University of Denver Computer Science Technical Report # 96-2

The Effect of Buffering on the Performance of R-Trees

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

Abstract

Past R-tree studies have focused on the number of nodes visited as a metric of query performance. Since database systems usually include a buffering mechanism we propose that the number of disk accesses is a more realistic measure of performance. We develop a buffer model to analyze the number of disk accesses required for spatial queries using R-trees. The model can be used to evaluate the quality of R-tree update operations, such as various node splitting and tree restructuring policies, as measured by query performance on the resulting tree. We use our model to study the performance of three well known R-tree packing algorithms. We show that ignoring buffer behavior and using number of nodes accessed as a performance metric can lead to incorrect conclusions, not only quantitatively, but also qualitatively. In addition, we consider the problem of how many levels of the R-tree should be pinned in the buffer.

*This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the first author was visiting the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

1 Introduction

R-trees [3] are a common indexing technique for spatial data and are widely used in spatial and multi-dimensional databases. Typical applications include computer-aided design, geographic information systems, computer vision and robotics, multi-keyed indexing for traditional databases, temporal databases, and scientific databases.

A significant amount of effort has been devoted to the development of better R-tree construction algorithms [9, 1, 8, 4, 5, 6]. Most of this work has focused on proposing and comparing a new algorithm to previous ones, but little work has been done on methodology for comparing algorithms. Notable exceptions are the work of Kamel and Faloutsos [4] and Theodoridis and Sellis [10].

In [4] the authors develop an analytical model for prediction of query performance. The model provides good insight into the problem, especially by establishing a quantitative relationship between performance and the total area and perimeter of the minimum bounding rectangles (MBRs) of the tree nodes, but suffers from one major drawback: the primary objective function for comparison is the number of nodes visited. In real databases some portion of the tree is buffered in main memory. This buffering of portions of the tree can significantly affect performance. Consequently, we propose that a better metric is the average number of disk accesses required to satisfy a query. The work of Theodoridis and Sellis [10] provides a fully analytical model not requiring the minimum bounding rectangles of the R-tree nodes as an input. This work also focuses on the number of nodes visited, but our model could be coupled with their model as we have done with the model of Kamel and Faloutsos.

In order to derive a model based on the number of disk accesses we borrow techniques developed by Bhide *et al* [2] (in the context of modeling databases with uniform access within each of several partitions) and develop a new buffer model. Our model can be used to evaluate the quality of any R-tree update operation, such as node splitting policies [3] or packing algorithms [8, 4] as measured by query performance of the resulting tree. The model is very accurate and simple to understand, making it easy for researchers to integrate it into their studies. Furthermore, it is not only applicable to R-trees, but can easily be modified to model B-tree performance.

The main contributions of this paper are: first, the buffer model methodology; second, a demonstration of the importance of considering a buffer, third, insight into how large a buffer should be used, and fourth, showing the effect of pinning top levels of the R-tree into the buffer. We emphasize that the inclusion of a buffer not only changes the quantitative performance of the algorithms, but in some cases changes the qualitative ordering of the algorithms.

Our buffer model is an analytic model based on simple probability theory, but our overall performance model is a hybrid model. We start by using a packing algorithm to create actual R-trees for various data sets. Then, we compute the minimum bounding rectangles of tree nodes and use these as input to our buffer model.

The rest of the paper is organized as follows. In Section 2 we provide background information on R-trees and describe three loading algorithms that were chosen to test our results. Since this choice is by no means exhaustive, we emphasize that the purpose of this paper is not to draw irrevocable conclusions about the best loading algorithm (although it allows comparisons between the chosen ones), but rather to demonstrate the need and utility of the buffer model. In Section 3 we present the query performance models of Kamel and Faloutsos, our modification of the region query model, and our new buffer model. In Section 4 we present the validation of our model. Section 5 contains results from our model experiments and Section 6 concludes.

2 Overview of R-tree and Loading Algorithms

In this section we provide a brief overview of the R-tree and loading algorithms. A detailed understanding of the loading algorithms is useful but not necessary for the rest of this paper. The reader interested in a more detailed description is referred to the articles listed in the bibliography.

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. An R-tree stores a collection of rectangles which can change in time through insertion and deletion of rectangles. Arbitrary geometric objects can also be handled by representing each object by the smallest upright rectangle enclosing it. R-trees generalize easily to dimensions higher than two. For notational simplicity we describe only the 2D case.

Each node of the R-tree stores a maximum of n entries. Each entry consists of a rectangle R and a pointer P. 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 a path along the tree corresponds to a sequence of nested rectangles, the

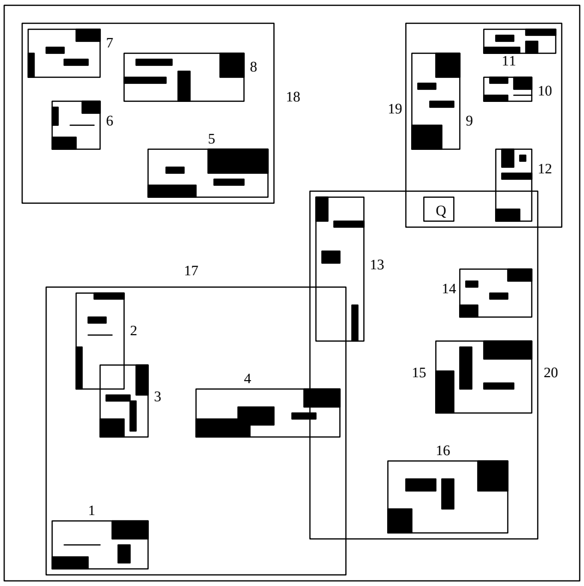

Figure 1: A sample R-tree. Input rectangles are shown solid.

last of which is an actual data object. Note also that rectangles at any level may overlap and that the R-tree for a set of rectangles is by no means unique.

