STR: A Simple and Efficient Algorithm for R-Tree Packing [*]

Scott T. Leutenegger Jeffrey M. Edgington Mario A. Lopez
Mathematics and Computer Science Department
University of Denver Denver, CO 80208-0189
{*leut,jedgingt,mlopez*}@*cs.du.edu*

Abstract

In this paper we present the results from an extensive comparison study of three R-tree packing algorithms, including a new easy to implement algorithm. The algorithms are evaluated using both synthetic and actual data from various application domains including VLSI design, GIS (tiger), and computational fluid dynamics. Our studies also consider the impact that various degrees of buffering have on query performance. Experimental results indicate that none of the algorithms is best for all types of data. In general, our new algorithm requires up to 50% fewer disk accesses than the best previously proposed algorithm for point and region queries on uniformly distributed or mildly skewed point and region data, and approximately the same for highly skewed point and region data.

[*]The work of Leutenegger and Lopez was supported in part by Colorado Advance Software Institute grant number TT-97-05. The work of Leutenegger was additionally supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while visiting the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681-0001

1 Introduction

R-trees [5] are a common indexing technique for spatial data and are widely used in spatial and multi-dimensional databases. By storing the bounding boxes of arbitrary geometric objects, such as points, polygons, or more complex objects, R-trees can be used to determine which objects intersect a given query region. Typical applications include computer-aided design, geographic information systems, computer vision and robotics, multi-keyed indexing for traditional databases, temporal and scientific databases. R-trees are dynamic structures, in the sense that their contents can be modified without reconstructing the entire tree, and Guttman [5] provides efficient routines for insertion and deletion of objects.

Unfortunately, building an R-tree by inserting one object at a time as specified by Guttman has several disadvantages: (a) high load time, (b) sub-optimal space utilization, and, most important, (c) poor R-tree structure requiring the retrieval of an unduly large number of nodes in order to satisfy a query. Other dynamic algorithms [1, 13] improve the quality of the R-tree, but still are not competitive with regard to query time when compared to loading algorithms that are allowed to preprocess the data to be stored. Preprocessing is particularly reasonable for applications where the data is fairly static (i.e., does not change often) or available *a priori* and, when done properly, results in R-trees with nearly 100% space utilization and improved query times (due to the fact that fewer nodes need to be accessed while performing a query). Such *packing algorithms* were first proposed by Roussopoulos [12] and later by Kamel and Faloutsos [6].

Kamel and Faloutsos [6] propose a packing algorithm based on the Hilbert Curve ordering, and compare it with the Nearest-X algorithm proposed by Roussopoulos [12]. The latter is simpler to implement and in some cases results in better trees for point queries. However, due to the smaller perimeter of the bounding rectangles at internal nodes, the algorithm of [6] significantly outperforms that of [12] for region queries. Consequently, the Hilbert-based packing algorithm is usually the preferred choice for region queries while remaining competitive for point queries.

In this paper we propose a new packing algorithm (Sort-Tile-Recursive) that is simple to implement and compare it with the Hilbert and Nearest-X packing algorithms for a wide range of data and buffer sizes. In addition to area and perimeter metrics, we provide experimental evidence based on real implementations utilizing an LRU buffer on VLSI design, GIS, computational fluid dynamics, and synthetic data sets. We know of no other work that has considered such a wide range of data set

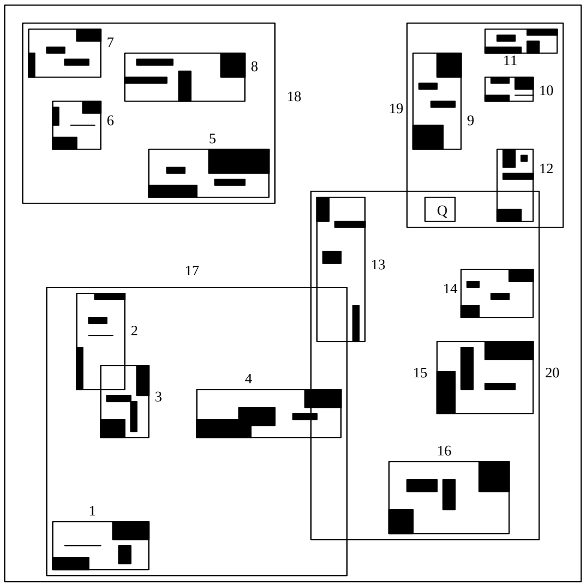

Figure 1: A sample R-tree. Input rectangles are shown solid.

types and the effect they have on packing performance. In real databases some portion of the tree is buffered in main memory. This buffering of portions of the tree can significantly affect performance as shown in [8]. Consequently, our experimental studies utilize a buffer as described in Section 3.

The rest of the paper is organized as follows. In Section 2 we provide background information on R-trees and describe the three packing algorithms considered in this paper. In Section 3 we present our experimental methodology. Section 4 contains results from our experiments and Section 5 concludes.

2 Overview of R-tree and Packing Algorithms

In this section we provide a brief overview of the R-tree and describe several packing algorithms for R-trees, including Nearest-X [12], Hilbert [6], as well as Sort-Tile-Recursive (STR), a new packing algorithm which we are proposing. Detailed knowledge of Nearest-X and Hilbert packing is useful but not required for understanding the remainder of this paper. Readers interested in more detailed descriptions should refer to [12, 6].

2.1 R-trees

An R-tree is a hierarchical data structure derived from the B-tree and designed for efficient execution of intersection queries. R-trees store a collection of rectangles which can change over time through insertions and deletions. Arbitrary geometric objects are handled by representing each object by its *minimum bounding rectangle*, i.e., the smallest upright rectangle which encloses the object. R-trees generalize easily to dimensions higher than two, but for notational simplicity we review only the two dimensional case.

Each node of the R-tree stores a maximum of n entries. Each entry consists of a rectangle R and a pointer P. For nodes at the leaf level, R is the bounding box of an actual object pointed to by P. At internal nodes, R is the minimum bounding rectangle (MBR) of all rectangles stored in the subtree pointed to by P. Note that every path down through the tree corresponds to a sequence of nested rectangles, the last of which contains an actual data object. Note also that rectangles at any level may overlap and that an R-tree created from a particular set of objects is by no means unique.

Figure 1 illustrates a 3-level R-tree where a maximum of 4 rectangles fit per node. We assume that the levels are numbered 0 (root), 1, and 2 (leaf level). There are 64 rectangles represented by the small dark boxes. The 64 rectangles are grouped into 16 leaf level nodes, numbered 1 to 16. The MBR enclosing each leaf node is the smallest box that fully contains the rectangles within the node. The MBRs of the leaf nodes are the rectangles stored in the nodes at the next higher level of the tree.

For example, leaf nodes 1 through 4 are placed in node 17 which is at level 1. The MBR of node 17 (and nodes 18,19,20) is purposely drawn slightly larger than needed for clarity. The root node contains the four level 1 nodes: 17, 18, 19, and 20.

To perform a query Q, all rectangles that intersect the query region must be retrieved and examined (regardless of whether they are stored in an internal node or a leaf node). This retrieval is accomplished by using a simple recursive procedure that starts at the root node and which may follow several paths down through the tree. A node is processed by first retrieving all rectangles stored at that node which intersect Q. If the node is an internal node, the subtrees corresponding to the retrieved rectangles are searched recursively. Otherwise, the node is a leaf node and the retrieved rectangles (or the data objects themselves) are simply reported.

For illustration, consider the query Q in the example of Figure 1. After examining the root node, we determine that nodes 19 and 20 of level 1 must be searched. The search then proceeds to each of these nodes. It is then determined that the query region does not intersect any rectangles stored in node 19 or node 20 and each of these two subqueries are terminated.

The R-tree shown in Figure 1 is fairly well structured. Inserting these same rectangles into an R-tree using the insertion algorithms of Guttman [5] would likely result in a less well structured tree. Algorithms to create well structured trees have been developed and are described in Section 2.2. These algorithms cluster rectangles in an attempt to minimize the number of nodes visited while processing a query.

For the rest of the paper we will assume that exactly one node fits per disk page, and hereafter we use the two terms interchangeably.

2.2 Packing Algorithms

In this section we describe three packing algorithms. All of the algorithms use a similar framework. In the following text we assume that the data file consists of r rectangles and that each R-Tree node can hold n rectangles.

The general process is similar to building a B-tree from a collection of keys by creating the leaf level first and then creating each successively higher level until the root node is created [11].

General Algorithm:

1. Preprocess the data file so that the r rectangles are ordered in $\lceil r/n \rceil$ consecutive groups of n rectangles, where each group of n is intended to be placed in the same leaf level node. Note that the last group may contain fewer than n rectangles.

2. Load the $\lceil r/n \rceil$ groups of rectangles into pages and output the (MBR, page-number) for each leaf level page into a temporary file. The page-numbers are used as the child pointers in the nodes of the next higher level.

3. Recursively pack these MBRs into nodes at the next level, proceeding upwards, until the root node is created.

The three algorithms differ only in how the rectangles are ordered at each level.

Nearest-X (NX):

This algorithm was proposed in [12]. The rectangles are sorted by x-coordinate. No details are given in the paper so we assume that the x-coordinate of the rectangle's center is used. The rectangles are then packed into the nodes, in groups of size n, using this ordering.

Hilbert Sort (HS):

A fractal based algorithm was proposed in [6]. The algorithm orders the rectangles using the Hilbert (fractal) space filling curve. The center points of the rectangles are sorted based on their distance from the origin, measured along the Hilbert Curve. This determines the order in which the rectangles are placed into the nodes of the R-Tree.

Kamel and Faloutsos only provide details on how to handle integer coordinates, which can be extended to arbitrary floating point values as described below.

Floating point numbers are usually stored as a sign, signed exponent, and a mantissa (which may or may not be normalized). Since the exponent determines the starting position of the mantissa relative to the binary point, all floating point numbers could be represented using $2^{\text{sizeof}(Exponent)} + \text{sizeof}(Mantissa)$ bits. For example, for 32-bit float numbers native in the Sun Sparc architecture, $2^8 + 23$ bits would required. (This is a conceptual representation only. Coordinates are not stored in this form.) Once the numbers are viewed using this representation, it is clear that the method used for integers can be applied.

