Near-Neighbor Query Performance in Search Trees

Shuanhu Wang, Joseph M. Hellerstein, and Ilya Lipkind
University of California, Berkeley*

Abstract

Near-neighbor search is an increasingly important operation for queries over multimedia, text, and other non-standard datatypes. In large databases, near-neighbor searches must be enhanced by indexed retrieval for efficiency. In this paper, we present a detailed analysis of three proposals for near-neighbor search: one based on the R-tree, and two which motivated the invention of new trees, namely the SS-tree and SR-tree. We find that while the new trees do improve performance, the reason for their improvement comes mostly from a new Penalty metric, and not from a variety of other details in their implementation. Our analysis was done using a Generalized Search Tree, which both allowed us to easily do a fair comparison, and also provided the framework for a clearer analysis of the issues at hand.

1 Introduction

Near-neighbor search is a technique required by a variety of new applications. Given a particular data item, near-neighbor search finds the k closest matches to the item and outputs them in order of proximity. Example applications that use near-neighbor search include full-text document retrieval, image retrieval, geographic information systems, and sequence alignment in bioinformatics programs.

A natural way to handle "rough" queries over non-standard data types is to map data to points in a multidimensional space, and map rough queries to points in the same space. Then a natural Euclidean version of the near-neighbor problem can be used to retrieve data in the appropriate order. This is done in document retrieval, for example: each document is mapped into a multidimensional vector space, and a query (itself a "document" consisting of a set of keywords) is similarly mapped into the same space – documents "close" to the query in the vector space are returned as matches, ranked by their proximity. In order to get acceptable performance in large database systems, multidimensional vector data is typically indexed by a spatial data structure like the R-tree [4] or one of its related enhancements, e.g. the R*-tree [2].

Although R-trees and their variants were originally intended to support multidimensional selection predicates, Roussopoulos et al. presented a near-neighbor search algorithm [9] that works for R-trees (and hence for the superior R*-trees). One year after that paper, White and Jain proposed a somewhat different data structure called the SS-tree, which – using the search algorithm of [9] – outperformed the R*-tree [11]. Just one year later, Katayama and Satoh presented the SR-tree, which achieved even better performance with the same search algorithm [8].

*Current addresses: wangsh@pangea.stanford.edu, jmh@cs.berkeley.edu, ilipkind@uclink.berkeley.edu. This work was supported by NASA grant 1996-MTPE-00099, and NSF grant IRI-9703972.

Name	BP Shape	Reinsertion	Penalty	PickSplit
R*-tree	□	R*	R*	R*
SS-tree	○	SS	SS	SS
SR-tree	○, □	SS	SS	SS

Table 1: A summary of previous indexing tree structures.

While these results represent swift practical progress, the conclusions to be drawn from this body of work are somewhat unclear. The data structures vary in four different ways, and the effects of each individual variation on performance are not clear. In essence, the performance results to date are not sufficiently detailed to generate a clear understanding of the underlying issues.

This paper rectifies that problem by presenting a detailed analysis of the performance distinctions between these trees. We implemented all three trees using a Generalized Search Tree (GiST) library [5], which allowed us to easily "mix and match" techniques proposed for each tree, and fairly compare the performance differences. Our analysis confirms the results presented in the previous papers, and also shows that (1) most of the performance benefits of the SS-tree arise from only one of the four parameters it modified in the R*-tree (*i.e.* the Penalty metric), and (2) the rectangular shape of R*-tree keys is generally advantageous for near-neighbor search, particularly in high dimensional spaces. The combination of these results suggests that R*-tree with a modified Penalty metric is an attractive solution for near-neighbor search. Our study also highlights the advantages of the GiST framework for access method development and analysis, and suggests that future efforts in access method design should shift focus: the invention of "new" trees no longer seems as important as the careful analysis of the individual factors affecting tree performance.

2 The Search Trees

The GiST software reduces the distinctions between search trees to a matter of search key type, and a few simple "extension" methods. This minimizes the code required to generate a new search tree. An added advantage is that the differences between search trees can be clearly stated by focusing on the GiST extension methods. To describe the differences between the trees we study here, we outline the salient GiST extensions of each, including both the data in the tree's search keys, and the methods on those keys. These features are summarized in Table 1. We focus on the R*-tree, SS-tree, and SR-tree.

- *Bounding predicates:* Internal nodes in each search tree contain search keys, which are perhaps better described as *bounding predicates* (BPs). A BP is associated with a pointer, and can be thought of as a compressed representation of all data items at the leaves of the subtree referenced by the pointer. Since BPs typically must be small in order to provide high fanout, they typically represent *lossy compressions* of the set of data below them – lossy in the sense that they over-generalize. The danger of being over-general is that searches may delve into subtrees that contain no relevant data.

  R*-tree BPs are bounding hyperrectangles, *i.e.* ranges from *low* to *high* in each of n dimensions. The SS-tree uses bounding hyperspheres, *i.e.* an n-dimensional point and a radius. All points below a BP's pointer are contained in the corresponding hyperrectangle or hypersphere. The SR-tree maintains both a bounding hyperrectangle and hypersphere; the conjunction of structures allows the SR-tree to prune searches in every situation that either the SS-tree or the R*-tree can. Moreover, the information provided by the hyperrectangles allows the SR-tree to maintain smaller hyperspheres than the SS-tree.

- *Penalty Metric:* In all the trees we consider here, BPs in a given node may overlap. When a new entry is to be inserted below a particular internal node of the tree, the insertion algorithm has to choose a subtree to hold the new entry. The Penalty metric uses the BPs in the node to estimate the overhead of placing a new entry into a particular subtree; the subtree of least Penalty is chosen for insertion.

  The R*-tree Penalty metric uses two different techniques, one for insertion into leaf nodes, and another for insertion into internal nodes. When the R*-tree Penalty metric is invoked on a BP B pointing to a leaf nodes, it measures the area of overlap a between B and all other entries on the node. It then computes B', the value B would change to if the new entry were inserted in B's subtree, and measures the area of overlap a' between B' and all other entries on the node. The R*-tree penalty in this case is $a' - a$. When the R*-tree Penalty metric is invoked on a BP B at a higher level of the tree, it computes B', and returns the difference in (hyper-)volume between B' and B.

  For a BP B, the SS-tree's Penalty metric measures the Euclidean distance of the new item to the center of the hypersphere B. The SR-tree also uses the SS-tree's Penalty metric.

- *Reinsertion Policy:* Multidimensional search trees can become poorly clustered over time. The R*-tree introduced a heuristic to alleviate this problem. When an R*-tree node n overflows, it is not split; instead, 30% of the entries on n (those furthest from the center) are reinserted into the tree one by one. If a reinserted entry causes an overflow in some node m at the same level of the tree as n, then m is split.