Figure 1 illustrates a 3-level R-tree assuming that 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 pages, numbered 1 to 16. Note that the MBR enclosing each leaf node is the smallest box that fully contains the rectangles within the node. These MBRs serve as rectangles to be stored at the next level of the tree. For example, leaf level nodes 1 through 4 are placed in node 17 in 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 4 level 1 nodes: 17, 18, 19, and 20.

To perform a query Q, all rectangles (internal or not) that intersect the query region must be retrieved. This is accomplished with a simple recursive procedure that starts at the root and possibly follows several paths along the tree. A node is processed by first retrieving all rectangles stored at that node that intersect Q. If the node is an internal node the subtrees pointed to by the retrieved rectangles (if any) are processed recursively. Otherwise, the node is a leaf node and the retrieved rectangles are simply reported. For illustration, consider the query Q in the example of Figure 1. After processing the root node, we determine that nodes 19 and 20 of level 1 must be searched. The search then proceeds in both of these nodes. In both cases it is determined that the query region does not intersect any rectangles within the two nodes 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 based on the insertion algorithms of Guttman [3] 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 attempt to cluster rectangles so as to minimize the number of nodes visited while processing a query.

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

2.2 Loading Algorithms

In this section we describe three loading algorithms. The first is based on inserting a tuple at a time, using one of the insertion algorithms proposed by Guttman [3].

Tuple-At-a-Time (TAT):

> This algorithm just inserts one tuple at a time into the R-tree using the *quadratic split* heuristic of Guttman [3]. Note that the resultant R-tree has worse space utilization and structure relative to the two algorithms discussed next. Thus, more node and disk accesses may be necessary to satisfy a query.

The following two *packing algorithms* use a similar framework, based on preprocessing the entire data file so as to determine how to group rectangles into nodes. In the following we assume that a data file of R rectangles will be stored in a tree with up to n rectangles per node. The whole processes is similar to building a B-tree out of a collection of keys from the leaf level up [7].

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 are intended to be placed in the same leaf level node. Note that the last group may contain less than n rectangles.

2. Load the $\lceil R/n \rceil$ groups of rectangles into pages and output the tuples (MBR, page-number) for each leaf level page into a temporary file. The page-numbers are needed for setting child pointers in the nodes one level up.

3. Recursively pack these MBRs into nodes at the next level and up until only the root node remains.

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

Nearest-X (NX):

This algorithm was proposed in [8]. The rectangles are sorted by x-coordinate. No details are given in the paper so we assume that x-coordinate of the rectangle's center is used. The rectangles are then packed into the nodes using this ordering.

Hilbert Sort (HS):

A fractal based algorithm was proposed in [4]. 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 as measured along the Hilbert Curve. This determines the order in which the rectangles are placed into the nodes of the R-Tree.

3 Model Description

Kamel and Faloutsos [4] introduced an analytical model for computing the average response time of a query as a function of the geometric characteristics of the R-tree. Their model does not consider the effect of a buffer and performance is measured by the number of R-tree nodes accessed. In practice, query performance is mainly affected by the time required to retrieve nodes touched by the query which do not reside in the buffer, as the CPU time required to retrieve and process buffer resident nodes is usually negligible. We start by describing the model of Kamel and Faloutsos. We then modify this model to better fit our definition for uniformly distributed queries of a fixed size. Finally, we extend the model to take into account the existence of a buffer, including how to handle the pinning in the buffer of the top few levels of the R-tree.

In this section we consider a 2-dimensional data set consisting of rectangles to be stored in an R-tree T with $H+1$ levels, labeled 0 through H. All input rectangles have been normalized to fit within the unit square $U = [0,1] \times [0,1]$. We assume that queries are rectangles Q of size $q_x \times q_y$ uniformly distributed over the unit square. Note that a point query corresponds to the case $q_x = q_y = 0$. Our description concentrates on 2D. Generalizations to higher dimensions are straightforward.

Throughout this section we will use the following notation,

$$
\begin{aligned}
M_i &= \text{number of pages at the } i\text{th level of } T \\
M &= \text{Total number of pages in } T\text{, i.e., } \textstyle\sum_{i=0}^{H} M_i \\
R_{ij} &= j\text{th rectangle at the } i\text{th level of } T \\
X_{ij} &= x\text{-extent of } R_{ij} \\
Y_{ij} &= y\text{-extent of } R_{ij} \\
A_{ij} &= \text{area of } R_{ij}\text{, i.e., } X_{ij} \cdot Y_{ij} \\
A_{ij}^{Q} &= \text{probability that } R_{ij} \text{ is accessed by query } Q \\
B_{ij} &= \text{number of accesses to } R_{ij} \\
\mathcal{A} &= \text{Sum of the areas of all MBRs in } T \\
L_x &= \text{Sum of the x-extents of all MBRs in } T \\
L_y &= \text{Sum of the y-extents of all MBRs in } T \\
N &= \text{number of queries performed so far} \\
N^* &= \text{expected number of queries required to fill the buffer} \\
B &= \text{buffer size} \\
D(N) &= \text{number of distinct pages (at all levels) accessed in } N \text{ queries} \\
E_T^P(q_x, q_y) &= \text{expected number of pages (buffer resident or not) of } T \\
&\ \text{accessed while performing a query of size } q_x \times q_y \\
E_T^D(q_x, q_y) &= \text{expected number of disk accesses while performing a query} \\
&\ \text{of size } q_x \times q_y
\end{aligned}
$$

3.1 The Model of Kamel and Faloutsos

Kamel and Faloutsos consider a bufferless model where performance is measured by number of nodes accessed (independent of whether they currently reside in the buffer). They observe that for uniform point queries the probability of accessing R_{ij} is just the area of R_{ij}, namely, A_{ij}. They point out that the level of T in which R_{ij} resides is immaterial as all rectangles containing Q (and only those) must be retrieved. Accordingly, for a point query the expected number of nodes retrieved as derived by Kamel and Faloutsos is[1]

$$E_T^P(0,0) \;=\; \sum_{i=0}^{H}\sum_{j=1}^{M_i} A_{ij} = \mathcal{A} \tag{1}$$

which is the sum of the areas of all rectangles (both leaf level MBRs as well as MBRs of internal nodes).