We now briefly describe the processing of a 2-dimensional data set. Consider a grid of size $2^{2^{\text{sizeof}(Exponent)}+\text{sizeof}(Mantissa)}$. The Hilbert Curve for this grid is used to produce the packing order of the rectangles. The first bit of the x- and y-coordinates of a point determine which quadrant contains it. Successive bits determine which successively smaller subquadrants contain the point. When two center points (x_1, y_1) and (x_2, y_2) need to be compared, the bits of each coordinate are examined until it can be determined that one of the points lies in a different subquadrant than the other (one can use the sense and rotation tables described in [6] to accomplish this task). The information gathered is used to decide which point is closer to the origin (along the Hilbert Curve). Conceptually, the process computes bit positions, one at a time, until discrimination is possible. In practice, one does not store or compute all bit values on the hypothetical grid.

Sort-Tile-Recursive (STR):

Consider a k-dimensional data set of r hyper-rectangles. A hyper-rectangle is defined by k intervals of the form $[A_i, B_i]$ and is the locus of points whose i-th coordinate falls inside the i-th interval, for all $1 \leq i \leq k$.

STR is best described recursively with $k = 2$ providing the base case. (The case $k = 1$ is already handled well by regular B-trees.) Accordingly, we first consider a set of rectangles in the plane. The basic idea is to "tile" the data space using $\sqrt{r/n}$ vertical slices so that each slice contains enough rectangles to pack roughly $\sqrt{r/n}$ nodes. Once again we assume coordinates are for the center points of the rectangles. Determine the number of leaf level pages $P = \lceil r/n \rceil$ and let $S = \lceil \sqrt{P} \rceil$. Sort the rectangles by x-coordinate and partition them into S vertical slices. A slice consists of a run of $S \cdot n$ consecutive rectangles from the sorted list. Note that the last slice may contain fewer than $S \cdot n$ rectangles. Now sort the rectangles of each slice by y-coordinate and pack them into nodes by grouping them into runs of length n (the first n rectangles into the first node, the next n into the second node, and so on).

The case $k > 2$ is is a simple generalization of the approach described above. First, sort the hyper-rectangles according to the first coordinate of their center. Then divide the input set into $S = \lceil P^{\frac{1}{k}} \rceil$ slabs, where a slab consists of a run of $n \cdot \lceil P^{\frac{k-1}{k}} \rceil$ consecutive hyper-rectangles from the sorted list. Each slab is now processed recursively using the remaining $k - 1$ coordinates (i.e., treated as a $k - 1$-dimensional data set).

To aid in visualizing the result of these packing algorithms, consider the leaf level nodes obtained by using the Long Beach Tiger data set assuming 100 rectangles fit per node. Figures 2, 3, and 4 show the resultant leaf level MBR for the same data set for each of the three algorithms. Note the vertical slices in Figure 4 for the STR packing algorithm.

3 Experimental Methodology

In this section we describe our experimental methodology. Our goal is to provide a solid experimental comparison of the algorithms through actual R-tree implementations over a wide range of data using both "real world" and synthetic data sets. We intend to provide insight into how well R-trees would perform as part of a typical database system which supports spatial queries. Thus, in order to provide realistic and meaningful performance measurements, the effect of buffering must be taken into consideration.

Our primary comparison metric is the number of disk accesses required to satisfy a query of a given size. Note that this metric also allows us to get an accurate indication of performance even using non-dedicated workstations. If we had focused on retrieval time, interference from other users would have clouded our results.

We assume an LRU buffer management routine. A slightly better buffer management routine may arguably be to pin the root and some number of the first few R-tree levels and then use an LRU scheme for the remaining nodes of the R-tree. As shown in [8] there is often no gain from this pinning, except in unusual circumstances where a level near the root just fits into the buffer pool, in which case it should be pinned. We use LRU for all the nodes (regardless of their level) to simplify the parameter space of our experiments.

To accurately assess the impact of buffer size, we implement our buffer manager using a raw disk partition. When a node is pushed out of the buffer the node is immediately written to disk and not "false-buffered" by the operating system's virtual memory manager. Thus, we can easily vary the actual buffer size without reconfiguring the OS or hardware.

Many of our data sets are larger than those used in previous studies, yet they are still smaller than data sets likely to be used by near term future applications. Data set size affects R-tree performance in two ways: 1) it increases the depth of the R-tree, and 2) it decreases the percentage of the data set that will fit in the buffer. Since most R-trees have a fan out of 25 to 100 the first consideration is not as significant as the second. Thus, one of the experimental parameters of interest is the percentage of the data set that can be buffered. Since several of our experiments use small data sets (approximately 50,000 rectangles), we consider small buffer sizes. This allows us to obtain the same type of results as would be obtained using larger data sets and buffers, at a great savings in experimental time. Even with these smaller data sets our experiments took two months utilizing four Sun Sparc 5 workstations.

For each of our experiments, we build the R-tree according to the specific packing algorithm being considered. In each experiment the exact same data set is used for all algorithms. We then query the data set with 2,000 queries. No attempt at collecting confidence intervals or smoothing curves is made. Thus, to err on the side of caution we advise that differences of less than a few percent should not be considered significant.

To provide a uniform experiment space we normalize all data sets to the unit square. Point queries are uniformly distributed in the unit square. We consider region queries whose region equals 1% and 9% of the unit square. The lower left hand corner is uniformly distributed in the unit square. The upper right hand corner is computed by adding ϵ to the x- and y-coordinates where $\epsilon = 0.1$ or 0.3 for region queries of 1% and 9% respectively. If the x- or y-coordinate is larger than 1.0 we set the coordinate to 1.0. For uniformly distributed data a region query of 9% will return roughly 9% of the data, but for highly skewed data (as in the VLSI and CFD data sets described below), the variance on amount of output is large and a query that covers 9% of the unit square may return much more or much less than 9% of the data.

Our secondary comparison metric is the sum of the area and perimeter of the MBRs of the R-tree nodes. These measures are good indicators of the number of nodes accessed by a query [6] but can be misleading if buffering is not considered [8]. We include these measures as additional information and present area and perimeter metrics for both the whole tree (summed over all nodes at all levels) and also only for the leaf level. We argue that the leaf level metric is of most interest since the non-leaf level nodes will likely be buffered.

The applicability of our conclusion in a general setting is dependent on how representative our data sets are. This is a non-trivial question. We address this issue by considering many different types of real data as well as synthetic data. In particular we consider the following data sets:

1. (GIS): As a data set representative of geographic information systems we chose the Long Beach data of the TIGER system of the U. S. Bureau of Census. This data set contains 53,145 line segments and has been used extensively in past studies.

2. (VLSI): We consider a CIF data set of 453,994 rectangles provided by Bell Labs and used in the design of a chip [9]. This data is interesting because the input rectangle distribution is highly skewed, both in location and in size. For example, the largest rectangle is roughly 40,000 times larger than the smallest one. Similarly, there are regions of the chip covered by several thousand rectangles and some covered by no rectangles at all.

3. (CFD): One of the primary motivations of this work is to apply the techniques to scientific data sets obtained from Computational Fluid Dynamics. We consider a 2-dimensional problem. A system of equations is used to model the air flows over and around aero-space vehicles [10]. The data sets are for a cross section of a Boeing 737 wing with flaps out in landing configuration at

MACH 0.2. The data space consists of a collection of points (nodes) of varying density. Nodes are dense in areas of great change in the solution of the equations and sparse in areas of little change. To help the reader understand the nature of the data we include a plot of a data set with 5088 nodes (see Figure 5). The experimental results use a data set with 52,510 nodes, which is similar but looks like a black smudge when plotted due to the density of points. Note that the black region in the middle of Figure 5 accounts for the majority of the data. In Figure 6 we plot only the area around the centroid of the data set. The blank oval-ish areas are parts of the wing. It is evident that the data set is highly skewed.

These CFD data sets and two other (smaller and larger) plus the tiger data sets will soon be available from `http://www.cs.du.edu/~leut/MultiDimensionalData.html`.

4. (Synthetic): Uniformly distributed data sets were created containing between 10,000 and 300,000 squares. All squares are fully contained in the unit square. For each square the lower left corner was uniformly distributed over the unit square. The area of the square is uniformly distributed between 0 and 2 times the average area. The value of the average area of a square is determined by the *density* [6] of the data set, where density equals the sum of the areas of all the squares in the data set. Specifically, let r equal the number of squares in the data set and d equal the density. Then, the average area of a square equals $\frac{d}{r}$. For each square, the actual area is chosen uniformly between 0 and two times the average area. The upper right corner is chosen to give the desired area unless it exceeds the bounds of the unit square, in which case the coordinate(s) that exceeds 1.0 is set to 1.0. We considered data densities of 0 (point data), 1.0, 2.5, and 5.0. We present results for densities of 0 and 5.0.

4 Results

In this section we present the results of our experimental methodology. We present results for point and region queries on 2-D synthetic, GIS (tiger), VLSI, and CFD data sets. All results are obtained from R-trees with 100 rectangles per node, with a range of buffers sizes examined. The curves for the NX algorithm are not included in the figures since the NX algorithm is not competitive, requiring 2 - 8 times as many disk accesses as the STR algorithm for all experiments except point queries on point data. To be complete we do include NX results in the tables of this section.

Data Size	R-Tree Pages	Buffer = 10	Buffer = 250
10,000	101	9.90%	100%
25,000	254	3.94%	98.43%
50,000	506	1.98%	49.41%
100,000	1011	0.99%	24.73%
300,000	3031	0.33%	8.25%

Table 1: Percent of R-Tree Held By Buffer

4.1 Synthetic Data

We first consider synthetic data. We consider buffer sizes of 10 and 250 pages. In Table 1 we show what percent of the R-tree fits in the buffer for the different sizes of data sets. The first column is the number of rectangles, the second is the number of R-tree pages (including non-leaf) assuming 100 rectangles per page, the third is the percent of the R-tree a buffer of 10 pages can hold, and the fourth is the percent a buffer of 250 pages can hold. We do not consider a data size of 10,000 for a buffer of 250 pages as the entire R-tree fits in the buffer.