  The SS-tree proposed a more aggressive splitting policy. If a node m overflows while reinserting nodes from n, the SS-tree recursively calls the reinsertion code, reinserting 30% of the elements in m rather than splitting. The SS-tree only splits a node m if the overflowing insertion *to* m is a recursive call of a reinsertion *from* m. The SR-tree also uses the SS-tree's reinsertion algorithm.[1]

- *PickSplit Algorithm:* When the insertion algorithm decides to split a page, the PickSplit method is called to determine which entries remain on the old page and which are moved to a new page. The R*-tree picksplit algorithm attempts to minimize the hyper-surfaces (*margin*) of the two resulting nodes, and also the volume of the overlap between the two nodes. To do this, it considers splitting along each dimension in turn; the dimension which provides some way to split with minimum sum-of-margins (the margin of the old node's bounding hyperrectangle plus the margin of the new node's bounding hyperrectangle) is chosen for splitting. Once the dimension is chosen, all possible splits along that dimension are considered, and the one of minimum overlap-volume is chosen.

  The SS-tree finds the dimension with the largest variance of center-coordinate across all entries in the original node. It splits the entries up along this dimension in a way that minimizes the variance of the center-coordinates in the resulting nodes. The SR-tree uses the SS-tree's PickSplit algorithm.

[1] The reinsertion policy was not proposed as an extension method in the original paper on GiST [5]; this made our implementation of the SS-tree reinsertion policy somewhat inconvenient. We are currently exploring the extension of reinsertion to more general tree reorganizations, which we hope will be more beneficial and flexible than reinsertion policies, and will interact better with concurrency control protocols like that of [7].

2.1 Discussion

The Penalty, Reinsertion and PickSplit methods primarily affect the way that an index tree organizes itself; the data in the BPs affects the behavior of the search algorithm. If a particular tree performs poorly, it could be the result of one or both of the following reasons:

1. *Poor clustering:* If the data in the tree is not clustered well, then a search algorithm will have to "jump" from leaf node to leaf node to satisfy a query. Such a problem can only be addressed by better implementations of Penalty, Reinsertion and PickSplit.

2. *Poor BPs:* Even if the data in the tree is well clustered, if the BPs are not sufficiently accurate then a search algorithm may visit many nodes unnecessarily. Such a problem can be addressed by storing more descriptive information in each BP to minimize the "lossiness" of the BP's compressed representation of its subtree. Note however that the storage requirement of the BPs determines the fanout of internal nodes of the tree, so it is advantageous to keep the BPs small.

Table 1 clearly shows that the SS-tree and SR-tree are very similar: they differ only in that the SS-tree keeps more data in its bounding predicates than the SR-tree. Thus if an SS-tree clusters a particular dataset poorly for near-neighbor queries, an SR-tree will do the same. The performance differences between these two trees result primarily from (1) the difference in fanout, and (2) the enhanced ability of the SR-tree to guide the near-neighbor search algorithm. The R*-tree, on the other hand, provides different Penalty, PickSplit and Reinsertion methods, and hence organizes data into leaf nodes quite differently.

2.2 Framework of This Study

Our categorization of the previously-proposed search trees suggests an entire family of possible "new" trees, corresponding to different choices along each of the four columns in Table 1. In table 2 we enumerate all the possible trees one could construct from the BPs and extension methods used in previous work. We also provide a preview of our results by checking off trees that demonstrate acceptable performance for some workloads.

In the remainder of the paper we compare all the trees in this table. Our goal is not merely to find the combinations that work best, but to *isolate the factors that actually affect performance.* For the extension methods, this can be done via simply trying out the different combinations of methods and seeing how the resulting trees differ in performance. For the BPs, we wish to understand both the benefit of additional information, and the performance cost of reduced fanout.

The entries in Table 2 present the naming scheme we use throughout the paper. The prefix of the tree name is R*T, SST, or SRT, and corresponds to the BP shape of the tree (rectangle, sphere, or both, respectively). Then each tree name has three "bits" which describe its Reinsertion, Penalty, and PickSplit method, in that order. Bit value "r" corresponds to the R*-tree implementation of the particular method, while "s" corresponds to the SS-tree implementation.

Name	BP Shape	Reinsertion	Penalty	PickSplit	Competitive
R*Trrr (R*-tree)	□	R*	R*	R*	
R*Trrs	□	R*	R*	SS	
R*Trsr	□	R*	SS	R*	√
R*Trss	□	R*	SS	SS	√
R*Tsrr	□	SS	R*	R*	
R*Tsrs	□	SS	R*	SS	
R*Tssr	□	SS	SS	R*	√
R*Tsss	□	SS	SS	SS	√
SSTrrr	◯	R*	R*	R*	
SSTrrs	◯	R*	R*	SS	
SSTrsr	◯	R*	SS	R*	√
SSTrss	◯	R*	SS	SS	√
SSTsrr	◯	SS	R*	R*	
SSTsrs	◯	SS	R*	SS	
SSTssr	◯	SS	SS	R*	√
SSTsss (SS-tree)	◯	SS	SS	SS	√
SRTrrr	◯, □	R*	R*	R*	
SRTrrs	◯, □	R*	R*	SS	
SRTrsr	◯, □	R*	SS	R*	√
SRTrss	◯, □	R*	SS	SS	√
SRTsrr	◯, □	SS	R*	R*	
SRTsrs	◯, □	SS	R*	SS	
SRTssr	◯, □	SS	SS	R*	√
SRTsss (SR-tree)	◯, □	SS	SS	SS	√

Table 2: Structures considered in this paper. Structures with a check mark performed competitively for some workload.

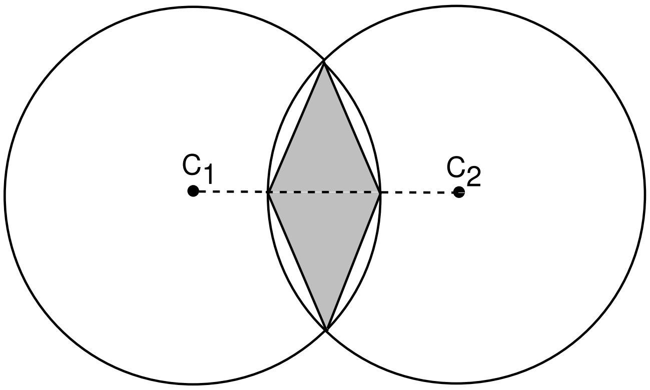


Figure 1: An approximation of the overlap area of two circles.

2.3 Measuring the Overlap of Spheres

One minor complication arises in constructing SS-trees that use the R*-tree PickSplit algorithm. That algorithm needs to compute the volume of overlap of BPs, and this is geometrically complex for spheres. In our implementation we approximate this volume by the volume of the hyper-parallelopiped that is enclosed by the overlapped area. This is illustrated for two-dimension in Figure 1, where the shaded area is taken as a rough estimation of the overlap area. While this estimations is rough (particularly in high-dimensional spaces), we will see that the resulting inaccuracy in PickSplit has little effect on overall performance.

