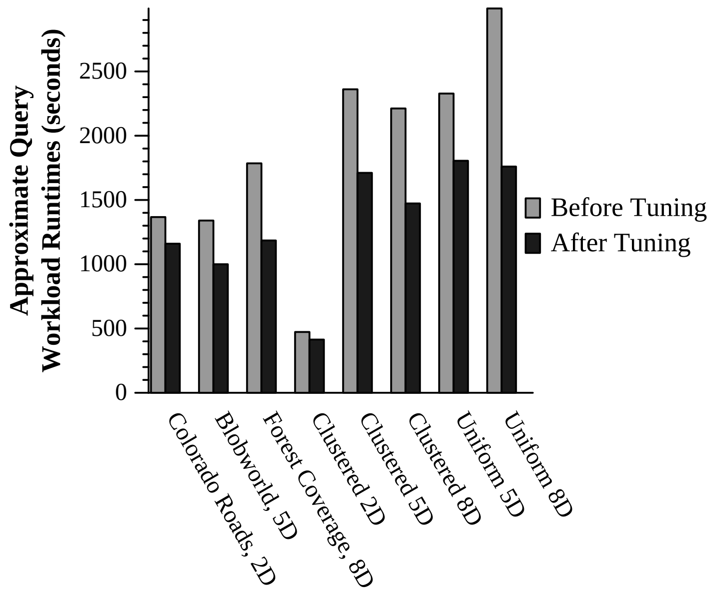


Figure 11: Performance of Node Tuning: Change in approximate query workload runtimes (seconds) for various data sets, after all nodes have been tuned using NSA.

6 Node Prioritization

To accomodate the situations where there is not enough time to tune all the nodes in an index, we require a means of selecting the index nodes whose tuning will result in the greatest reduction in overall workload I/Os. When we do not know our tuning time budget in advance, we are interested in the *differential benefit* of tuning a single node – i.e., we want to maximize the slope of the graph where the x-axis is the number of nodes tuned so far and the y-axis is the number of I/Os saved.

Perfect node prioritization would require knowledge of the number of I/Os performed for a query workload before and after each possible set of nodes has been tuned. While it is possible to collect such knowledge by tuning each possible set of nodes and analyzing a test query workload, that would be prohibitively expensive. Therefore, we use heuristics to maximize the benefit of each node tuning step. We tested the performance of three heuristics that order nodes for tuning: Strict Sort (SS), Strict Sort and Climb (SSC) and Crop Root (CR).

For *SS* (Strict Sort), we tune the nodes in descending order of node_badness. SS does not require that the number of nodes to be tuned be known in advance – this is useful if, for example, a total tuning time is specified, but per-node tuning time is not well calibrated.

Our next heuristic, *SSC* (Strict Sort and Climb), requires a fixed budget to work correctly. Given a budget of B nodes to tune, SSC picks $b < B$ nodes with highest node_badness such that they have at most $B - b$ distinct ancestors and b is as large as possible. There may be a small (less than the height of the tree) number of unallocated tuning operations left if there is not enough room in the budget to make it from the node with the next highest node_badness all the way to the root. These remainder tunings are allocated to the nodes with the highest excess coverage losses not already in the set to be tuned. The B selected nodes are sorted by height in the index and tuned from the bottom of the index tree up, in order to allow the increased precision of BPs lower in the tree to be reflected when the BPs higher up are tuned.

Because we noted that sharp drops in the number of I/Os performed over a workload frequently occurred just after the root node had been tuned, we tried the Crop Root (CR) ordering, which simply tunes the root node of the index tree. CR and SS have an advantage over SSC in that they do not require a budget B of nodes to be known in advance.