We now turn our attention to region queries. Let $\langle (a,b),(c,d)\rangle$ denote an upright rectangle with bottom left and top right corners (a,b) and (c,d), respectively. Consider a rectangular query

[1] We have modified the notation of [4] to make it consistent with the notation used in this paper.

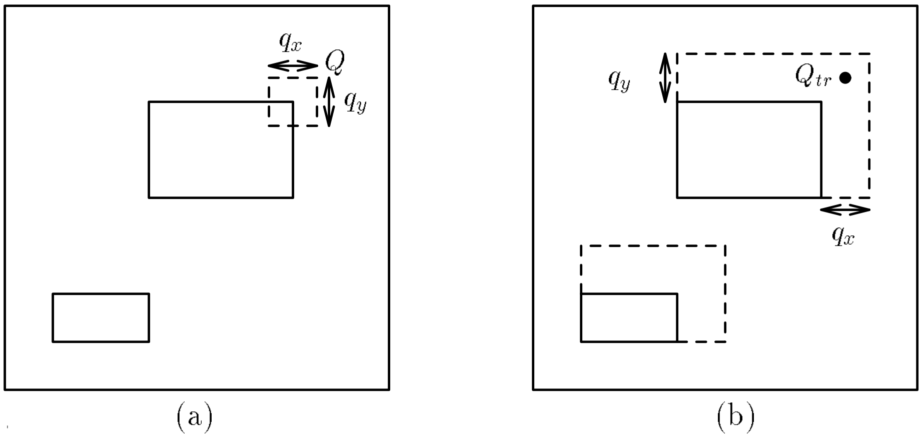

Figure 2: (a) Two data rectangles and region query Q (b) Corresponding extended rectangles and equivalent point query Q_{tr}.

$Q = \langle Q_{bl}, Q_{tr} \rangle$ of size $q_x \times q_y$. Q intersects $R = \langle (a,b),(c,d) \rangle$ if and only if Q_{tr} (the top right corner of Q) is inside the *extended rectangle* $R' = \langle (a,b),(c+q_x, d+q_y) \rangle$, as illustrated in Figure 2.

Kamel and Faloutsos infer that the probability of accessing R while performing Q is the area of R', as the region query Q is equivalent to a point query Q_{tr} where *all* rectangles in T have been extended as outlined above. Thus, the expected number of nodes retrieved (as derived in [4]) is:

$$
\begin{aligned}
E_T^P(q_x, q_y) &= \sum_{i=0}^{H} \sum_{j=1}^{M_i} (X_{ij} + q_x)(Y_{ij} + q_y) \\
&= \sum_{i=0}^{H} \sum_{j=1}^{M_i} (X_{ij} Y_{ij} + q_x \sum_{i=0}^{H} \sum_{j=1}^{M_i} Y_{ij} + q_y \sum_{i=0}^{H} \sum_{j=1}^{M_i} X_{ij} + M q_x q_y \\
&= \mathcal{A} + q_x L_y + q_y L_x + M q_x q_y \qquad (2)
\end{aligned}
$$

Equation 2 illustrates the fact that a good packing algorithm should cluster rectangles so as to minimize both the total area and total perimeter of the rectangles in internal nodes of the R-tree. For point queries, on the other hand, $q_x = q_y = 0$, and minimizing the total area is enough.

3.2 A Modification to the Model for Region Queries

Consider the set of uniformly distributed region queries of size $q_x \times q_y$. Such a query is a rectangle of the prescribed dimensions that fits entirely within the unit square (as otherwise, a rectangle would be equivalent to a possibly much smaller rectangle).

There are two potential problems with using Equation 2 for analyzing the performance of queries of a fixed size:

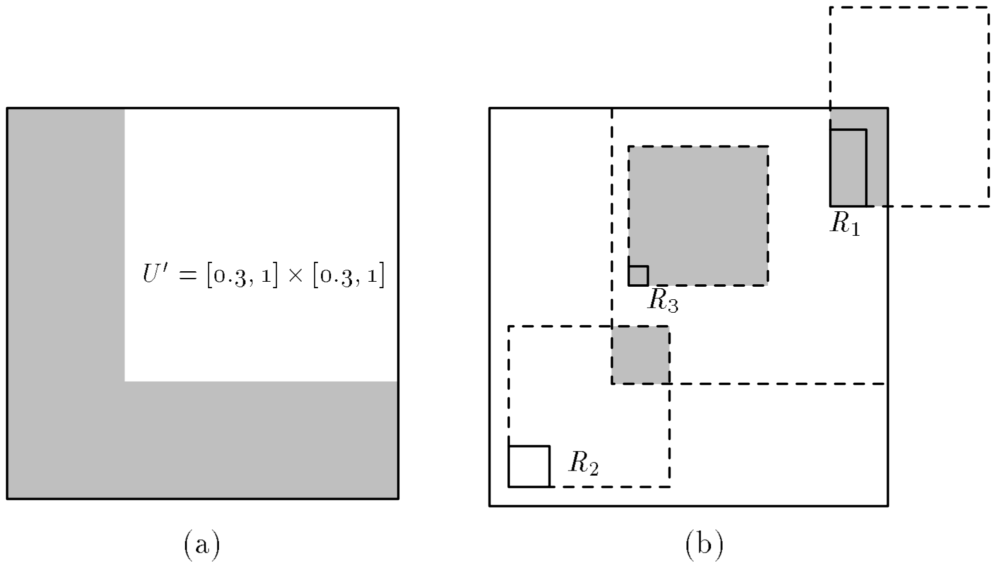

Figure 3: (a) The domain of Q_{tr} for a query of size 0.3×0.3 is U' (area not shaded), (b) Three data rectangles. The probability of accessing R_i is the area of $R_i' \cap U'$ (shown shaded) divided by the area of U'. Formula (2) uses the area of R_i' (shown dashed) instead..