2.4 Near Neigbor Search Algorithm

In this section, we briefly describe the near neighbor search algorithm proposed in [9], which was used in the previous work on near-neighbor search in R*, SS and SR-trees. We sketch the algorithm for completeness; the interested reader should consult the original near-neighbor search paper and subsequent work [6, 1] for further details. The algorithm of [9] traverses the tree in a branch-and-bound fashion, returning the k points closest to the query point. At all times it maintains a list of the k nearest points seen so far and their distances; these are intialized to virtual points at infinite distance. The algorithm also maintains a priority queue of nodes to visit; this is initialized to contain the root node. The algorithm traverses the tree by always visiting the first node on the priority queue. If the current node it is visiting is a leaf, the algorithm first updates the list of k nearest points based on the points contained in the leaf. If it is not a leaf, it orders all the BPs on the node using a metric MINDIST, defined as the smallest possible estimate of the minimum distance between the search point and any point in the BP. The BP with the smallest MINDIST is hoped to contain more points close to the search point; hence the priority queue is maintained ordered by MINDIST. Another metric MINMAXDIST, which measures the smallest possible upper bound between the query point and all points in the BP, is used together with MINDIST to prune some of the BPs prior to the visit. The algorithm terminates when the priority queue is empty, at which point the k nearest neighbors are returned.

Although the traversal algorithm is the same for all the trees, search performance depends heavily on the accurate estimation of the MINDIST and MINMAXDIST metrics. As we will see, the performance improvement of SR-tree over SS-tree comes directly from a better estimation of these two metrics, even though SR-tree has a smaller fanout in the non-leaf levels.

3 Experimental Study

We implemented the trees described above in C++ using the libGiST package, version 0.9b1 [3]. LibGiST includes the R*-tree reinsertion policy as an option, and we enhanced it to offer the SS-tree reinsertion policy as well. The trees were stored in UNIX files, and no buffer management was used in our experiments, which simply measure I/O requests from libGiST to the file system. The machine used for the experiments was a Dell PowerEdge 4100 Server, with 2 200Mhz Pentium Pro processors, 256 Mb of RAM, and one 4Gb hard drive, running the SunOS 5.5.1 operating system.

3.1 Datasets

In our experiments we use both synthetic and real point datasets. We generate synthetic data in 3, 10 and 16 dimensions, with the range for each dimension being [0,1]. We examine the performance of the trees with varied dataset sizes, from 10,000 to 100,000 entries, increasing by intervals of 10,000 entries. Each of the 10,000 entries contains 100 clusters, with cluster centers distributed randomly in the space. Each cluster contains 100 points, distributed randomly in a small hypercube with length of 0.1 in each dimension. These datasets are modeled on those of [8], but differ slightly in that the clusters of [8] are distributed randomly in a hypersphere instead of a hyperrectangle. Uniform datasets are also used for test, but are found to be unsuitable for evaluating index structures [8] in high dimensions, and are not shown in this paper. All of our queries fetch the 21 nearest neighbors of a point, and the results are averaged over 1000 queries.

Our real dataset consists of latitude/longitude coordinates of 1,355,825 place names in the United States, from the USGS Geographic Names Information System [10]. The two dimensional BPs (x_i, y_i) were first normalized to give (X_i, Y_i) where X_i and Y_i are between 0 and 1: $X_i = (X_i - X_{min})/(X_{max} - X_{min})$. Here, X_{min} is the minimium of all Xs, and X_{max} is the maximum. Similar scaling is done on the y coordinates. This linear scaling does not affect our results, but made the generation of queries somewhat simpler.

In the rest of the paper, our conclusions based on synthetic data are derived from complete experiments on 3, 10 and 16 dimensional datasets, even though some of the results are not shown. For clarity, we choose to first present experiments on point data in 3, 10 and 16 dimensions. Based on the results of these initial experiments, we go into details and present data that are only relevent to the understanding of *why* a specific organization was good (or bad).

3.2 General Performance: Penalty Metric Dominates

In order to test the performance of the different trees, we generated near-neighbor queries by choosing a point at random from an n-dimensional hypercube, and requesting the 21 nearest neighbors of that point. For each query we measured the number of leaf-level I/O requests generated by libGiST. All results shown in this paper were averaged over 1,000 queries. We chose to focus on leaf-level I/Os since, in the presence of a buffer manager, these I/Os dominate query performance time.

Figure 2 shows experimental results for all the trees of Table 2, with experiments over 16-dimensional data sets of increasing size. The most significant conclusion that arises from this experiment is that trees with a "1" in the second bit outperform the corresponding trees with that bit flipped to "2", indicating that the Penalty metric plays the most important role in determining