Figures 7 and 8 plot the number of disk accesses versus data set size (in thousands of rectangles) for point queries using a buffer size of 10 and 250 pages respectively. The top two curves in each figure are for a data density of 5 (i.e., the expected sum of the areas of the input rectangles equals 5) and the bottom two curves are for a density of 0 (i.e., point data). The legends in the figures show the ordering of the lines from top to bottom. The solid lines are for STR and the dashed lines are for HS. For a buffer of size 10, HS requires 31 - 42% more disk accesses than STR for point data, and 26 - 32% more disk access for region data of density 5. For a buffer of size 250, HS requires 33 - 41% more disk access than STR for point data, and 26 - 32% more disk access for region data of density 5. Note, that for the 25,000 rectangle data set almost the entire tree fits and hence the comparison is not particularly meaningful.

In Figure 9 we plot the number of disk accesses versus the data set size (in thousands of rectangles) for region queries of 1% of the data space using a buffer of 10 pages. The plot for a buffer of 250 pages is similar. For point data, the bottom two curves, HS requires 6 - 22% more disk access than STR. For region data of density 5, the top two curves, HS requires 6 - 16% more disk access than STR. Note that as the query region size increases, the difference between STR and HS becomes smaller (but STR always requires fewer disk accesses). This result is not surprising since the more data that needs to be retrieved the more naive the search algorithm can afford to be [3].

	Point Data					Region Data, Density = 5.0				
Data Size	STR	HS	NX	HS/STR	NX/STR	STR	HS	NX	HS/STR	NX/STR
Point Queries										
10	0.89	1.26	0.87	1.42	0.99	1.40	1.85	3.52	1.32	2.51
25	1.03	1.41	1.04	1.38	1.01	1.67	2.19	6.11	1.31	3.67
50	1.27	1.74	1.27	1.37	1.00	1.97	2.57	8.43	1.30	4.28
100	1.61	2.18	1.57	1.35	0.97	2.31	2.99	12.45	1.29	5.39
300	1.95	2.55	1.83	1.31	0.94	2.60	3.27	19.70	1.26	7.56
Region Queries, Query Region = 1% of Data										
10	3.27	3.99	10.86	1.22	3.33	4.25	4.97	13.78	1.17	3.24
25	6.85	8.00	26.61	1.17	3.89	8.53	9.87	31.44	1.16	3.69
50	11.48	12.81	50.64	1.12	4.41	13.12	14.55	57.52	1.11	4.38
100	18.21	19.93	98.47	1.09	5.41	20.40	22.14	108.37	1.09	5.31
300	41.46	44.02	290.05	1.06	7.00	44.73	47.26	307.38	1.06	6.87
Region Queries, Query Region = 9% of Data										
10	11.73	13.02	26.86	1.11	2.29	13.57	14.80	29.91	1.09	2.20
25	26.40	28.07	67.26	1.06	2.55	29.01	30.76	72.17	1.06	2.49
50	46.20	48.74	131.96	1.05	2.86	49.48	51.97	48.74	1.05	0.98
100	84.54	87.51	261.35	1.04	3.09	89.25	92.18	271.58	1.03	3.04
300	229.75	234.82	779.96	1.02	3.39	237.42	242.41	797.94	1.02	3.36

Table 2: Number of Disk Accesses, Synthetic Data, Buffersize = 10

Carrying this argument to an extreme, if the query regions encloses all input data then no search needs to be performed: all Rtree packing schemes exhibit the same performance as all leaves need to be examined.

More exhaustive results are presented in Tables 2 and 3 for buffer sizes of 10 and 250 pages, respectively. The first column is the number of data items in thousands, the second through fourth columns are the number of disk accesses to satisfy the query for STR, HS, and NX on point data, the fifth and sixth columns are the ratio of HS and NX relative to STR for point data, and columns 7-11 are the same as 2-6 but for region data of density 5. Note that NX is not competitive except for point queries on point data, and, as expected, the difference between STR and HS diminishes as the query size increases.

We present area and perimeter information for the 50K and 300K data sets in Table 4. We include the sum of both area and perimeter for both the MBRs at leaf level and all MBRs in the tree. The second through fourth columns are for the 50K data set for STR, HS, and NX respectively, while the fifth through seventh column are for the 300K data set. The STR algorithm produces a smaller area and perimeter than the HS algorithm for both data sets, whereas the NX algorithm produces a slightly smaller total area for the point data but the same leaf level area. Note that

	Point Data					Region Data, Density = 5.0				
Data Size	STR	HS	NX	HS/STR	NX/STR	STR	HS	NX	HS/STR	NX/STR
Point Queries										
25	0.13	0.14	0.13	1.03	1.00	0.14	0.14	0.20	1.05	1.43
50	0.52	0.69	0.52	1.33	1.01	0.79	1.00	3.85	1.27	4.88
100	0.74	1.04	0.77	1.41	1.04	1.16	1.53	8.05	1.32	6.95
300	0.91	1.23	0.92	1.35	1.01	1.45	1.83	16.98	1.26	11.67
Region Queries, Query Region = 1% of Data										
25	0.16	0.17	0.49	1.06	3.05	0.17	0.19	1.01	1.14	6.04
50	4.74	5.30	26.24	1.12	5.54	5.57	6.15	29.85	1.10	5.36
100	12.14	13.27	76.60	1.09	6.31	13.84	14.94	84.53	1.08	6.11
300	36.72	38.92	279.67	1.06	7.62	39.84	42.04	296.80	1.06	7.45
Region Queries, Query Region = 9% of Data										
25	0.20	0.21	1.29	1.08	6.51	0.25	0.25	2.88	1.00	11.57
50	20.11	21.20	76.11	1.05	3.78	22.12	23.10	81.78	1.04	3.70
100	61.78	64.18	228.98	1.04	3.71	65.52	68.12	239.43	1.04	3.65
300	218.61	224.09	769.74	1.03	3.52	226.77	231.62	787.30	1.02	3.47

Table 3: Number of Disk Accesses, Synthetic Data, Buffersize = 250

the NX algorithm has much larger perimeters than the other two algorithms accounting for its poor performance on regions queries.

4.2 GIS tiger data

We now present results for the Long Beach County TIGER data set. In Figure 10 we plot the number of disk accesses versus buffer size for point queries. The data set requires 532 leaf level nodes and 7

Point Data						
	STR 50K	HS 50K	NX 50K	STR 300K	HS 300K	NX 300K
leaf area	0.97	1.33	0.97	0.97	1.31	0.97
total area	3.05	3.64	2.97	3.12	3.76	2.97
leaf perimeter	88.21	106.26	982.49	216.24	258.36	5882.38
total perimeter	101.74	120.76	998.48	243.85	289.45	5948.36
Region Data, Density = 5.0						
leaf area	1.53	1.96	7.58	1.54	1.96	17.47
total area	3.65	4.31	9.63	3.74	4.46	19.63
leaf perimeter	110.82	127.46	1000.77	272.79	312.57	5937.22
total perimeter	124.51	142.09	1016.88	300.78	344.03	6003.54

Table 4: Synthetic Data Areas and Perimeters

Buffer Size	STR	HS	NX	HS/STR	NX/STR
Point Queries					
10	0.72	1.07	3.54	1.49	4.90
25	0.52	0.73	3.08	1.41	5.94
50	0.48	0.63	2.76	1.33	5.78
100	0.42	0.54	2.31	1.27	5.49
250	0.31	0.38	1.39	1.20	4.45
Region Queries, Query Region = 1% of Data					
10	10.51	11.11	35.89	1.06	3.41
25	9.90	10.40	35.59	1.05	3.59
50	8.98	9.38	34.44	1.04	3.83
100	7.83	8.10	31.61	1.04	4.04
250	5.12	5.34	19.25	1.04	3.76
Region Queries, Query Region = 9% of Data					
10	51.17	52.13	107.51	1.02	2.10
25	50.82	51.72	107.32	1.02	2.11
50	49.52	50.54	106.23	1.02	2.15
100	45.67	46.60	101.47	1.02	2.22
250	30.50	31.11	77.17	1.02	2.53

Table 5: Number of Disk Accesses, Long Beach Data, Point and Region Queries and Different Buffer Sizes

	STR	HS	NX
leaf area	0.53	0.76	2.85
total area	2.00	2.51	4.27
leaf perimeter	74.11	76.67	544.30
total perimeter	86.04	89.77	557.07

Table 6: Tiger Long Beach Data, Areas and Perimeters

index nodes for a total of 539 pages. Thus, a buffer of size (10 25 50 100 250) holds (1.86% 4.64% 9.28% 18.55% 46.38%) of the Rtree. The HS algorithm requires 20 - 50% more disk accesses than STR. Again, the relative difference increases as the buffer size decreases. For region queries of sizes up to 9% of the space the two algorithms are similar, with HS requiring 2 - 6% more disk accesses. In Table 5 we present the number of disk accesses and the ratio relative to STR for point and region queries.

We present area and perimeter information in Table 6. The STR algorithm produces significantly smaller areas than both HS and NX, and slightly smaller perimeters than HS.