1. For uniformly distributed rectangular queries of size $q_x \times q_y$ the top right corner, Q_{tr}, of the query region Q, cannot be an arbitrary point inside the unit square if the entire query region is to fit within the unit square. Rather, Q_{tr} must be inside the box $U' = [q_x, 1] \times [q_y, 1]$.

2. The probability of accessing a rectangle $R = \langle(a,b),(c,d)\rangle$ is *not* the area of $R' = \langle(a,b),(c+q_x, d+q_y)\rangle$ as this value can be bigger than one. Rather, the access probability is the percentage of U' covered by the rectangle $R' \cap U'$:

$$A_{i,j}^{Q} = \frac{\text{area}(R' \cap U')}{\text{area}(U')} = \frac{[\min(1, c+q_x) - \max(a, q_x)][\min(1, d+q_y) - \max(b, q_y)]}{(1-q_x)(1-q_y)} \tag{3}$$

The difference between Formulas 2 and 3 is small if Q is small but becomes much larger as the size of Q increases. This is illustrated in Figure 3, where a region query of size 0.3×0.3 is considered. Note that Q_{tr} must fall inside U', i.e., outside the shaded area of Figure 3a. Three rectangles (R_1, R_2, and R_3) of sizes 0.1×0.2, 0.1×0.1, and 0.05×0.05, respectively, are shown in Figure 3b. Only for R_3 does Equation 2 compute the same probability as Equation 3. The probability that Q will access R_1 (resp. R_2) is 7.65% (resp. 4.59%) with our modified model and 20% (resp. 16%) with the original model of Kamel and Faloutsos.

The difference is, arguably, a matter of interpretation, but we point out that the original model predicts that the probability of accessing rectangle $\langle(0.1, 0.1),(0.9, 0.9)\rangle$ with a query of size 0.3×0.3 is 121%! With our modification the probability of accessing this rectangle is 100%. We point out that under our model the probability of access of input rectangles is no longer uniform: rectangles within the shaded region of Figure 3a are less likely to be accessed than rectangles inside U'. An alternative that guarantees uniform access probability is to allow Q_{tr} to fall uniformly inside $[0,1+q_x] \times [0,1+q_y]$ (which still requires a correction to the model of Kamel and Faloutsos, namely $A_{i,j}^{Q} = (X_{ij}+q_x)(Y_{ij}+q_y)/(1+q_x)(1+q_y)$). Under this model, query windows could partially lie outside the area of interest, thus effectively reducing the actual query size. Since we want the model to measure performance as a function of a specific query size, in the rest of the paper we use our former interpretation, even though this means that data along the edges of the data set might be accessed less frequently.

In both our buffer model as well as in the experiments performed to validate it, access probabilities were computed by using Equation 3.

From now on we use A_{ij}^{Q} to denote the probability that rectangle R_{ij} will be accessed while performing query Q. This probability is computed by applying Formula 3. Note that if Q is a point, $q_x = q_y = 0$, and A_{ij}^{Q} is simply A_{ij}, the area of R_{ij}.

3.3 A Buffer Model

Bhide *et al* [2] analyze the LRU buffer replacement policy for databases consisting of a number of partitions with uniform page access within each partition. While modeling performance during buffer warm-up they observed that the buffer hit probability at the end of the warm-up period is a good estimator of the steady state buffer hit probability.

We conjectured that similar behavior would occur in the context of R-trees when applying the LRU replacement policy, namely, that the steady state buffer hit probability is virtually the same as the buffer hit probability when the buffer *first* becomes full. Our conjecture was verified experimentally as discussed in Section 4.

Under uniformly distributed point queries, the probability of accessing rectangle R_{ij} while performing a query is A_{ij}. Accordingly, the probability that R_{ij} is not accessed during the next N queries is $P[B_{ij} = 0|N] = (1 - A_{ij})^N$. Thus, $P[B_{ij} \geq 1|N] = 1 - (1 - A_{ij})^N$ and the expected number of *distinct* pages accessed in N queries is,

$$D(N) = \sum_{i=0}^{H} \sum_{j=1}^{M_i} P[B_{ij} \geq 1|N] = M - \sum_{i=0}^{H} \sum_{j=1}^{M_i} (1 - A_{ij})^N \tag{4}$$

Note that $D(0) = 0 < B$ and $D(1) = \mathcal{A}$ (which may or may not be bigger than B). The buffer, which is initially empty, first becomes full after performing N^* queries, where N^* is the smallest integer that satisfies $D(N^*) \geq B$. The value of N^* can be determined by a simple binary search:

ComputeN^*()
    high $\leftarrow 1$;
    while $D(high) < B$ **do**

```
        high ← 2 * high;
    low ← high/2;
    N* ← ⌈(low + high)/2⌉;
    while D(N*) < B or D(N* − 1) ≥ B do
        if (D(N*) < B)
            low ← N*;
        else
            high ← N*;
        N* ← ⌈(low + high)/2⌉;
    return N*;
end
```

Now, while the buffer is not full the probability that R_{ij} is in the buffer is equal to $P[B_{ij} \geq 1]$. The probability that a random query requires a disk access for R_{ij} is $A_{ij} \cdot P[B_{ij} = 0]$.