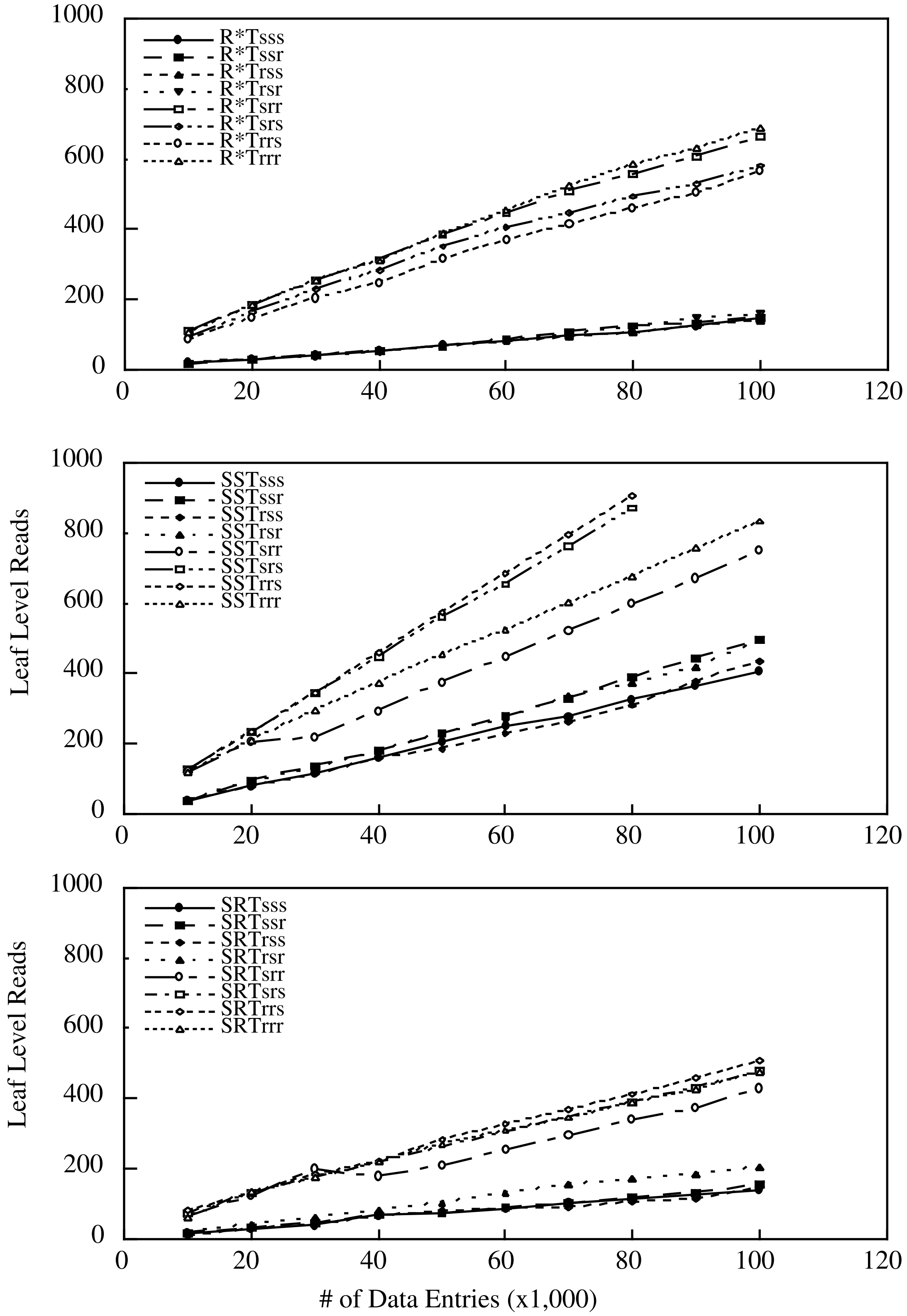


Figure 2: Summary of Leaf-level Reads on different implementations of R*-tree, SS-tree and SR-tree for 16-D data.

performance. For example, the performance of a conventional R*-tree (R*Trrr) is improved by a factor of between five and seven when it is modified to use the SS-tree Penalty metric. The same trend is also observed for the SS-tree and SR-tree. For these two trees, when the R*-tree Penalty metric is used, their performance deteriorates by a factor of 4. Compared to the Penalty metric, the PickSplit and Reinsertion methods affect the performance far less (by a factor of 30-40% at the most). More importantly, no general trend suggests that one implementation of these methods outperforms the other. For this reason, we do not present further results comparing trees that differ only on the Reinsertion and the PickSplit policies.

Although we do not present the results here, we confirmed the dominant influence of the Penalty metric for our 10-dimensional and 3-dimensional datasets. In the 3-dimensional case, the SS-tree Penalty metric does not outperform the R*-tree Penalty metric as much as before, but it still makes a significant difference (about a factor of 3).

Broadly put, these results suggest that *the conventional R*-tree does not perform well for near-neighbor queries because it does not cluster data well.* This conclusion is reached through three observations. First, both the R*Trrr and SRTrrr do not perform well, and touch at least half of the leaf level pages. Second, a close look at the leaf level nodes reveals that distant pairs of points are frequently stored in the same page. Third, and most important, when the SS-tree Penalty metric is used, R*-tree performance increases dramatically. In Section 3.4 we will see that the R*-tree provides relatively effective techniques for the near-neighbor search algorithm. Thus the inefficiency of the coventional R*-tree can be mainly attributed to its Penalty metric, which does not cluster the data well.

3.3 Search and BP Shape: Rectangles Dominate

Having demonstrated the Penalty metric as the predominant performance factor among the extension methods, we turn our attention to bounding predicates themselves, and their effect on search. In Figure 3a, we compare the performance of four implementations on 16 dimensional data: the traditional R*-tree (R*Trrr), the original SS-tree (SSTsss) and SR-tree (SRTsss), along with an R*-tree that uses SS-tree extension methods (R*Tsss).

The first observation to make is that we confirm the results of the previous work: SRTsss outperforms SSTsss, while SSTsss outperforms R*Tsss. Second and more important, we see that *R*Tsss performs much better than SSTsss.* This is not only true for 16-dimensional data, but also true for 3-dimensional data as shown in figure 3b. Note that the SR-tree has all the information of the SS-tree in its keys, and is clustered similarly (though it has a smaller non-leaf fanout). The inefficiency of the SS-tree compared to the SR-tree suggests that the spherical BPs do not provide efficient search when the dimensionality is high. In this sense, rectangular BPs seem better than spherical ones. Since the performance of SRTsss and R*Tsss are almost the same, we conclude that *for SR-trees on high dimensionality data, the rectangular portion of the key is used to direct the near-neighbor search algorithm.*

The fact that the rectangle directs the search for high dimensional SR-trees was actually observed by Katayama and Satoh. In the original SR-tree paper they explain this by arguing that spherical keys have smaller diameter, and hence may often provide tighter bounds than rectangles even though they typically have larger volume than rectangles. While it is true that the sphere often has a larger volume, its advantage of having smaller diameter may not be important. Our results demonstrate that R*Tsss is comparable to SRTsss, despite the fact that that R*Tsss uses