Figures 12 and 13 show the results of experiments comparing the performance of these heuris-

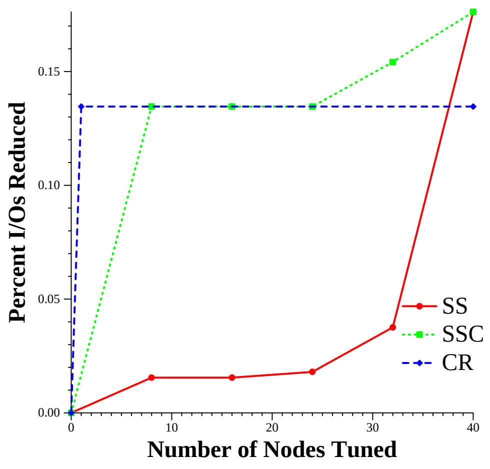


Figure 12: Performance of Node Prioritization: Percent change in excess coverage loss I/Os. Results after all nodes of Blobworld data set tuned using NSA algorithm

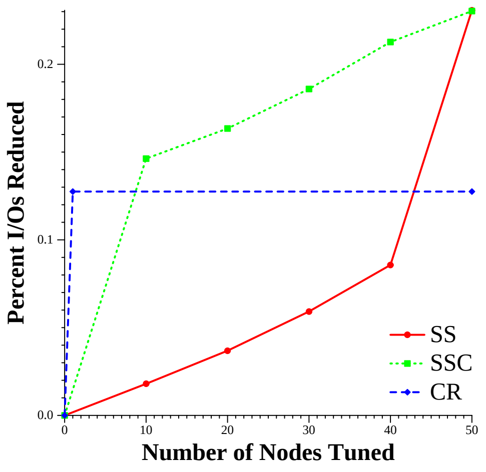


Figure 13: Performance of Node Prioritization: Percent change in excess coverage loss I/Os. Results after all nodes of Clustered 5D data set tuned using NSA algorithm

tics.[3] The algorithm which performs the least amount of work, CR, achieves almost as great I/O savings as SS and SSC achieve. The sharp rise in the SS line occurs after the root is tuned. In view of the fact that 13+% of the total I/Os (21% of the excess coverage loss I/Os) in the Blobworld query workload are eliminated through only tuning one node, the root, and that similar results were achieved for the other data sets, we believe the Crop Root algorithm is the best to use in the absence of time to tune all of the nodes.

7 Related Work

The traditional way to improve the query performance of an index is to drop and rebuild it. We have introduced a new index operation, tuning, which improves the performance of even a freshly built index, and which could be applied to many varieties of tree-based multi-dimensional indexes, examples of which can be found in [6, 9].

Similar work has been performed for B-trees [21], though that research focused on compacting and moving data items between leaf-level nodes, then updating the upper levels of the tree. Our work focuses on modifying the upper levels of the index tree without, at present, touching the leaf level. In terms of the vocabulary presented in Section 2, the work of [21] focussed on utilization loss, while we chose to address excess coverage loss because our experiments showed that utilization loss was not the most pressing problem with bulk-loaded multidimensional indexes. Also, their work is only applicable to inherently one-dimensional B-trees; our work is only applicable to indexes with data dimensionality greater than one.

The hB-tree [17] index is related to our tuning work in that it, too, explicitly represents empty data space within an index node by subtracting ("extracting") it from the data space covered by subtree, though they do so by describing all the children of a given inner node through a kd-tree [4] stored on each inner node.

The hB-tree approach and our approach to indexes are different in many ways. hB-trees use a single data structure, a kd-tree, on each inner node to describe all the children of that node; we use list of small structures, the bounding predicates, one per child node. The hB-tree kd-tree node structures' efficiency depends on the shape of the child nodes and the order in which data was inserted; in this paper we bulk-load our indices so that the data on each node is well-grouped and large gaps have been minimized.

[3]Appendix B contains more result graphs.

In our terminology, hB-trees use minus to describe empty regions in the data space an inner node covers, though they allow minus anywhere within that space instead of restricting it to the corners. (However, their proof that hB-trees achieve good space utilization only applies to the case where all extracted regions are at corners.) We allow the use of both minus and union operations.