Buffer Size	STR	HS	NX	HS/STR	NX/STR
Point Queries					
10	14.13	13.67	197.57	0.97	14.45
25	12.80	11.84	197.15	0.93	16.65
50	11.54	10.36	196.27	0.90	18.94
100	9.57	8.48	193.50	0.89	22.81
250	6.46	5.78	177.10	0.90	30.63
500	4.26	4.01	134.57	0.94	33.55
Region Queries, Query Region = 1% of Data					
10	93.98	92.98	605.61	0.99	6.51
25	93.68	92.71	605.48	0.99	6.53
50	93.11	92.07	604.96	0.99	6.57
100	91.53	90.34	602.17	0.99	6.67
250	85.53	84.05	593.18	0.98	7.06
500	76.50	75.51	570.91	0.99	7.56
Region Queries, Query Region = 9% of Data					
10	398.78	396.26	1243.05	0.99	3.14
25	398.44	396.01	1242.93	0.99	3.14
50	398.07	395.63	1242.54	0.99	3.14
100	396.97	394.76	1241.10	0.99	3.14
250	389.71	389.00	1235.85	1.00	3.18
500	369.43	366.31	1216.26	0.99	3.32

Table 7: Number of Disk Accesses, VLSI Data, Buffer Size Varied for Point and Region Queries

4.3 VLSI

The VLSI data set consists of approximately 453,994 rectangles which vary considerably in both size and location. As can be seen in Figure 11, HS and STR perform almost the same (HS performs slightly better) for both point and region queries regardless of buffer size.

In Table 7 we present the detailed results for point and region queries as the buffer size is varied. The HS algorithm performs slightly better than STR for point queries (by a factor of 3% - 11%) and practically the same for region queries. The NX algorithm is significantly worse for both point and region queries. In Table 8 we present area and perimeter information which is consistent with the experimental results.

4.4 Computation Fluid Dynamics

For the experiments in this section we restricted point and region queries to the area bounded by the box (0.48,0.48) (0.6,0.6). When allowed to range over the entire data set there was a large variance in the number of nodes accessed as the remaining area is extremely sparse. Note that the

	STR	HS	NX
leaf area	9.81	8.40	181.06
total area	14.79	14.33	194.54
leaf perimeter	707.92	686.92	7733.60
total perimeter	769.40	753.46	7852.27

Table 8: VLSI Data, Areas and Perimeters

region considered is also highly skewed. Point queries and the lower left corner of region queries are uniformly distributed in the reduced space. Region query size is also reduced to fit within this reduced space. The upper right corner of the region queries were obtained by adding 0.01 or 0.03 to the lower left corner coordinates and truncating at 0.6 if needed. This area roughly corresponds to the 1% and 9% of the data region used in the other experiments.

In Figure 12 we plot the number of disk accesses required for point queries for STR and HS. In Table 9 we present the results of all our experiments and in Table 10 the area and perimeter information for the 52,510 node data set. The data set requires 526 leaf level nodes and 7 index nodes for a total of 533 pages. Thus, a buffer of size (10 15 20 25 50 100 250) holds (1.88% 2.81% 3.75% 4.69% 9.38% 18.76% 46.90%) of the Rtree. As can be seen in Figure 12, for point queries the STR algorithm requires significantly fewer disk accesses than HS, especially for small buffer sizes. For region queries HS and STR perform similarly. The NX algorithm requires significantly more accesses than the other two.

5 Conclusions

All three algorithms studied are based on heuristics and provide no performance guarantees. Thus, it is not surprising that none of them is best for all data sets. By studying the performance of the algorithms on the different types of data we can gain insight into when a specific algorithm performs well and when it does not. We considered three general classes of data: 1) Uniformly distributed point and region data (synthetic) ; 2) Mildly skewed line segment data (tiger) ; 3) Highly skewed, in terms of location and size, region data (VLSI) ; 4) Highly skewed, in terms of location, point data (CFD).

Consider first the uniformly distributed data. For this type of data the HS algorithm requires up to 42% more disk accesses than the STR algorithm for both point and region queries. The NX algorithm performs as well as STR for point queries on point data but much worse for point queries

Buffer Size	STR	HS	NX	HS/STR	NX/STR
Point Queries					
250	0.19	0.21	0.26	1.11	1.38
100	0.25	0.28	0.38	1.15	1.56
50	0.41	0.47	0.53	1.15	1.30
25	0.69	0.81	0.72	1.18	1.05
20	0.79	0.95	0.79	1.20	1.00
15	0.89	1.23	0.88	1.38	0.99
10	1.05	1.76	1.06	1.68	1.01
Region Queries, Query Region Area = 0.0001					
250	2.37	2.42	14.19	1.02	6.00
100	4.88	4.79	23.73	0.98	4.86
50	6.08	5.83	26.81	0.96	4.41
25	6.88	6.66	28.32	0.97	4.12
20	7.15	6.98	28.84	0.98	4.03
15	7.43	7.50	29.27	1.01	3.94
10	7.87	8.41	29.64	1.07	3.77
Region Queries, Query Region Area = 0.0009					
250	11.83	11.72	44.48	0.99	3.76
100	19.05	18.95	66.23	0.99	3.48
50	20.90	20.82	72.68	1.00	3.48
25	22.80	22.31	74.97	0.98	3.29
20	23.45	22.75	75.53	0.97	3.22
15	24.46	23.42	75.91	0.96	3.10
10	25.55	25.02	76.32	0.98	2.99

Table 9: Number of Disk Accesses, CFD 52,510 Node Data, Buffer Size Varied for Point and Region Queries

	STR	HS	NX
leaf area	0.93	1.73	0.88
total area	2.93	4.68	2.87
leaf perimeter	62.15	30.78	206.69
total perimeter	75.54	45.23	223.61

Table 10: CFD 52,510 Node Data Set, Areas and Perimeters

on region data or region queries. As previously pointed out [6], by ignoring all but one dimension the NX algorithm packs in long skinny rectangles (see Figure 2) resulting in a large perimeter of the nodes and hence poor performance for region queries. For all other types of data the NX algorithm does not compete and we drop it from subsequent discussion.

Consider now the mildly skewed tiger data set. The HS algorithm requires up to 49% more disk accesses than STR for both point and region queries. As expected, the difference is more noticeable for smaller buffer sizes.

For highly skewed data, firm conclusions are more difficult to draw. For the VLSI (region) data, HS performed 3% - 11% faster than STR for point queries, and roughly the same for region queries. For the CFD (point) data the situation is reversed: HS required 11% - 68% more disk access than STR for point queries, and roughly the same for region queries.

In summary, no single algorithm is best under all cases. It is clear that the NX algorithm is not competitive, but which of STR or HS to use, depends on the situation at hand. It appears that STR outperforms HS for mildly skewed or uniform data. For highly skewed data, choosing a packing algorithm is a toss up between STR or HS, particularly for region queries. As expected, the importance of choosing a packing algorithm is diminished as either the query size or the buffer size increase.

Developing a new algorithm that works well for all types of data is a challenge that should be pursued. In the future we plan to continue our search for a better packing algorithm, investigate dynamic R-tree variants based on the STR packing algorithm, and also extend our results to a parallel shared-nothing platform.

Acknowledgements

We would like to thank Ken Sevcik for useful discussions about a preliminary version of this work. We would also like to thank Dimitri Mavriplis for providing the CFD data sets.

References

[1] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B., "The $R^{\star}$-tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. ACM SIGMOD, p. 323-331, May 1990.

[2] Bhide, A., Dan, A., Dias, D., "A Simple Analysis of LRU Buffer Replacement Policy and Its Relationship to Buffer Warm-up Transient," Proc. IEEE Data Engineering, 1993.

[3] Chazelle, B., "Filtering Search: A New Approach to Query Answering," SIAM J. Comput., vol. 15, p. 703-724, 1986.

[4] Faloutsos, C., Roseman, S., "Fractals for Secondary Key Retrieval," Proc. Eighth Symposium on Principles of Database Systems (PODS-89), p. 247-252, March 1989.

[5] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD, p. 47-57, 1984.

[6] Kamel, I., Faloutsos, C., "On Packing R-trees," Proc. 2nd International Conference on Information and Knowledge Management (CKIM-93), p. 490-499, Arlington, VA, November 1993.

[7] Kamel, I., Faloutsos, C., "Hilbert R-tree: An Improved R-tree Using Fractals," Proc. International Conference on Very Large Databases 1995 (VLDB-95).

[8] Leutenegger, S.T., Lopez, M.A., "The Effect of Buffering on the Performance of R-Trees," University of Denver Technical Report number 96-2, submitted for publication.

[9] Lopez, M.A., Janardan, R., Sahni, S., "Efficient Net Extraction for Restricted Orientation Designs," to appear in IEEE Transactions on CAD.

[10] Mavriplis, D.J,, "An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness," Journal of Computational Physics, vol. 117, p. 90-101, 1995.

[11] Rosenberg, A.L., Snyder, L., "Time and Space Optimality in B-Trees," ACM Trasactions on Database Systems, vol. 6, no. 1, March 1981.

[12] Roussopoulos, N, Leifker, D., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," Proc. ACM SIGMOD, May 1985.

[13] Sellis, T., Roussopoulos, N., Faloutsos, C., "The R+ Tree: A Dynamic Index for Multidimensional Objects," Proc. 13th International Conference on Very Large Databases (VLDB-87), p. 507-518, September 1987.