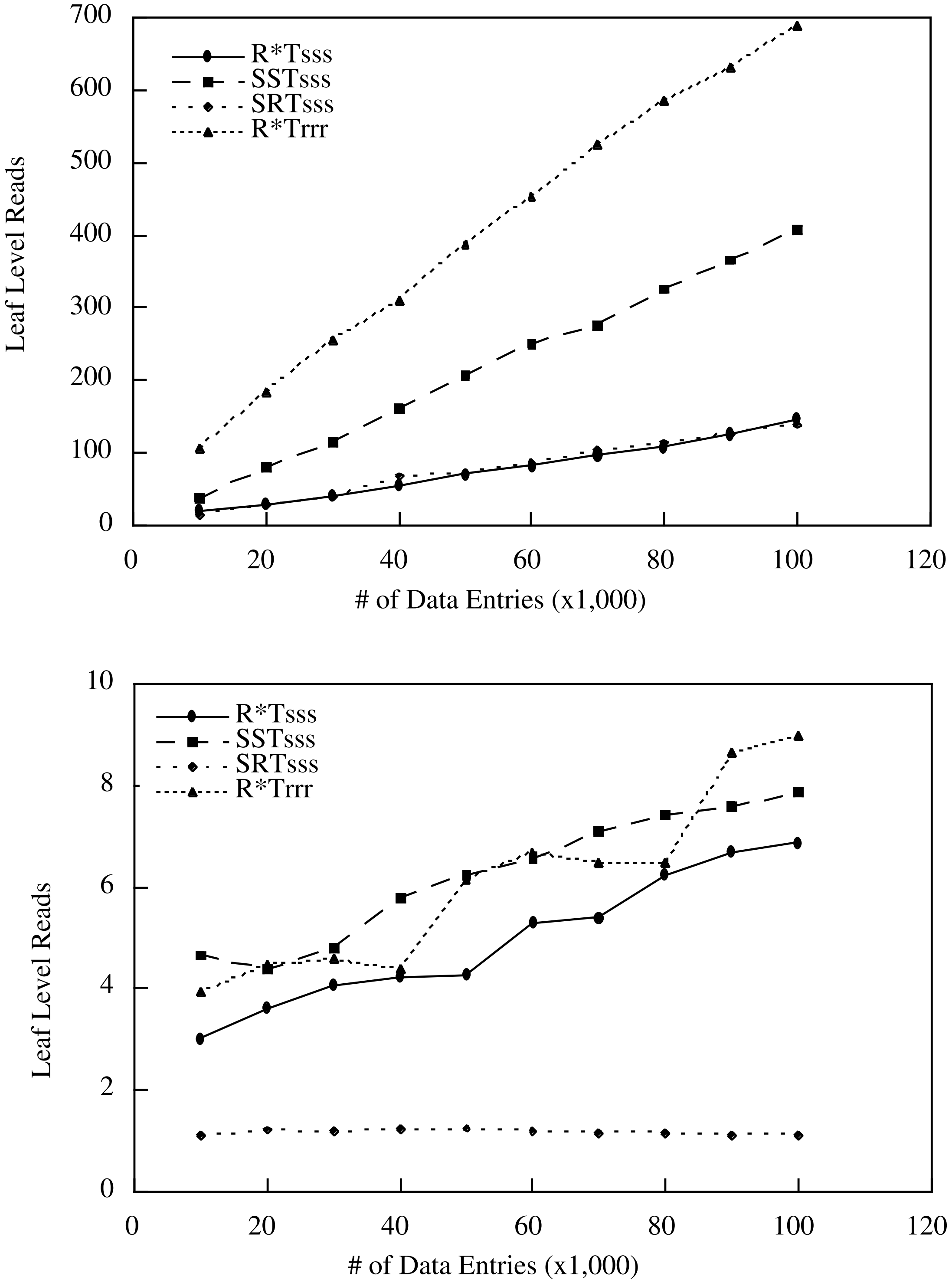


Figure 3: Comparison among R*Tsss, R*Trrr, SSTsss and SRTsss for 16D (upper) and 3D data (lower).

rectangular keys of arguably larger diameter.

We want to point out here that storage utilization difference for the four trees does not explain the performance difference. For R*Trrr, the storage utilization is around 73%, while for SSTsss, R*Tsss and SRTsss, the number is around 80%-85%. However, for some tree implementations (especially those SS-trees with R*-tree Penalty metric), the storage utilization may decrease to as low as 55%.

3.4 A Closer Look at Spheres and Rectangles

We proceed to more carefully examine the relative merits of sphere and rectangle keys. Before continuing, recall that during search, the near-neighbor algorithm computes two metrics for each BP on the node: MINDIST and MINMAXDIST [9]. They are used to order and prune the nodes to visit in a near-neighbor traversal of the tree.

We do not reiterate here the formulae for MINDIST and MINMAXDIST for rectangular and spherical BPs; the interested reader is referred to [9] and [11] for these details. We designate the MINDIST for rectangular and spherical BPs as $MINDIST_{R^*TREE}$ and $MINDIST_{SSTREE}$, respectively. Similarly, we define $MINMAXDIST_{R^*TREE}$ and $MINMAXDIST_{SSTREE}$ as the MINMAXDIST for the rectangular and spherical BPs. Since an SR-tree keeps both the rectangle and the sphere, MINDIST and MINMAXDIST are defined as

$$MINDIST_{SRTREE} = MAX(MINDIST_{SSTREE}, MINDIST_{R^*TREE}) \quad (1)$$
$$MINMAXDIST_{SRTREE} = MIN(MINMAXDIST_{SSTREE}, MINMAXDIST_{R^*TREE}) \quad (2)$$

In our experiments, we also implemented two variations of the SR-tree. The first one, designated as SRTSSsss, uses the SS-tree definition of MINDIST and MINMAXDIST. That is,

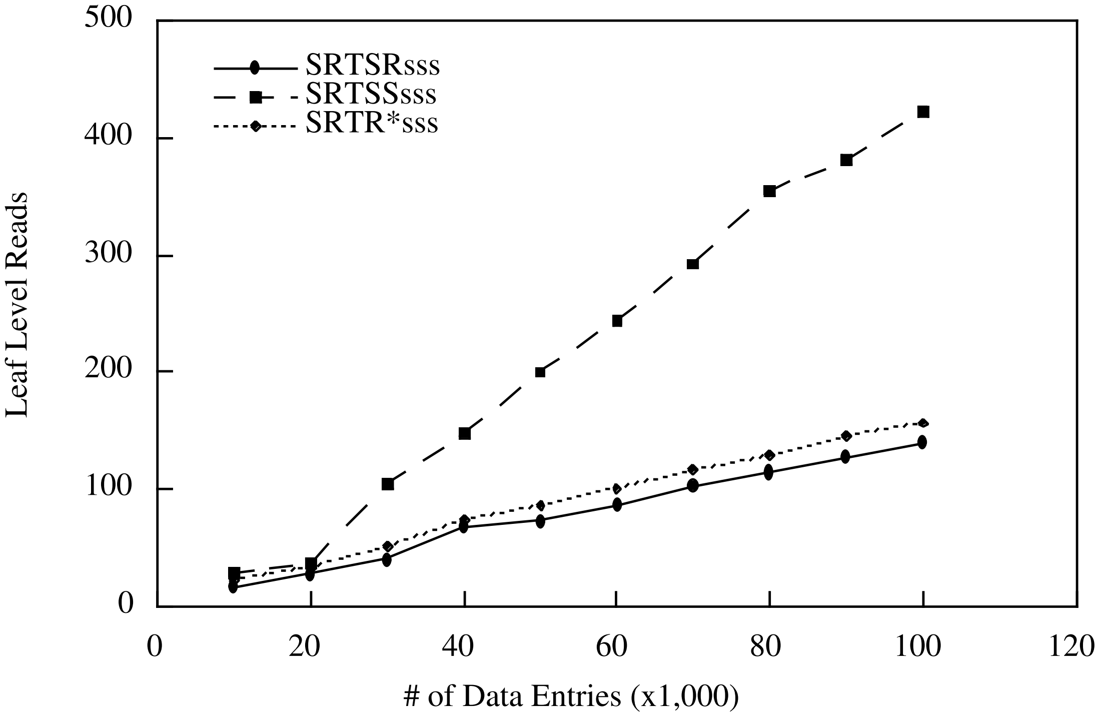
$$MINDIST_{SRTREE} = MINDIST_{SSTREE} \quad (3)$$
$$MINMAXDIST_{SRTREE} = MINMAXDIST_{SSTREE}. \quad (4)$$

Similarly, SRTR*sss uses the R*-tree definitions of MINDIST and MINMAXDIST:

$$MINDIST_{SRTREE} = MINDIST_{R^*TREE} \quad (5)$$
$$MINMAXDIST_{SRTREE} = MINMAXDIST_{R^*TREE}. \quad (6)$$

We compared the performance of SRTSRsss, SRTSSsss and SRTR*sss on our 16-dimensional test data, and the results are shown in Figure 4 (upper). Note that in this experiment the actual tree constructed in each case is identical; only the distance metrics of the search algorithm are changed. As one can see, the performance of SRTSRsss and SRTR*sss are comparable, while the SRTSSsss often touches 4 times more pages than SRTSRsss does. This clearly demonstrates that SS-tree does not perform well for high dimensional data, because it does not allow optimal estimation of many distance metrics.