We use the operators union and minus to build bounding predicates that describe child nodes. The kd-trees at hB-tree inner nodes have "holes" in their descriptors that are like our minus. Since hB-trees are actually directed acyclic graphs, rather than true trees, multiple inner nodes can point to the same child node; the union of the regions in each of those inner nodes can be used to describe the child node data, so hB-trees have union capabilities, in a roundabout fashion. hB-trees do not allow data in leaf-level nodes to overlap. While boolean bounding predicate logic does not concern itself with leaf node data overlapping or not, the R*-tree we implemented BBP logic in does allow leaf-level overlap.

The greatest difference between the two approaches is that boolean bounding predicate indexes actively perform tuning on inner nodes in order to shape the bounding predicates to correspond precisely to the shape of the existing data on the child nodes. hB-trees rely on splitting well when new data is inserted for their kd-tree inner node structures to develop accuracy.

8 Future Work

As future work, we plan to address the issues surrounding:

- Dynamic data: When a new data item is inserted onto a tuned – and therefore "full" node – should we "un-tune" a bounding predicate to make space, or split the node?
- The "Badness" Metric: We use hyper-volume as an approximation to the change in I/Os to perform queries. Is there a more intelligent, yet still computationally efficient, alternative?
- Data Movement: We currently do not alter anything other than the BPs of an index. However, situations may exist where minor movements of data items from one leaf node to another would lead to significant improvements in a boolean bounding predicate's precision. Is this the case? If so, how can we efficiently take advantage of that?
- Intersection: Our boolean BPs currently only use minus and union operators. Would adding intersection improve their performance?
- Auto-tuning: We have been triggering tuning operations manually. We would like to develop techniques whereby an index could detect "quiet times" and use them to prioritize and tune nodes without database administrator involvement.

9 Conclusions

We have presented new algorithms for spatial indexes, aimed at reducing the number of unnecessary I/Os an index performs during queries by making use of what free space may exist on index nodes to store more precise bounding predicates. We tested our algorithms using predicates that had simple rectangles as their building blocks, combined with the boolean operators union and minus. Tests show that 30% or more of the I/Os performed by a query workload can be eliminated, with most of the benefit achieved through simply tuning the root node.

Acknowledgments

This research was supported by fellowship stipend support from the National Physical Sciences Consortium and Lawrence Livermore National Laboratory. We thank Christos Faloutsos for the software we used to generate our fractal data and Erik Demaine for the initial pointer to constructive solid geometry. Conversations with Kris Hildrum provided many useful insights.

References

[1] UC Irvine KDD Archive, 2000. http://kdd.ics.uci.edu/.

[2] C.B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[3] 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 Int'l Conference on Management of Data*, pages 322–331, 1990.

[4] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering, SE-5*, 4:333–340, July 1979.

[5] Blobworld image retrieval system. http://elib.cs.berkeley.edu/photos/blobworld/.

[6] C. Bohm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces - index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[7] United States Census Bureau TIGER database, 1999. http://www.census.gov/geo/www/tiger/.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, 2000.

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

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

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

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

[13] T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47:45–76, August 1993.

[14] N. Katayama and S. Satoh. The SR-tree: An index structure for high dimensional nearest neighbor queries. In *Proc. of the ACM-SIGMOD Int'l 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 Int'l Conference on Data Engineering*, pages 497–506, New Orleans, LA, April 1997.

[17] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[18] W. Press, S. Teukolsky, W. Vettering, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, 2nd Ed.* Cambridge University Press, 1992.

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

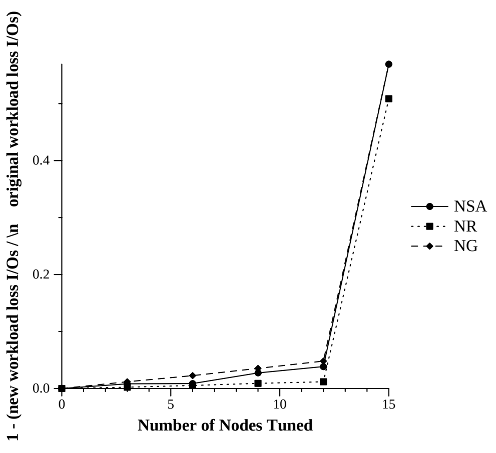
[20] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of the 12th IEEE Int'l Conference on Data Engineering*, pages 516–523, New Orleans, LA, February 1996.