Since the steady state buffer hit probability is approximately the same as the buffer hit probability after N^* queries, the expected number of disk accesses for a point query at steady state is

$$\sum_{i=0}^{H}\sum_{j=1}^{M_i} A_{ij} \cdot P[B_{ij} = 0|N^*] = \sum_{i=0}^{H}\sum_{j=1}^{M_i} A_{ij} \cdot (1 - A_{ij})^{N^*} \tag{5}$$

The above derivation also holds for region queries provided that A_{ij}^{Q} is used instead of A_{ij} (see Formula 3).

We summarize the results of this section below:

> The expected number of disk accesses required to satisfy a uniformly distributed random query Q of size $q_x \times q_y$ is:
>
> $$E_T^D(q_x, q_y) = \sum_{i=0}^{H}\sum_{j=1}^{M_i} A_{ij}^{Q} \cdot (1 - A_{ij}^{Q})^{N^*}$$
>
> where A_{ij}^{Q} is the probability that Q intersects R_{ij} as given by Equation 3.

Finally, we point out that it is easy to extend the above results to model a buffer management policy that pins the top few levels of the R-tree in the buffer: simply reduce the number of buffer pages by the number of pages in these pinned levels and omit the top levels from the model.

4 Model Validation

We validated the model by comparing it with simulation. The simulation models an LRU buffer and, like the model, takes as input the list of the MBRs for all nodes at all levels. It then generates

	Rtree-1			Rtree-2			Rtree-3		
bufsize	simul	model	% dif	simul	model	% dif	simul	model	% dif
50	2.3120	2.2682	1.189	2.2060	2.1651	1.185	1.5892	1.5807	0.53
100	1.7008	1.6862	0.86	1.6288	1.6166	0.75	1.2336	1.2275	0.49
200	1.1627	1.1599	0.24	1.1194	1.1172	0.20	0.8720	0.8710	0.11
400	0.9347	0.9340	0.07	0.9009	0.9007	0.02	0.6944	0.6944	0.00
1000	0.4701	0.4700	0.02	0.4534	0.4532	0.04	0.3551	0.3551	0.00
1400	0.1770	0.1769	0.06	0.1714	0.1712	0.12	0.1368	0.1368	0.00

Table 1: Validation: average number of disk accesses per query for model and simulation.

random point queries in the unit square and checks each node's MBR to see if it contains the point. If the MBR does contain the point the node is requested from the buffer pool. If the node is not in the buffer pool, the least recently used node in the buffer is pushed out and the new node put on the top of the LRU stack. Note that the simulation is accurate in the sense that an R-tree implementation retrieves all and only those rectangles (internal or not) that intersect the region query. Confidence intervals were collected using batch means with 20 batches of 1,000,000 queries each, resulting in confidence intervals of less than 3% at a 90% confidence level.

We ran comparisons for three different R-trees and 6 different buffer sizes for each tree. Each of the R-trees has 1668 nodes, but the MBRs of the nodes are different, as produced by the three different packing algorithms. In Table 1 we report the average number of pages required from disk per point query as predicted by the simulation, model, and the percent difference (relative to the simulation). All of the results are within 2%, which is less than the confidence intervals returned from the simulation. Other comparison experiments not shown resulted in agreement within 2% or less. Simulation of region queries gave similar results.

5 Model Results

In this section we present performance results as predicted by our model. We consider two types of queries: point queries and region queries. A point query specifies a point in the unit square and finds all rectangles that contain the point.

Region queries specify a region of a given size and find all rectangles that intersect with this region. Each query is a rectangle of size $q_x \times q_y$ whose upper right corner is contained in region U' as described in Section 3.2.

5.1 Need For Consideration of Buffer Impact

Figure 4 plots the number of disk accesses versus buffer size for the Long Beach data set extracted from the TIGER system of the U. S. Bureau of Census, assuming R-trees with 100 rectangles per node. The left plot is for point queries, the right for region queries. Consider the top two curves of the right plot. For small buffer sizes the TAT algorithm requires fewer disk access than the NX algorithm. At a buffer size of 200 the performance of the two algorithms crosses and the NX algorithm becomes the better algorithm. Hence, ignoring buffering would result in the incorrect conclusion that TAT is better than NX, thus underscoring the importance of including buffer effects in comparison studies.

A second experiment shows that not considering the buffer impact can introduce significant errors. In Figure 5 we present results for synthetic data. Uniformly distributed data sets were created containing between 10,000 and 300,000 rectangles. For each rectangle the lower left corner was uniformly distributed over the unit square. All rectangles are squares, where the length is chosen uniformly between 0 and ϵ. The value of ϵ is fixed for all data set sizes and is equal to $2\sqrt{0.25/10000}$. Thus, for a 10,000 rectangle data set the sum of the rectangle areas is equal to 0.25 of the unit square, for 100,000 rectangles the total area equals 2.5 times the unit square. This is similar to the experimental methodology used in [4].

The top left graph plots the number of nodes visited (i.e. no buffering considered) versus the number of rectangles in the data set. The top right graph plots the number of disk accesses versus data set size for a buffer size of 10, and the bottom graph for a buffer size of 300. Ignoring buffer impact (top left) leads to the conclusion that querying an R-tree of 300,000 rectangles is no more expensive than querying an R-tree of 25,000 rectangles. This could cause a query optimizer to produce a poor query plan. Once buffering is considered, the fact that larger trees are more expensive is evident.