We also compared the performance of SRTSRsss, SRTSSsss and SRTR*sss for three-dimensional data. As shown in Figure 4 (lower), the performance of the SS-tree metrics does exceed that of the R*-tree metrics in this case. The SR-tree-based metrics outperform both as expected, but in this case the superiority derives from their ability to use their bounding spheres. We noticed that when the dimensionality is low, spherical keys are advantageous, even though the total number of leaf-level nodes that are touched is so small that the difference between SRTSRsss and SRTSSsss does not exceed two pages.



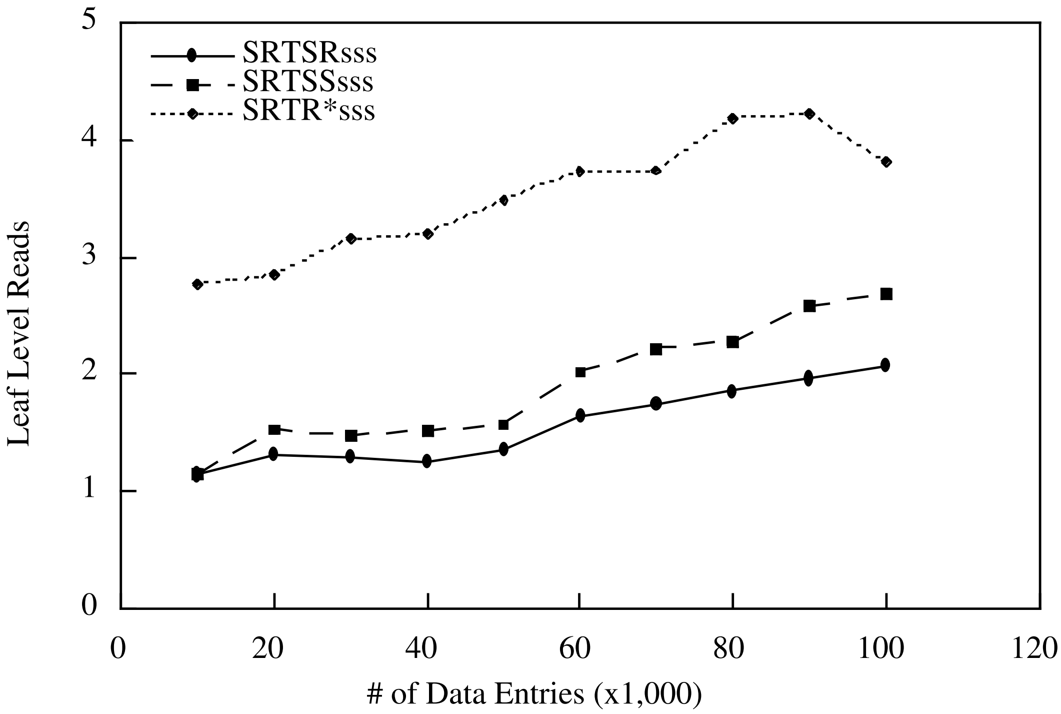


Figure 4: Performance of many Variations of SR-tree for 16D (upper) and 3D (lower) data.

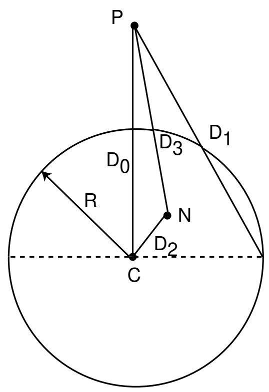


Figure 5: Diagram shows the modification to SS-tree

3.5 Improving the Sphere

We now examine the SS-tree more closely, to understand why the SS-tree cannot estimate the distance metrics effectively for high dimensional data. To do this, we try to modify the BPs for SS-tree to include more information, which can allow a better estimation of the distance metrics. One such modification, of course, is to keep a rectangle in addition to the sphere, as in the SR-tree. However this decreases the fanout for the non-leaf level of the SS-tree by a factor of three. We are more interested in modifications that improve the SS-tree performance without affecting the fanout significantly.

Our first modification to the SS-tree is to include a new field that stores the distance between the center and the point in the bounding sphere that is nearest to the center (D_2 in Figure 5). Without this distance, MINMAXDIST is calculated by

$$MINMAXDIST_1 = D_1 = \sqrt{D_0 * D_0 + R * R}, \tag{7}$$

where R is the radius of the sphere and D_0 is the distance between the search point P and the center of the sphere C. Using the new distance D_2, we have

$$\begin{aligned} MINMAXDIST_2 &= D_0 + D_2 & (8)\\ MINMAXDIST &= MIN(MINMAXDIST_1, MINMAXDIST_2). & (9)\end{aligned}$$

Notice that MINMAXDIST is a better estimation than $MINMAXDIST_1$. In our experiments, however, we found that this modification did not affect the performance noticeably.

In an attempt to improve things, we chose to store the coordinates of the point N that is nearest to the center, rather than just the distance D_2. With this point N, we have

$$\begin{aligned} MINMAXDIST_2 &= D_3 & (10)\\ MINMAXDIST &= MIN(MINMAXDIST_1, MINMAXDIST_2), & (11)\end{aligned}$$

where D_3 is the distance between P and N. In Figure 6, we compare the performance of the standard SS-tree with an SS-tree whose BPs also contain this new field.