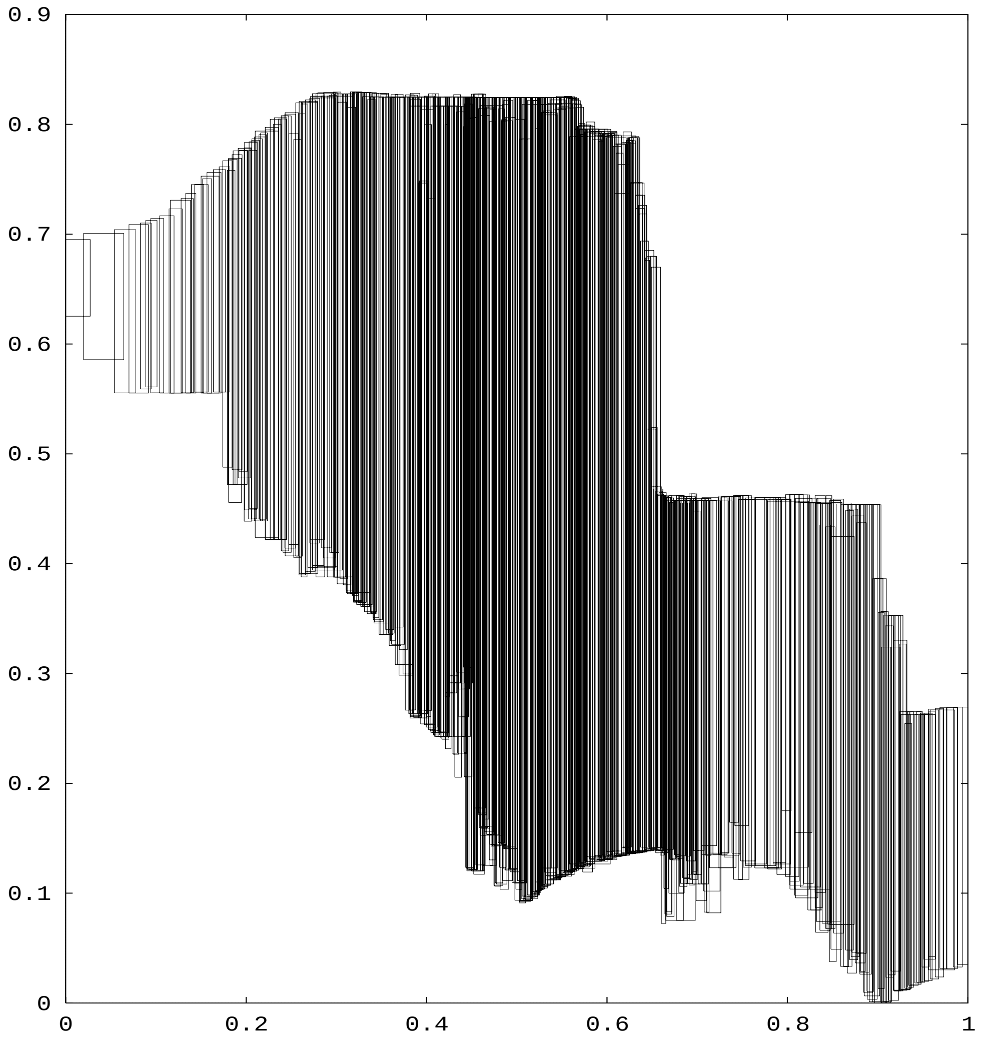

Figure 2: Leaf Bounding Rectangles for Long Beach Data using NX

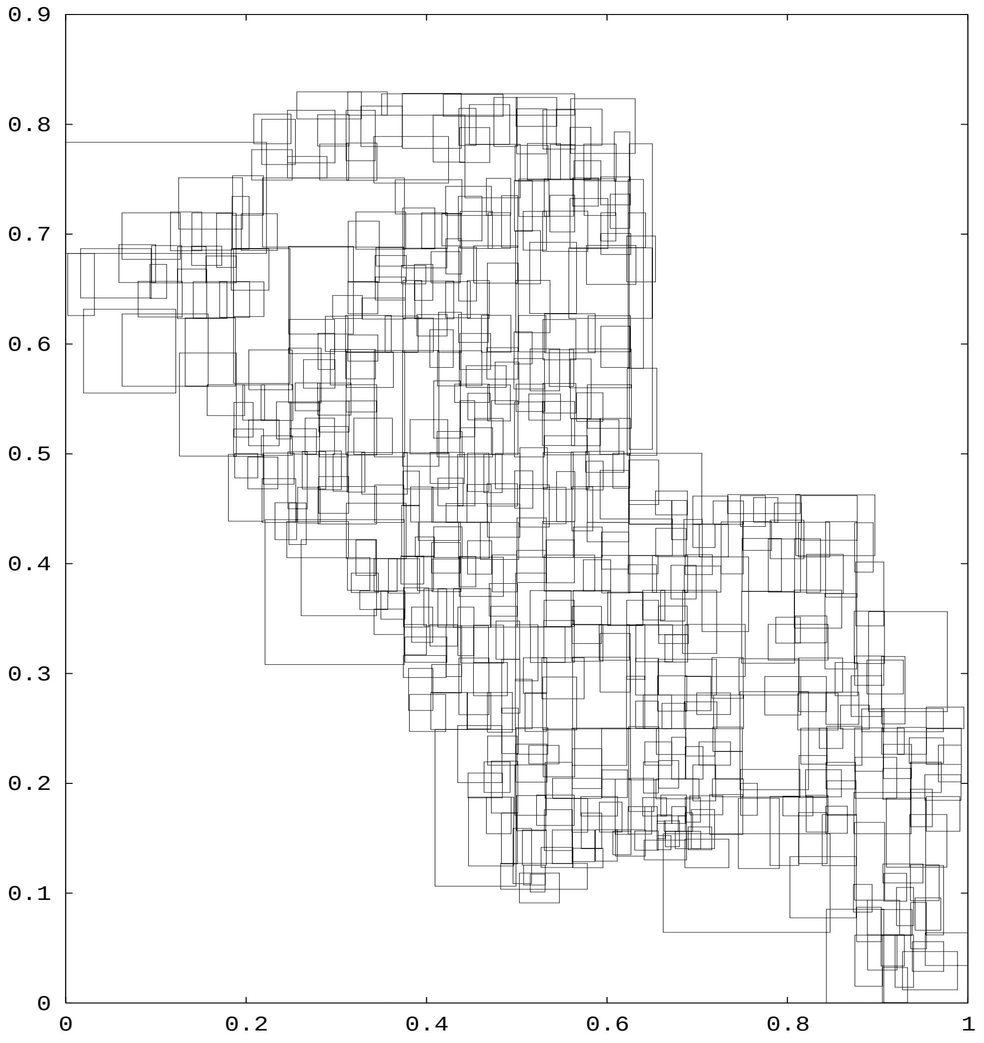

Figure 3: Leaf Bounding Rectangles for Long Beach Data using HS

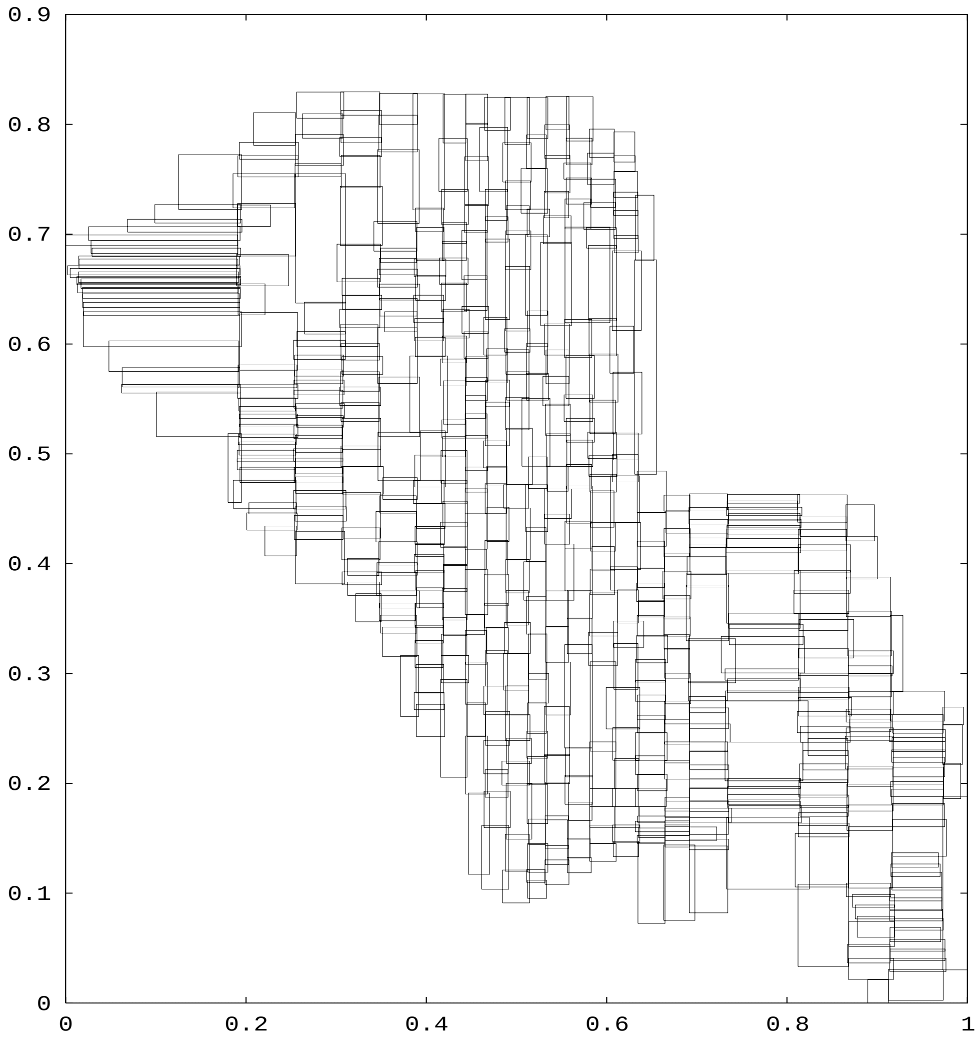

Figure 4: Leaf Bounding Rectangles for Long Beach Data using STR