5.2 Choosing Buffer Size for R-trees

In this section we study the reduction in disk accesses obtained from increasing the buffer size. Since main memory is a valuable resource, some insight into the gains from allocating more buffer space is needed. Consider Figure 4 again where the number of disk accesses versus buffer size is plotted for both point and region queries run against the Long Beach tiger data set. The data set has 53,145 rectangles (actually line segments). A node size of 100 rectangles is assumed resulting in the HS and NX algorithms having 532 pages at the leaf level, 6 pages at level 1, and 1 root node. Buffer size is varied from 2 to 500 pages (from 0.4% to 92.8% of the R-tree).

The left plot is for point queries. The curves plotted from top to bottom are TAT, NX, and HS. The R-tree produced by the TAT algorithm is poorly structured, and, as a result, seems to benefit significantly from each increase in buffer size. The HS tree is much better structured and, while experiencing a halving of the number of required disk accesses for a buffer size of 10%, additional buffer increases only help modestly. The NX algorithm does not experience as much of a reduction in needed disk accesses for small buffer sizes as the HS algorithm. We hypothesize that the better the R-tree structure, the more capable it is to capitalize on small amounts of buffer for point queries, whereas the worse the R-tree structure, the more linear the reduction in disk accesses.

The right plot is for 1% region queries. Note that for region queries none of the curves have a

data size	level 1 (root)	level 2	level 3	level 4
40000	1	3	64	1600
60000	1	4	96	2400
80000	1	6	128	3200
100000	1	7	160	4000
120000	1	8	192	4800
150000	1	10	240	6000
200000	1	13	320	8000
250000	1	16	400	10000

Table 2: Number of nodes per level

well defined “knee”. Thus, based on this experiment it appears it is possible that when executing region queries reductions in disk accesses for increasing the buffer size is more linear than for point queries.

5.3 Choosing the Number of Levels to Be Pinned

Past buffer management studies have shown that for B-trees the root and maybe the first level should be pinned in the buffer pool. Pinning nodes decreases the buffer size available for other pages, but guarantees that the pinned pages will be in the buffer. We present experiments to investigate what gains in performance that can be expected from pinning R-tree nodes, and how many levels should be pinned.

For the experiments in this section we wanted to get deeper R-trees while keeping the experiment time reasonable. To do this we created synthetic data sets with 40,000 - 250,000 rectangles and used R-tree nodes of size 25. This resulted in 4 level R-trees with numbers of nodes per level as shown in Table 2.

In Figure 6 we plot the number of disk accesses for point queries versus the data size for three buffer sizes and for trees created by the HS algorithm. Results for the other algorithms were similar. The plot on the top left is for a buffer of 500 pages; that on the top right if for a buffer of 1000 pages; and the bottom graph is for a buffer of 2000 pages. The number of disk access remains roughly the same for no pinning, pinning the first (root) level, and pinning the first two levels. All three scenarios are plotted as the single top line in all three graphs. The bottom line is for pinning the third level in addition to the first two levels.

As a general rule of thumb, pinning a level makes a big difference if the total number of pages pinned is within a factor of two of the buffer size, but when the total number of pages pinned is less than one third of the buffer size, only marginal benefits are seen. This is because for a smaller relative number of pinned pages the LRU replacement policy succeeds in keeping the top levels in the buffer without explicit pinning. For example, for a buffer size of 500 and a data size of 250,000 rectangles, pinning three levels pins 417 pages. This results in 53% fewer disk accesses. When the data size is 80,000 rectangles, 135 pages are pinned and the saving is only 4%. For a buffer size of 2000, pinning of the the first three levels makes almost no difference since the number of pages pinned is less than one fourth of the buffer size.

Thus, for point queries pinning is advantageous, but only when the total number of nodes pinned is within a factor of 2 of the buffer size. Note that pinning never hurts performance and hence for a fixed size buffer dedicated to an R-tree application as many levels as possible should be pinned. If the buffer is shared among many applications, the benefit of pinning must be compared to the other applications. Our buffer model can be used to predict the benefit accrued by pinning a number of R-tree levels.

For region queries all experiments to date (not shown) have resulted in only a modest (2-5%) improvement from pinning. Thus, it appears that pinning may mostly benefit point queries, and even then only under special scenarios.

6 Conclusions

We have developed a new buffer model to predict the number of disk accesses required per query for a given input R-tree (specified by the minimum bounding rectangles of each node in the tree). Our model has been shown experimentally to agree with simulation within 2% for numerous test cases. The model is both simple to implement and quick to solve, thus providing a useful methodology for further studies.

In order to develop our model, we first modified an earlier R-tree region query model. This modified query model is used as a component of our overall model to predict the number of disk accesses required per query.

With our model, we have demonstrated that using the number of nodes accessed per query as a performance metric is not sufficient, because it ignores buffer effects. We show that once buffer effects are taken into account not only does quantitative predicted performance change, but the qualitative predictions can change as well. In particular, the actual ordering of policies can differ when buffering is not considered versus when it is considered. Thus, it is essential to consider buffer effects in a performance study of R-trees and use number of disk accesses as the primary metric.

In addition, we used our model to determine that for the data considered, small amounts of buffer can super-linearly improve performance of well structured R-trees for point queries, but that poorly structured trees experience only a more linear improvement in performance due to the increase in buffer space. In addition, we find that for region queries even well structured R-trees result in a more linear improvement as buffer is added.

Finally, we show that in most cases pinning the top levels of the R-tree has little effect on performance relative to just using an LRU buffer. For point queries we did find certain scenarios where if the buffer size is a about the same or a few times larger than the number of nodes at a level, pinning that level does significantly improve performance.

Acknowledgments. We would like to thank Jeff Edgington for developing the R-tree packing algorithm code. We would also like to thank Ken Sevcik for useful discussions about a preliminary version of this work.

References

[1] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The $r^{\star}$-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*, pages 323–331, may 1990.

[2] A. Bhide, A. Dan, and D. Dias. A simple analysis of lru buffer replacemtn policy and its relationship to buffer warm-up transient. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, 1993.