As one can see, with the new field SS-tree actually performs worse. This is not surprising if one considers that this modification decreases the non-leaf level fanout by a factor of two. If this effect is compensated by doubling the page size at the nonleaf levels, the performance of the modified

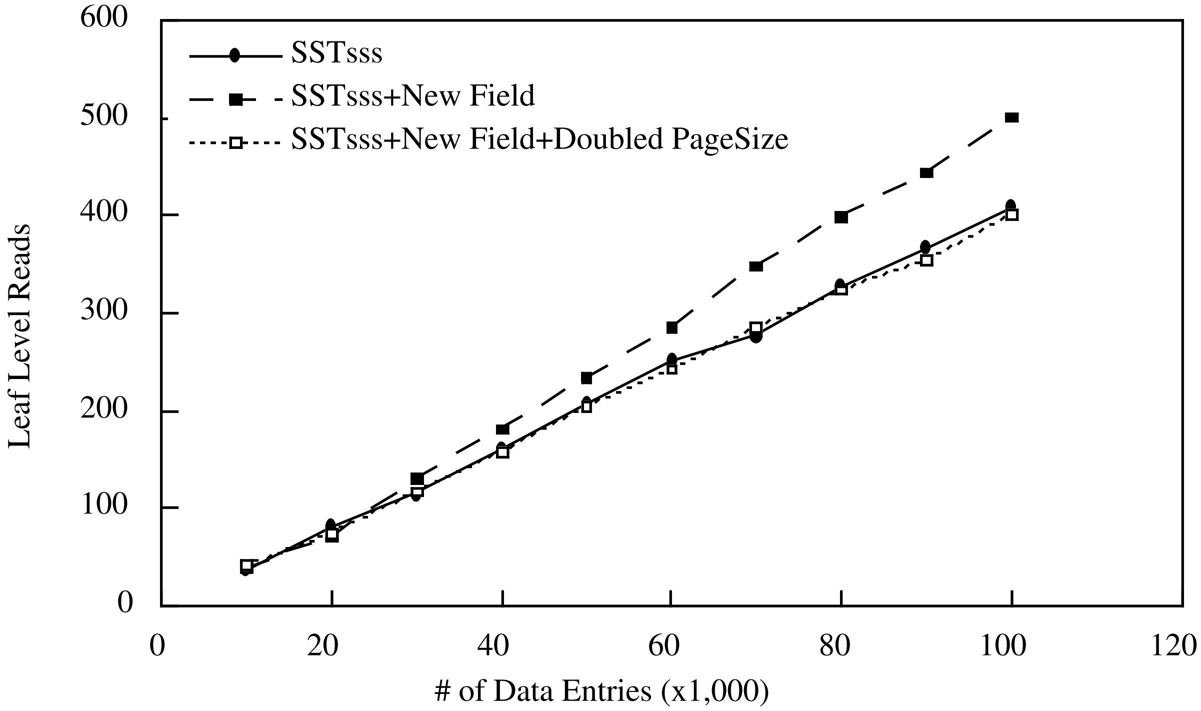


Figure 6: Performance of the Modified SS-tree for 16-D data.

SS-tree actually almost matches that of the SS-tree. This suggests that our second modification does not improve the performance of SS-tree either.

One should notice that while the above modifications allow a better estimation of MINMAXDIST, they do not improve the estimation of MINDIST. This inefficiency of the modifications suggests that *the SS-tree does not perform well because it gives an inaccurate estimation of MINDIST.* The R*-tree and SR-tree, which save a rectangle, allow a better estimation of MINDIST, and outperforms the SS-tree for high-dimensional dataset.

From these experiments, we can come up with a relatively clear picture of why SS-trees do not work well for high dimensional data. When visiting children of a node during a near-neighbor traversal, the child of smallest MINDIST is visited first. However, since SS-trees do not make a good estimation of MINDIST, the child with the smallest MINDIST often does not contain many points that are nearest to the search point.

3.6 The Effects of Fanout

A close look at Figure 6 suggests that page size does not affect the performance of the SS-tree too much. To verify this observation, we reran the 16D test for R*Tsss, SSTsss and SRTsss, halving their pagesize to 4096 bytes. The results are summarized in Figure 7. As one can see, when the page size is halved, the number of leaf level page accesses increases. However, the increase is only about 30% at most. Compared to the 3- to 7-fold differences in performance due to the Penalty metric, we expect any effects due to a 2x difference in fanout to essentially be noise.

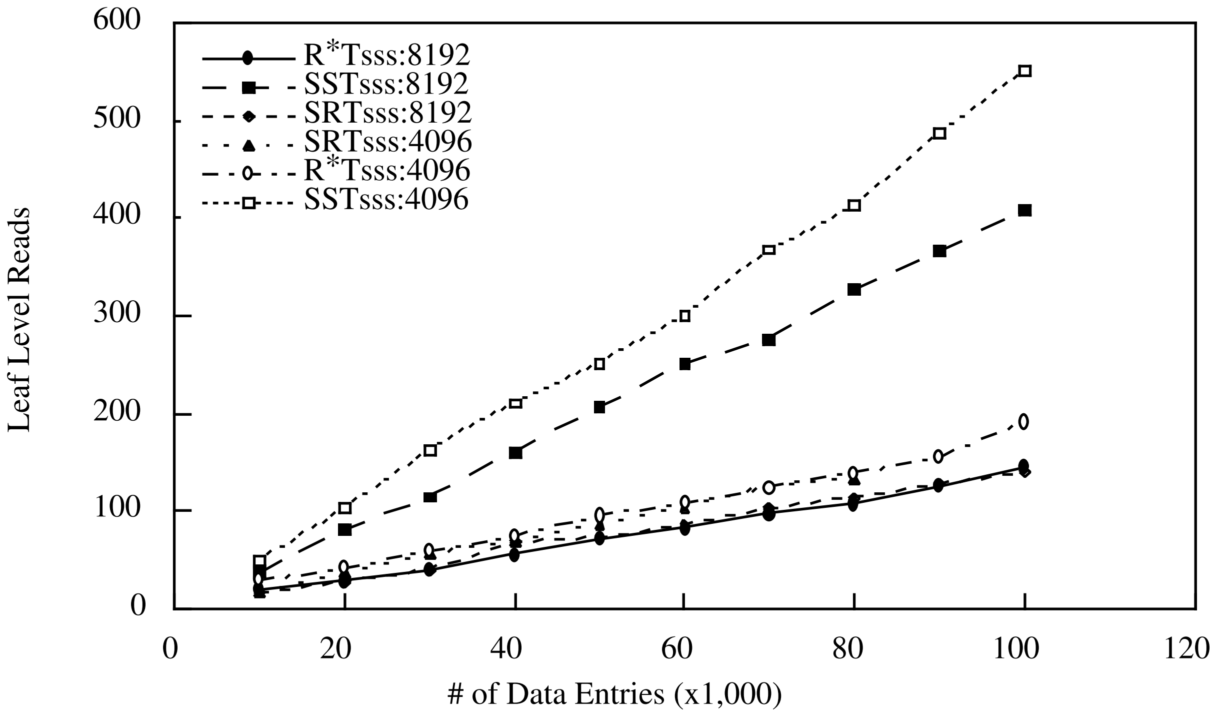


Figure 7: The dependence of the performance on pagesize for 16-D data.

3.7 Performance on Real Data

Finally, we checked our low-dimensional results by experimenting on real two-dimensional data, representing the locations of places in the United States. As shown in Figure 8, the performance of all three trees is good. This is expected for the low dimensionality of map data (2). The performance of SR-tree is the best, as expected. Compared to the synthetic 3-D data, two issues are worth noting. First, the performance difference between R*-tree and SR-tree is much less dramatic for the real data. This suggests that R*-trees may serve as good approximations of the superior SR-trees in realistic low-dimensional data sets. SS-trees remain ineffective: the number of leaf level pages that are touched by the SS-tree is about two times that of the R*-tree, and 3 times that of the SR-tree.