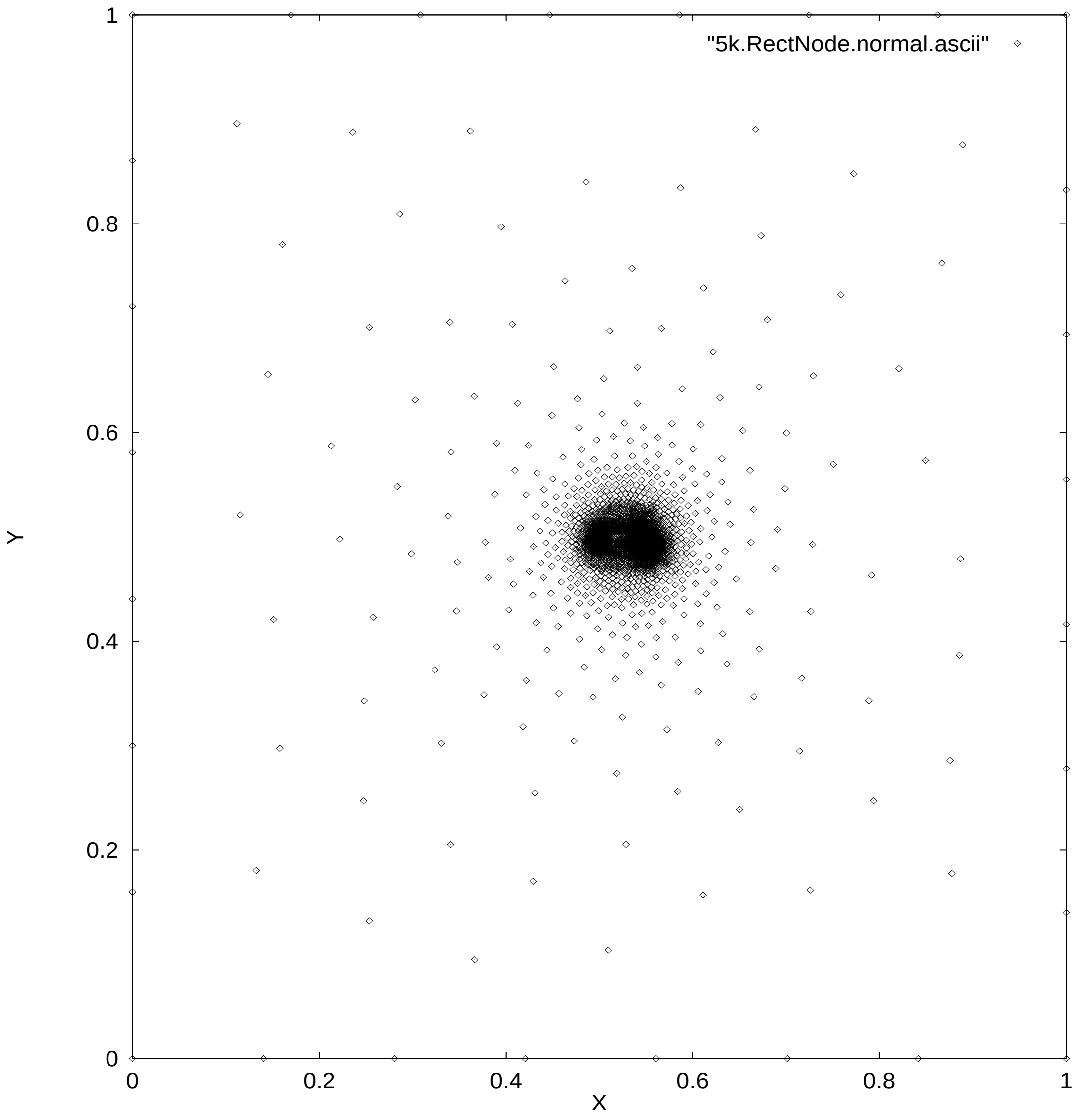

Figure 5: Full Data for 5088 Node Data Set

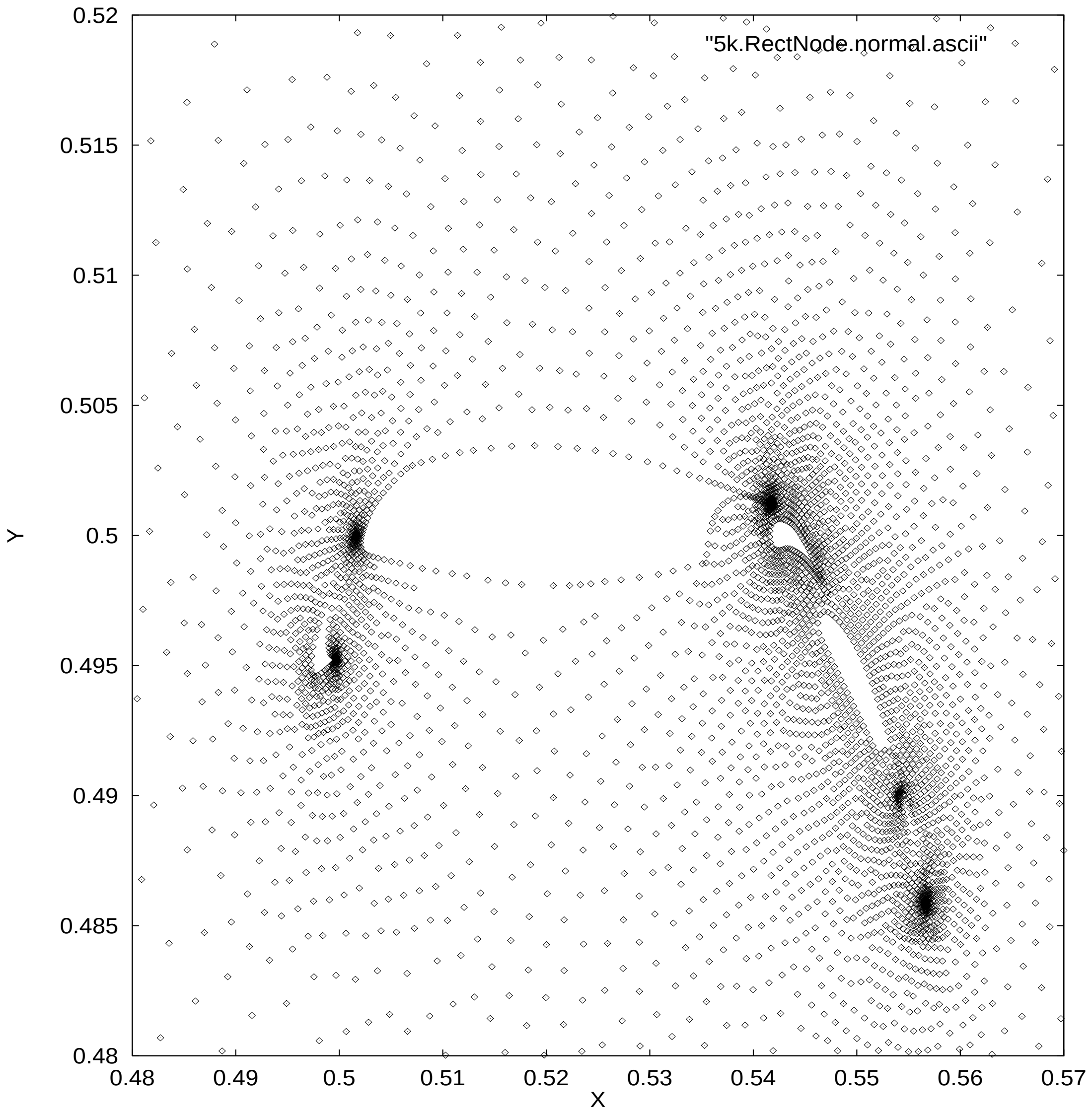

Figure 6: Data Around Center for 5088 Node Data Set

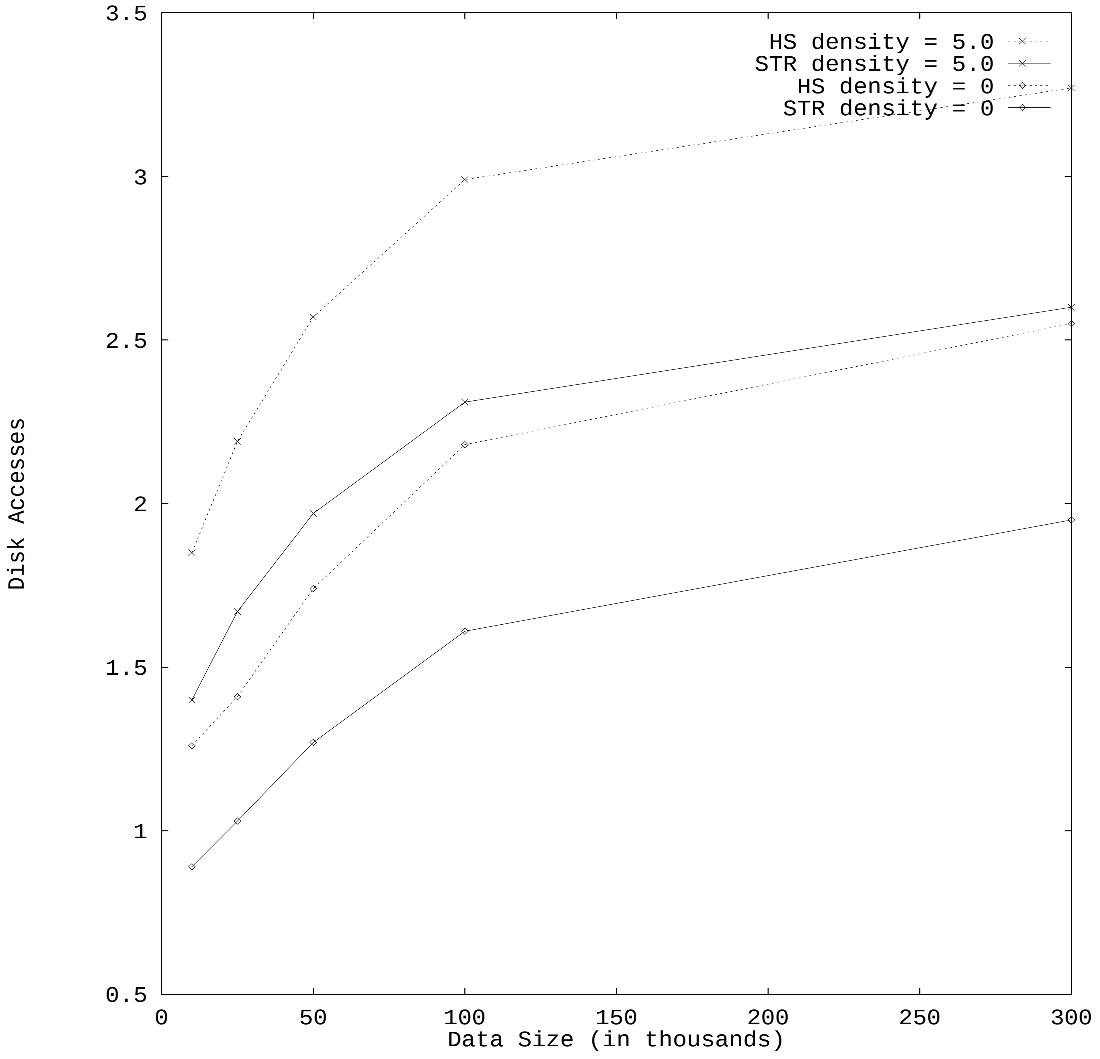

Figure 7: Disk Accesses vs. Data Size for Point Queries on Synthetic Data, Buffer Size 10