[21] C. Zou and B. Salzberg. On-line reorganization of sparsely-populated B+-trees. In *Proc. of the ACM-SIGMOD Int'l Conference on Management of Data*, Montreal, Canada, June 1996.

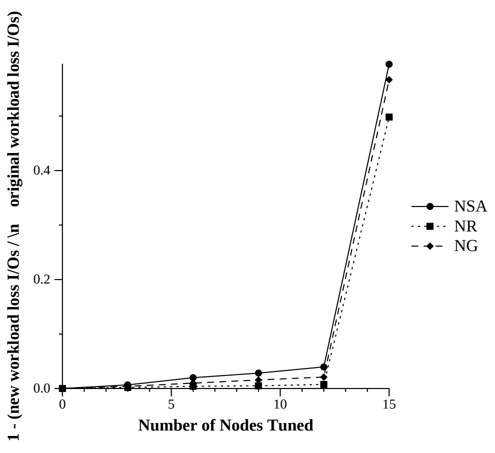
A Node Tuning Algorithm Performance

These graphs compare the performance of the node tuning algorithms NR, NG and NSA, showing how well they reduce the number of query workload I/Os lost to excess coverage loss. The results shown here are all from experiments performed using the SS (Strict Sort) ordering. The sharp upward jags many of the lines make towards the high end of the x-axis reflect the point at which the root node of the index is tuned.

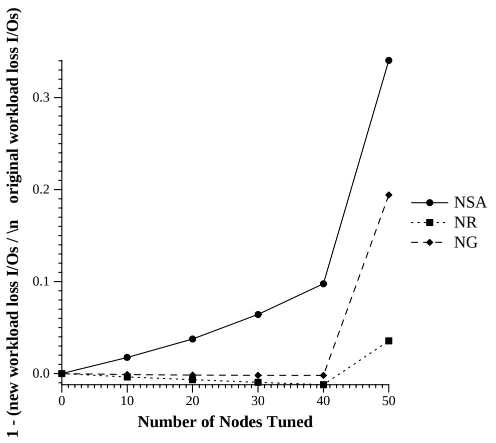
Synthetic Data Sets:



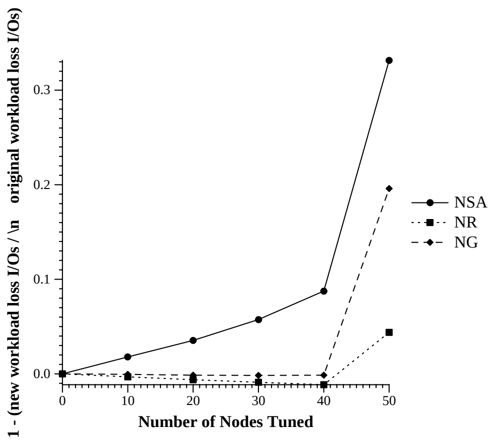
2D Clustered Data



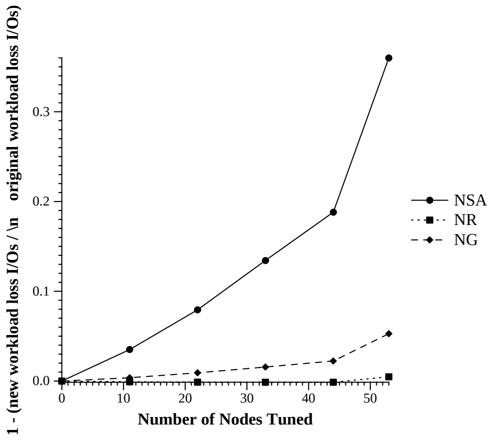
2D Uniform Data



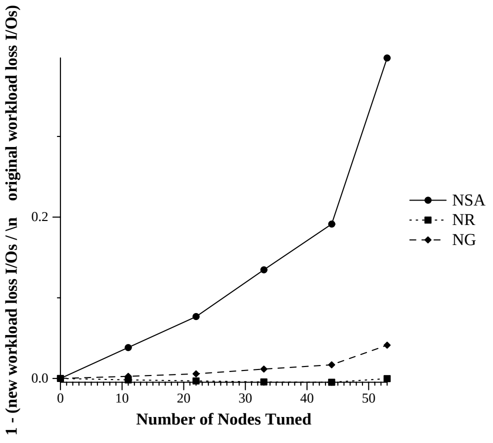
5D Clustered Data



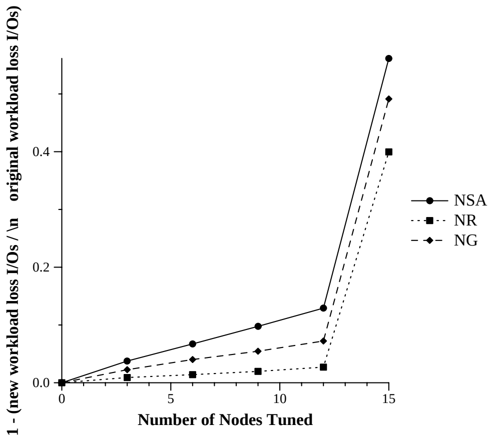
5D Uniform Data



8D Clustered Data



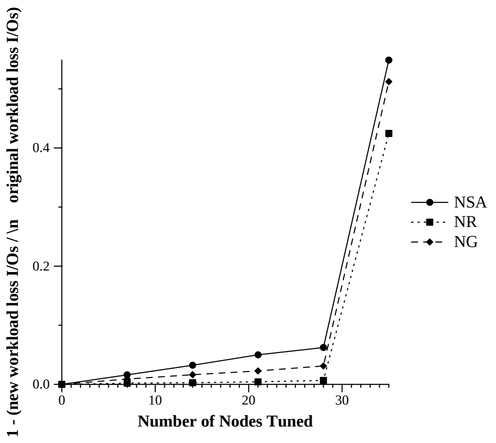
8D Uniform Data



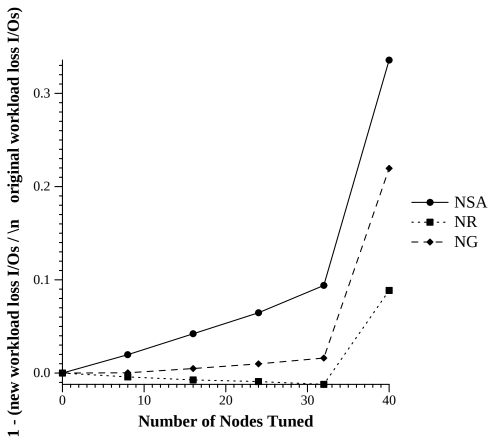
2D Sierpinski Fractal Data

Real Data Sets:

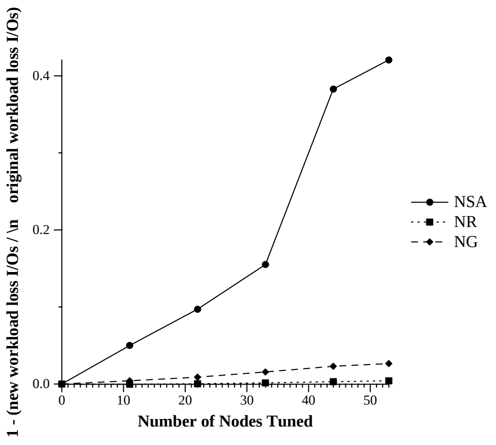
Note that the results of experiments run over real data sets correspond quite closely with the results of experiments over synthetic data sets of corresponding dimensionality.