[3] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

[4] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proc. of Second Int. Conf. on Information and Knowledge Management (CIKM)*, Washington, D.C., November 1993.

[5] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: an improved r-tree using fractals. In *Proc. of VLDB Conference,*, pages 500–509, Santiago, Chile, September 1994.

[6] S.T. Leutenegger, J.M. Edgington, and M.A. Lopez. Str: A simple and efficient algorithm for r-tree packing. Technical Report 96-01, Mathematics and Computer Science Department, University of Denver, 1996.

[7] A.L. Rosnberg and L. Snyder. Time and space optimality in b-trees. *ACM Transactions on Database Systems*, 6(1), 1981.

[8] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proc. ACM SIGMOD*, Austin, Texas, May 1985.

[9] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+ tree: A dynamic index for multidimensional objects. In *Proc. of 13th International Conference on Very Large Databases (VLDB-87)*, pages 507–518, sep 1987.

[10] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *Proc. of 8th Symposium on Principles of Database Systems (PODS)*, May 1996.

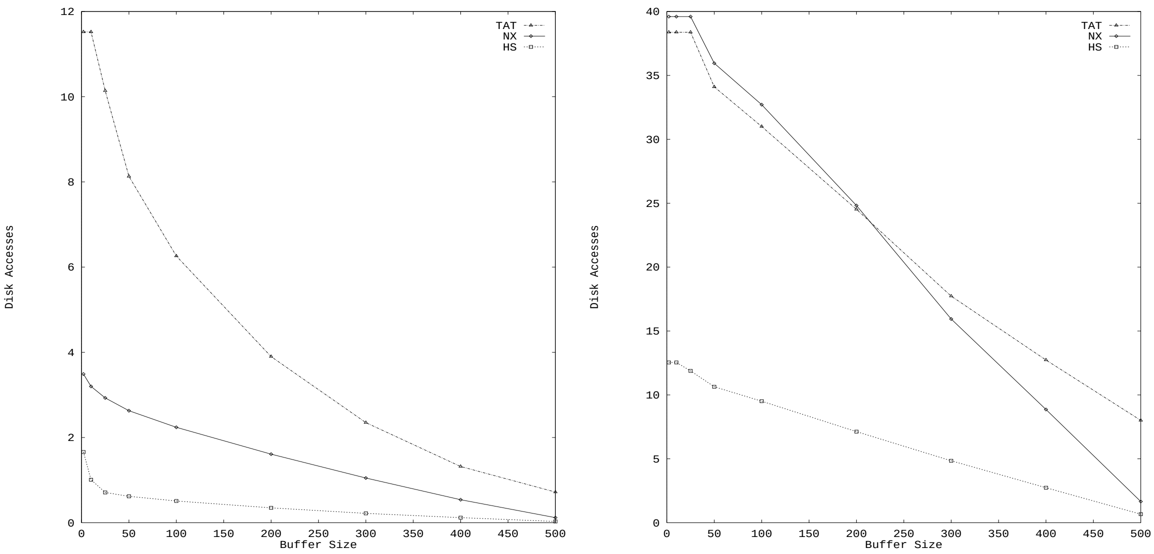

Figure 4: Sensitivity to Buffer Size, Long Beach Data: Left Point Queries; Right: 1% Region Queries