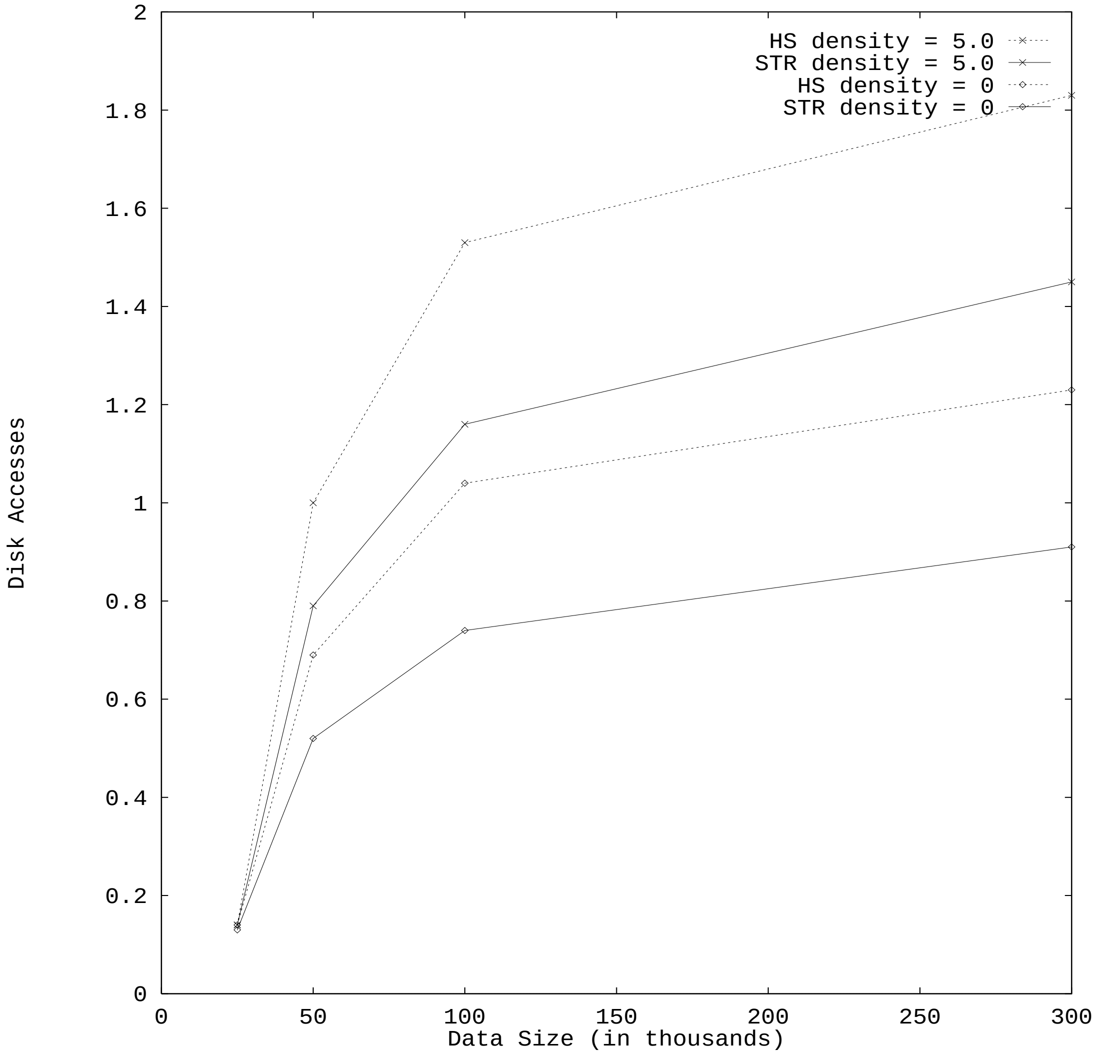

Figure 8: Disk Accesses vs. Data Size for Point Queries on Synthetic Data, Buffer Size 250

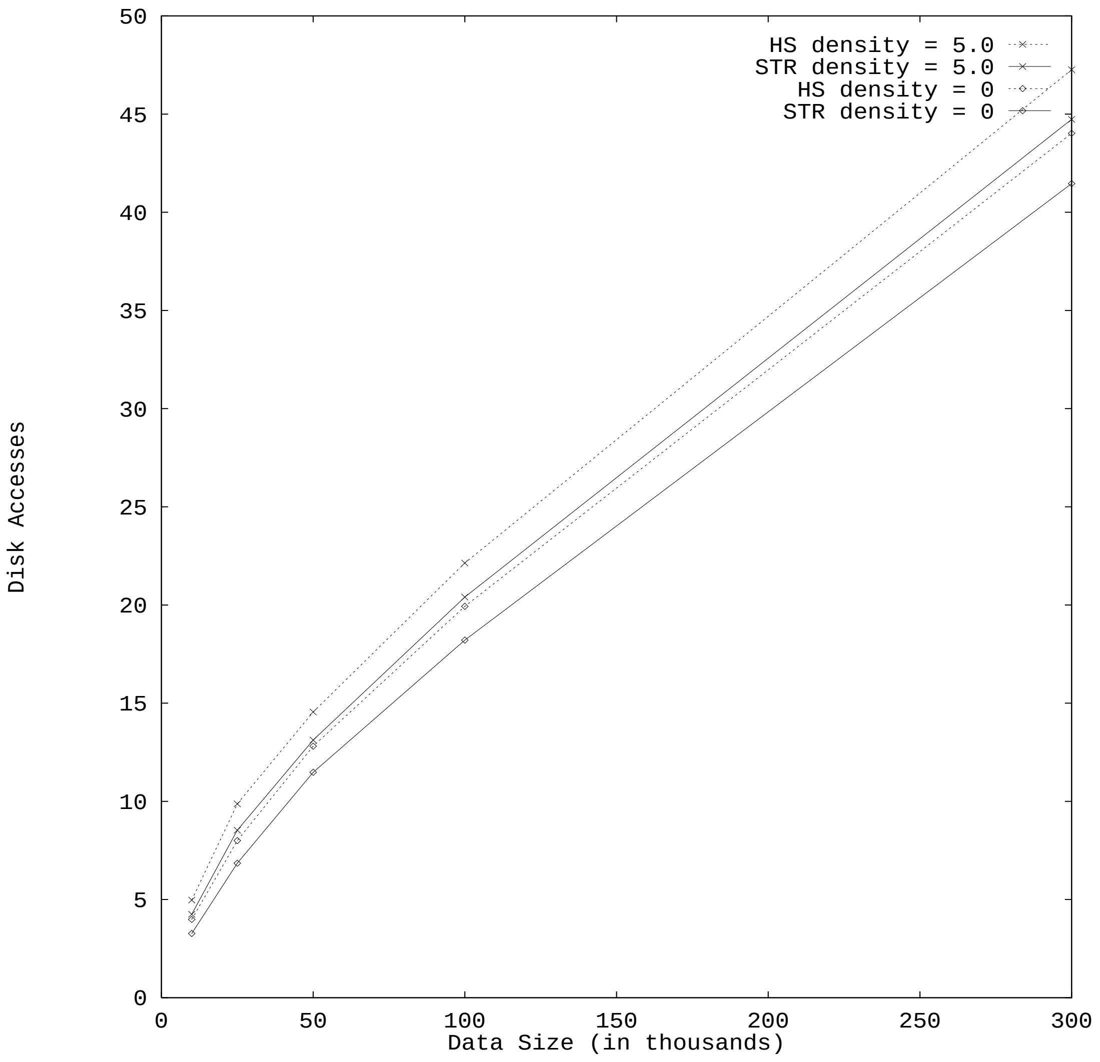

Figure 9: Disk Accesses vs. Data Size for Region Queries on Synthetic Data, Buffer Size 10