2D Colorado TIGER Data



5D Blobworld Color Data

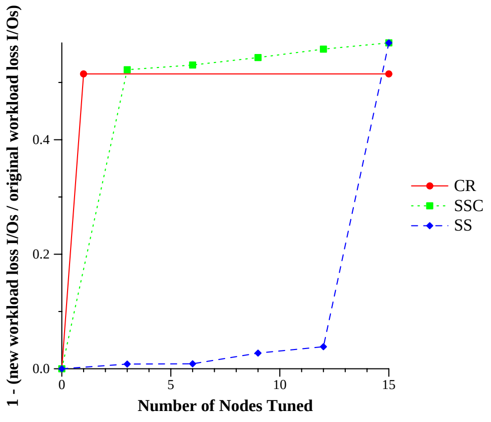


8D Forest Coverage Data

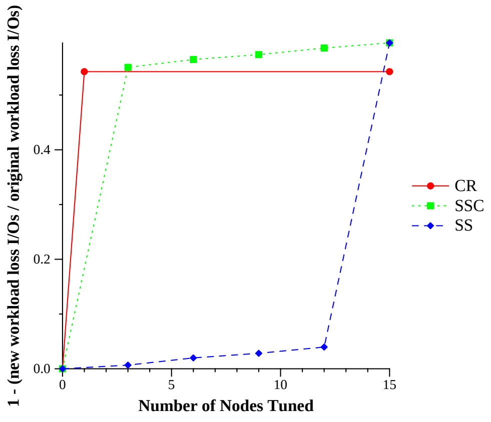
B Node Prioritization Algorithm Performance

These graphs compare the performance of the node prioritization algorithms SS, SSC, and CR, showing how well they reduce the number of query workload I/Os lost to excess coverage loss. The results shown here are all from experiments performed using the NSA (Node Simulated Annealing) node tuning algorithm.

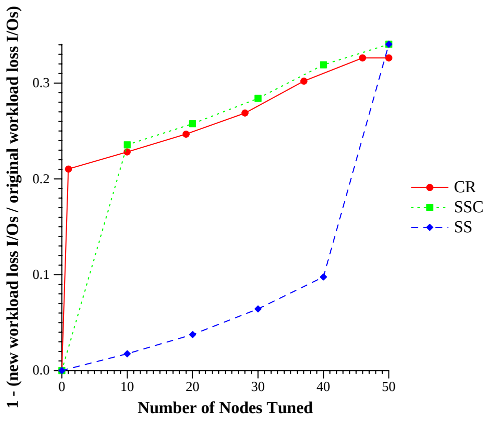
Synthetic Data Sets:



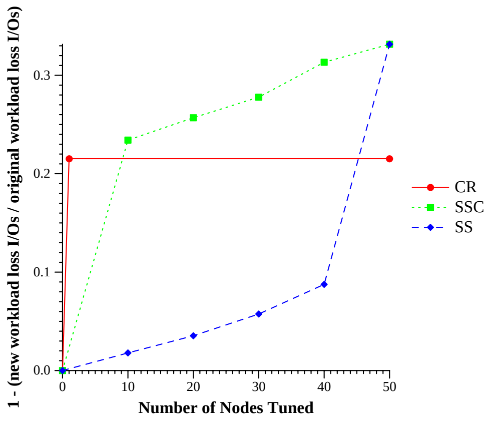
2D Clustered Data



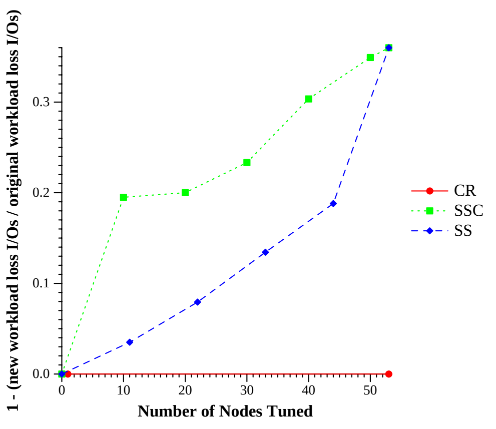
2D Uniform Data



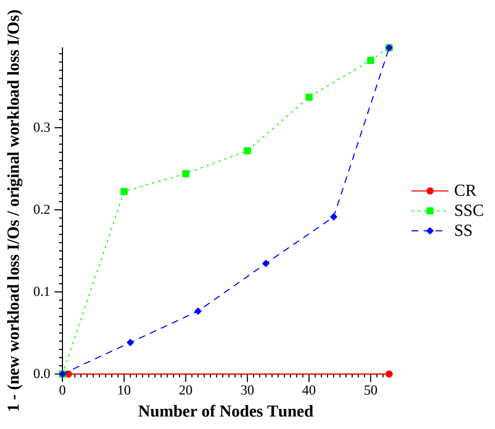
5D Clustered Data



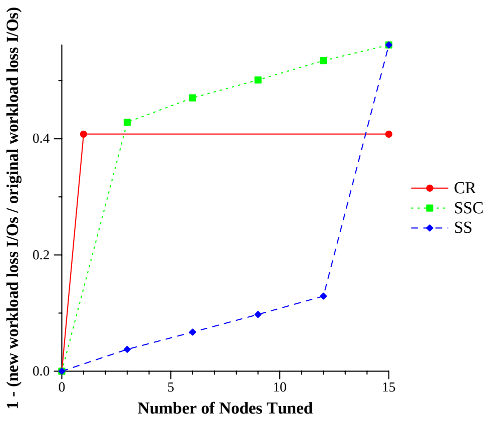
5D Uniform Data



8D Clustered Data



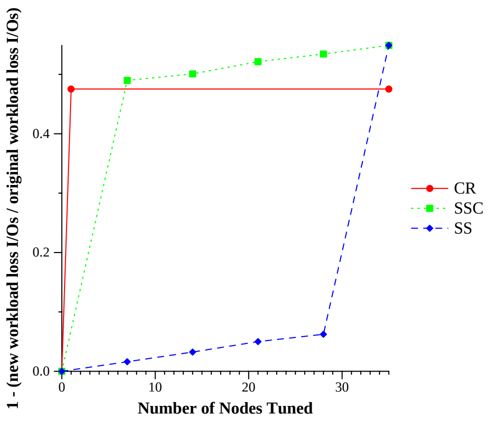
8D Uniform Data



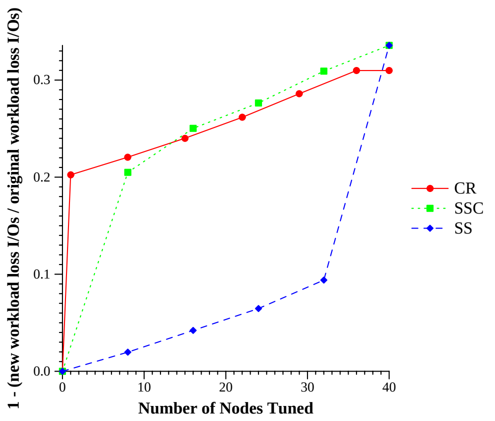
2D Sierpinski Fractal Data

Real Data Sets:

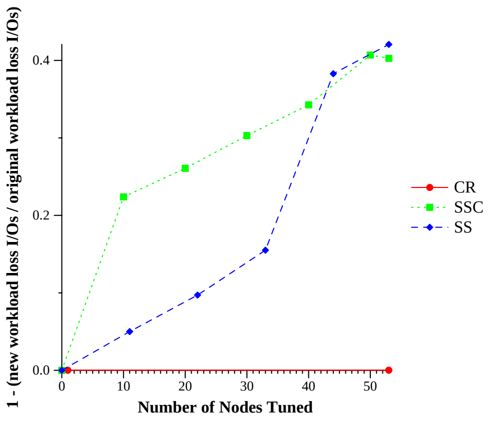
Note that the results of experiments run over real data sets correspond quite closely with the results of experiments over synthetic data sets of corresponding dimensionality.



2D Colorado TIGER Data



5D Blobworld Color Data



8D Forest Coverage Data