4 Conclusions and Future Work

We have performed a detailed study of the techniques proposed for near-neighbor search in the R*-tree, SS-tree, and SR-tree. Our analysis shows that the most important difference between these trees is that the Penalty metric used in the SS-tree and SR-tree is superior to that of the R*-tree, resulting in far better clustering for near-neighbor search. The different Reinsertion and PickSplit methods proposed for these trees do not seem to make much relative difference in performance.

With regarding to the bounding predicates, we find that rectangular bounding predicates are superior to spherical ones in high dimensions, since rectangular BPs produce smaller estimations of the MINDIST metric in the near-neighbor search algorithm.

Combining these results, we observe that when the R*-tree is modified to use the SS-tree Penalty

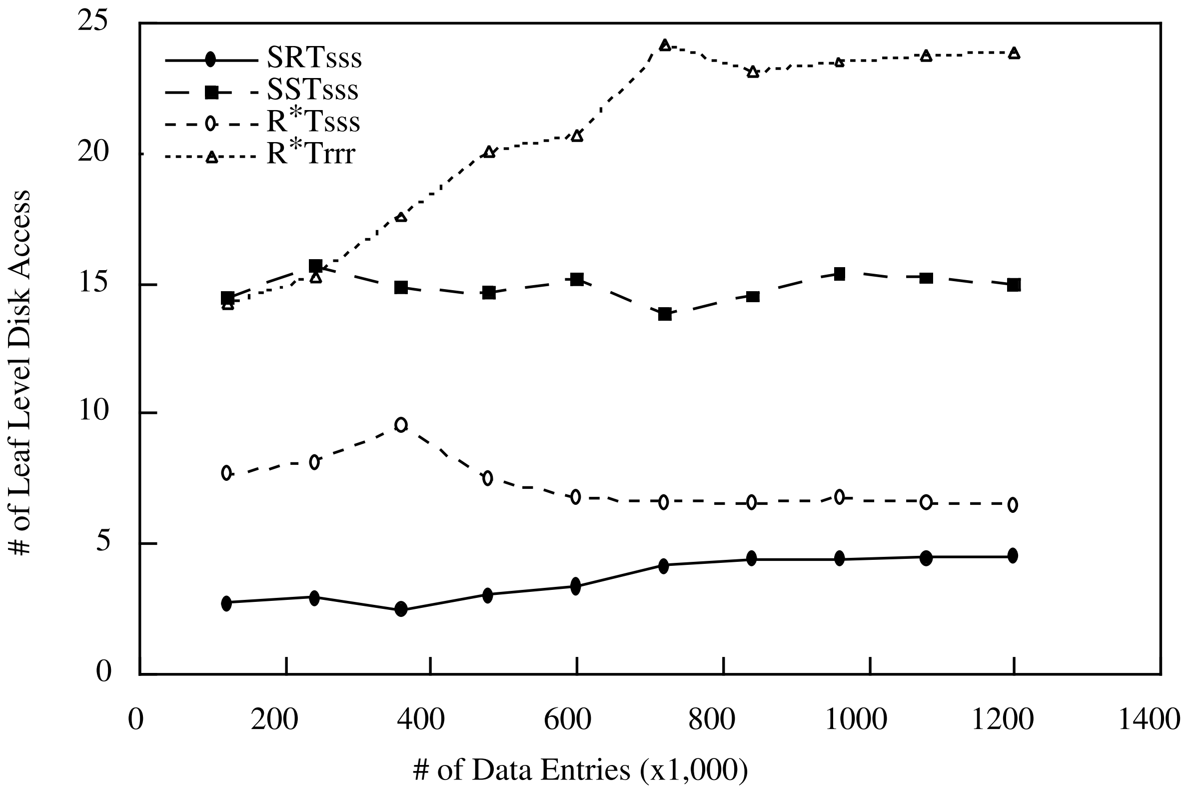


Figure 8: Performance comparison on 2-D geographical data.

metric, it actually outperforms the SS-tree in most of our experiments on sythetic and real data. The SR-tree outperforms all trees, but its performance for high dimensional data is not significantly superior to that of the modified R*-tree. For low-dimensional data, where the search often involves only a few page accesses, the SR-tree does outperform the R*-tree, though the absolute difference per lookup is only a couple I/Os. In short, we believe that R*-trees with a modified Penalty metric are an attractive choice for near-neighbor search in spaces of more than a few dimensions; this is particulary true since R* are more standard for range-search and have already been implemented in a variety of contexts.

A subsidiary result of this work is to illustrate the advantage of GiSTs not only for ease of implementation, but also for clarity in the analysis of design tradeoffs. The GiST framework exposes just those parameters which vary across trees, and hence motivates access method designers to examine those parameters closely. Although our study suggests that modified R*-trees are an attractive choice for near-neighbor search, we very consciously chose not to give these "new" trees a name. It is our belief that most multi-dimensional search trees are extremely similar, and that the interesting challenge in designing an efficient search tree is not in inventing new techniques, but in justifying the particular advantages of the techniques.

A few open problems remain in this body of work. First, we would like to extend our results to range queries. We believe that the Penalty metric of the SS-tree should be advantageous for such workloads, since it seems to cluster proximate points together. We are also interested in focusing closely on the data stored in BPs to support near-neighbor search; we believe that it may be possible to improve upon the combined rectangle/sphere BP of the SR-tree. Finally, there exist near-neighbor search algorithms [6] that are more efficient than that of [9], and we would like to revisit our results in that context.

References

[1] Stefan Berchtold, Christian Bohm, Daniel A Keim, Hans-Peter Kriegel. A Cost Model for Nearest Neighbor Search in High-Dimensional Space. In *Proc. 16th ACM SIGACT-SIGMOD SIGART Symposium on Principles of Database Systems (PODS)*, pages 78–86, Tucson, May 1997.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, May 1990.

[3] The GiST Project. Gist Home Page. http://gist.cs.berkeley.edu.

[4] Antonin Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.

[5] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[6] G. R. Hjaltason and H. Samet. Ranking in Spatial Databases. In *Proc. 4th International Symposium on Large Spatial Databases*, pages 83–95, Portland, ME, 1995.

[7] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997, pages 62–72.

[8] Norio Katayama and Shin'ichi Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997, pages 369–380.

[9] Nick Roussopoulos, Stephen Kelley, and Frederic Vincent. Nearest Neighbor Queries. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, May 1995.

[10] U.S. Geological Survey. Geographic Names Information System, National Mapping Program, 1987.

[11] David A. White and Ramesh Jain. Similarity Indexing with the SS-tree. In *IEEE International Conference on Data Engineering*, pages 516–523, 1996.