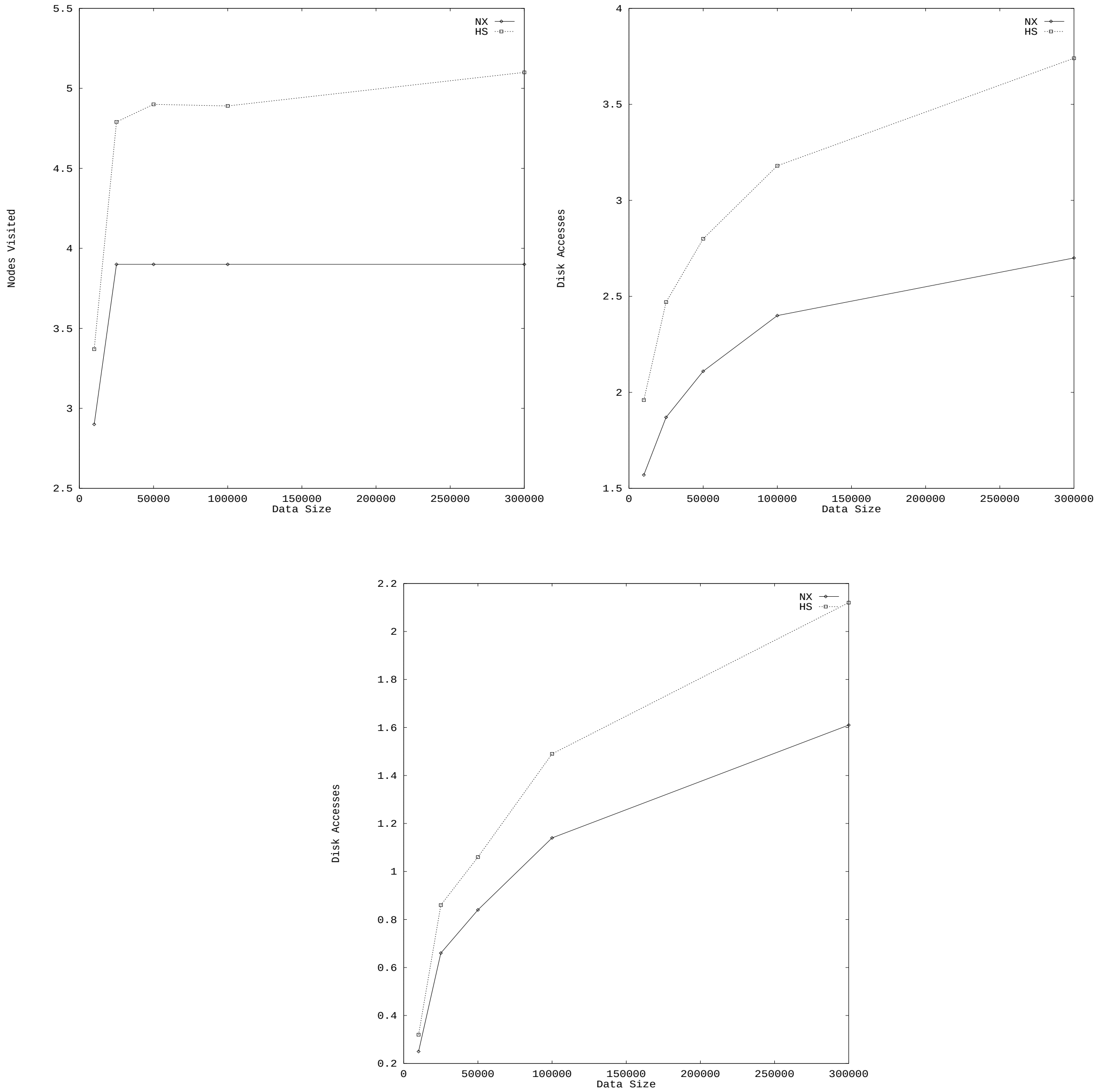

Figure 5: Disk Accesses versus Data Size for NX and HS: Top Left: NoBuffer ; Top Right: Buffer Size = 10 ; Bottom: Buffer Size = 300

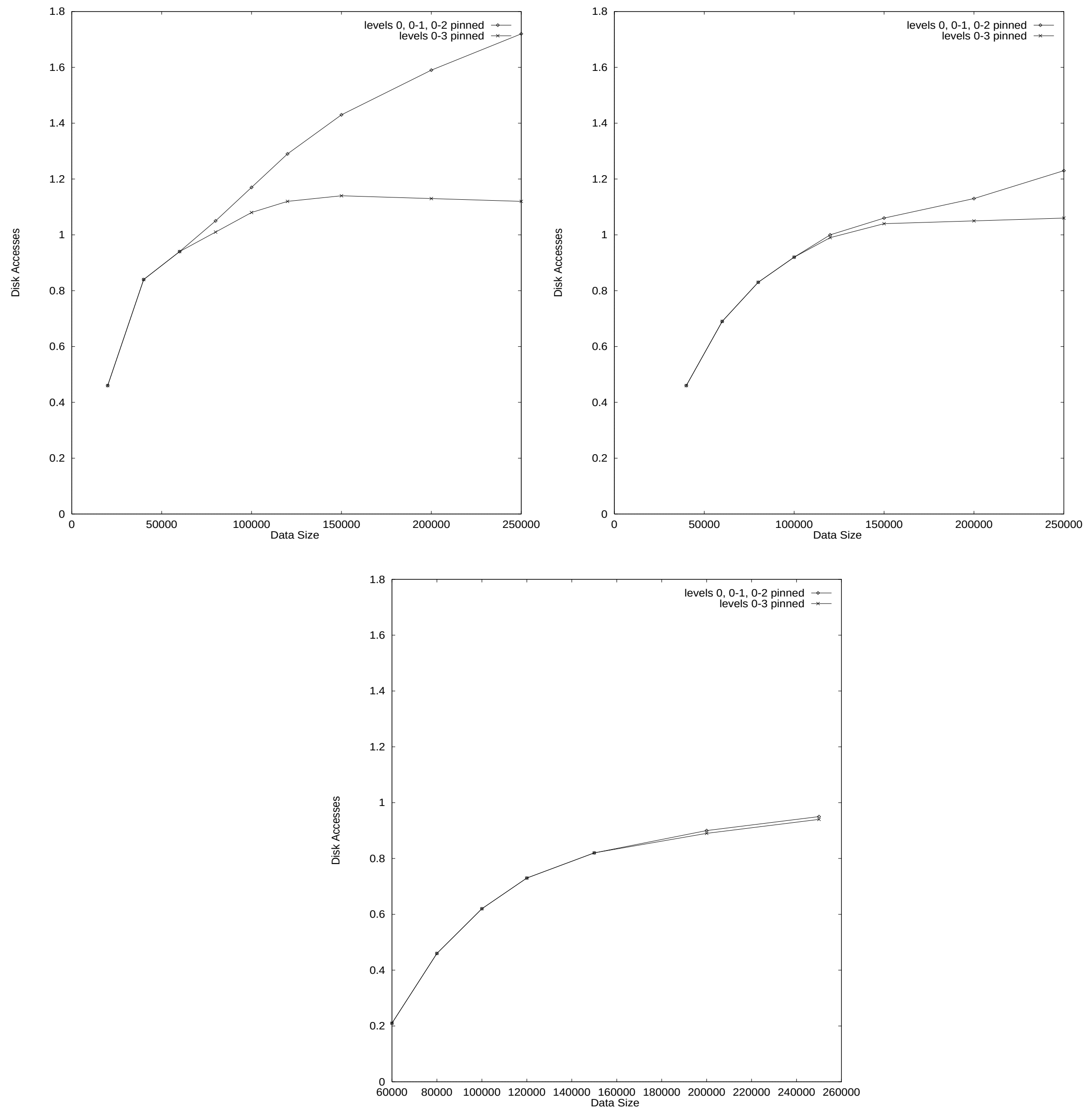

Figure 6: Effect of Pinning, Disk Accesses versus Data Size for HS: Top Left: Buffer Size = 500 ; Top Right: Buffer Size = 1000 ; Bottom: Buffer Size = 2000