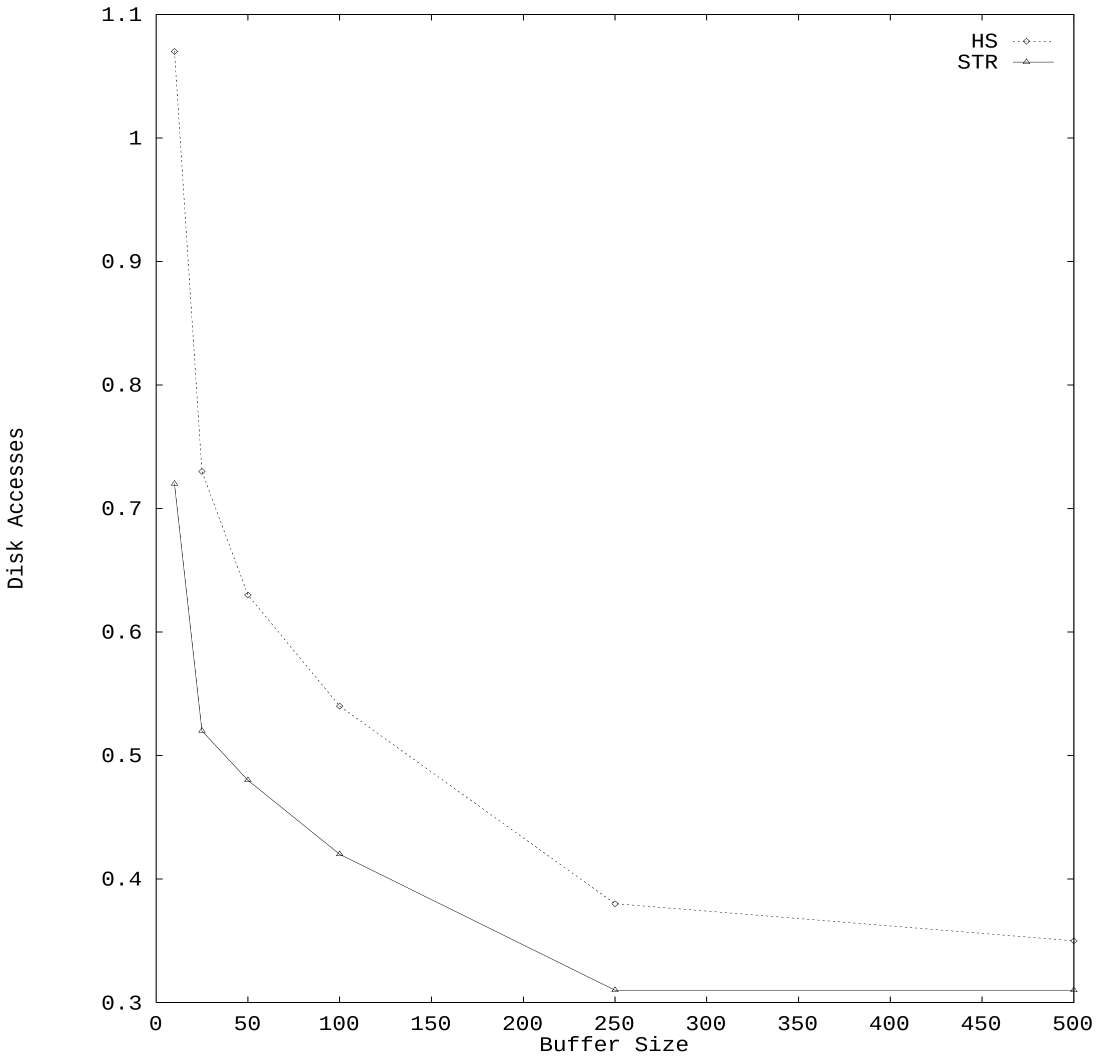

Figure 10: Disk Accesses vs Buffer Size for Point Queries on Long Beach Tiger Data

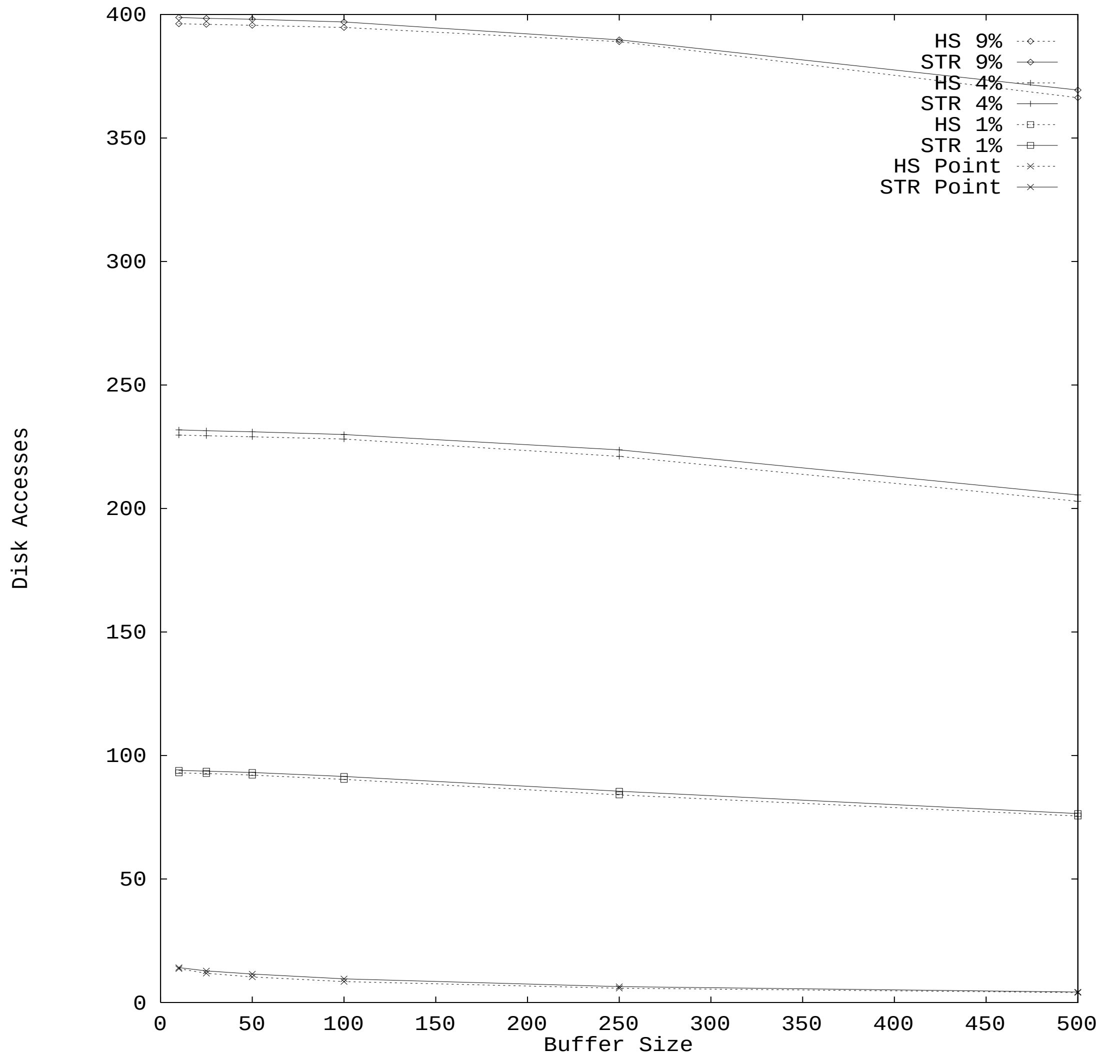

Figure 11: Disk Accesses vs. Buffer Size for Point and Region Queries on VLSI Data

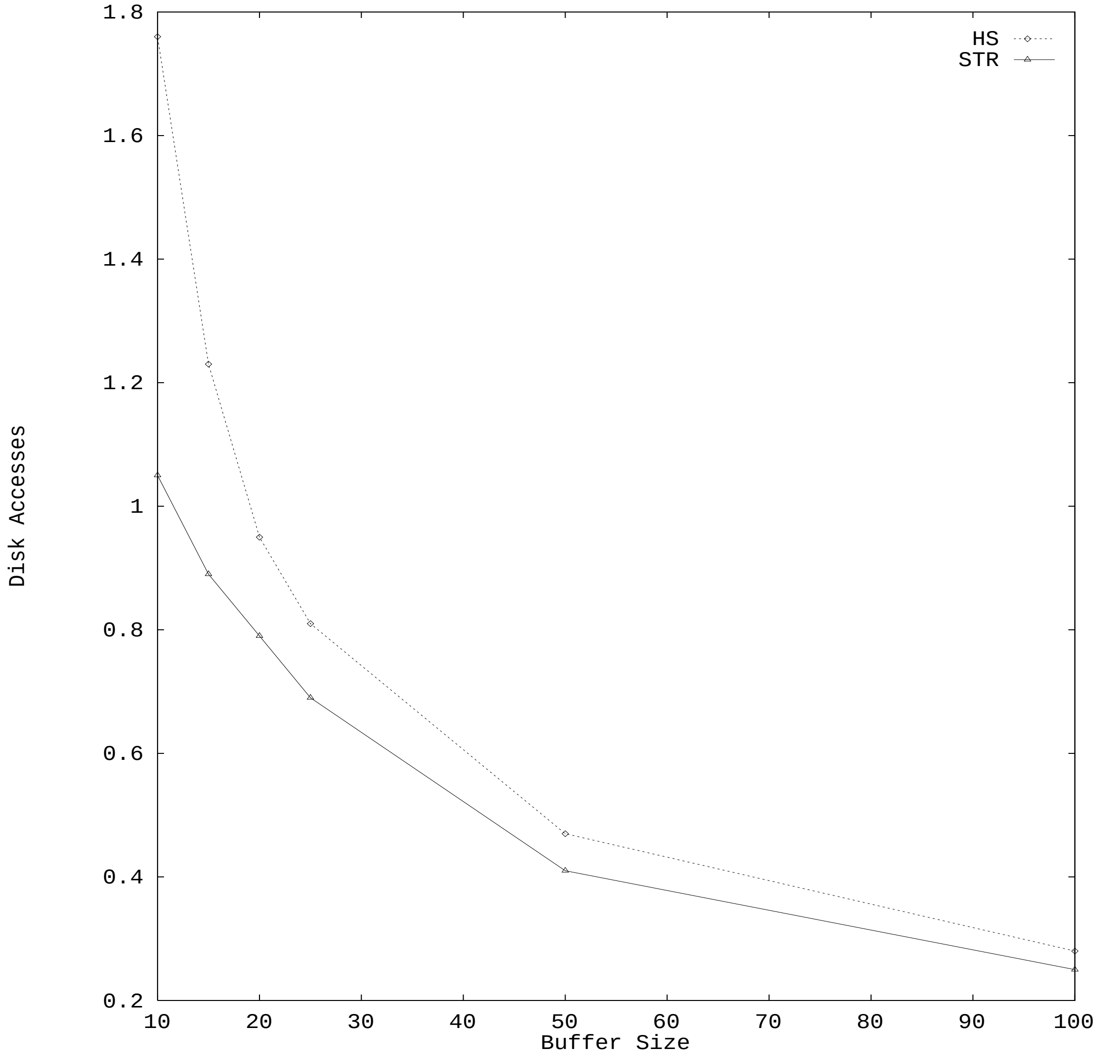

Figure 12: Disk Accesses vs. Buffer Size for Point Queries on CFD Data