Rethinking Choices for Multi-dimensional Point Indexing: Making the Case for the Often Ignored Quadtree

Position Paper

You Jung Kim and Jignesh M. Patel
Department of Electrical Engineering and
Computer Science
University of Michigan
2260 Hayward, Ann Arbor, MI, USA
{youjkim, jignesh}@eecs.umich.edu

ABSTRACT

Multi-dimensional point indexing methods play a critical role in a variety of data-centric applications, ranging from image retrieval, sequence matching, and protein structure comparison. Many of these applications require manipulating point data in low to medium dimensional space, either because of the inherent nature of the problem, or due to the use of dimensionality reduction techniques such as PCA. A common choice of indexing method for these applications is often the "ubiquitous" R*-tree. In this paper, we challenge this popular choice of indexing for low and medium dimensional point data and investigate the use of Quadtree as an alternative index structure.

Our paper shows that the regular and disjoint decomposition method used by Quadtrees provides a significant structural advantage over the R*-tree, which suffers from high overlap amongst MBRs even for low dimensional data. In addition, the unbalanced nature of the Quadtree has a surprisingly beneficial effect on the buffer pool utilization. Using analytical models and extensive empirical evaluation we show that the often ignored Quadtree far outperforms the R*-tree (and the Pyramid-Technique) for indexing low and medium dimensional point datasets. Consequently, our work makes an important contribution, which motivates the reconsideration of the often ignored Quadtree indexing structure for the common problem of multi-dimensional indexing.

1. INTRODUCTION

Due to the demanding need for efficient multi-dimensional indexing methods in many database applications, significant research effort has been invested towards developing new multi-dimensional indexing methods. Of these indexing methods, the R*-tree [2], one of the R-tree [10] variants,

is most widely used.

Since it is well-known that the performance of the R*-tree deteriorates rapidly with increasing data dimensionality [4], the R*-tree is not used for handling very high-dimensional datasets. However, high-dimensional datasets are plagued with the curse of dimensionality, and general high performance high-dimensional indexing methods that work across a variety of applications still remain a legitimate research goal (and is likely to continue to be an open research problem for at least the near future). However, for many high-dimensional applications, a practical method is to apply a dimensionality reduction method such as principal components analysis (PCA) to transform the high-dimensional dataset into a low or medium dimensional datasets (usually with 8 or fewer dimensions). The use of R*-tree for low and medium dimensional point indexing has been advocated for many years and many applications have been built using R*-trees. A few examples of such applications are similarity search in sequence databases [1], image retrievals in multimedia databases [6, 17], subsequence matching in time-series databases [15, 16], and similarity searches in protein structure databases [19]. A common characteristic of these examples is that they all rely on R*-trees for indexing multi-dimensional point datasets, which usually have less than 8 dimensions. In this paper we critically examine this conventional choice and explore the use of the Quadtree index [18] as a more efficient alternative to the "ubiquitous" R*-tree for indexing low and medium dimensional datasets.

We note that Quadtree structure has been around for many decades even within the context of database systems [9, 11], but it is still not as widely accepted as the R*-tree. We speculate that the reasons for largely ignoring the Quadtree for database indexing are: 1) the unbalanced nature of the index structure, especially for skewed data, and 2) the mismatch between the size of a non-leaf node and a disk block size. Nevertheless, the Quadtree has some nice properties as it employs a disjoint and regular space partitioning strategy. This partitioning method has an advantage over the R*-tree which suffers from rapidly increasing overlap of MBRs even for relatively low dimensionality. Furthermore, we observe that the Quadtree is better at handling skewed data as it is not encumbered by problems caused by methods that require producing a balanced index structure. In addition, its unbalanced structure actually provides an advantage for

buffer management as it results in better spatial locality compared to other balanced index structures. In addition, drawbacks related to the large fan-out and low node occupancy in the Quadtree can be alleviated to some extent with efficient node *packing* techniques which we employ in our implementation.

There are more than a few dozen indexing structures for multi-dimensional point data [3, 4, 8, 14], and each year a few more indexing structures are proposed. Among these alternatives, the Pyramid-Technique [3] is notable and it was shown to be far superior to the Hilbert R-tree [12] and the X-tree [4]. The Pyramid-Technique is also based on the disjoint space partitioning similar to the Quadtree. However, instead of using a balanced space split, it adopts an unbalanced space split strategy. We note that the Pyramid-Technique has been shown to outperform existing methods for very high dimensional data, but its performance is unknown for low and medium dimensional data. As an alternative to the R*-tree, we also examine the Pyramid-Technique extensively.

Some previous work [13] has compared the R-tree and the Quadtree in a commercial database, but that study only focuses on a 2 dimensional GIS spatial data and the performance of the R-tree and the Quadtree is not compared for higher-dimensional point data.

In this paper, we compare the performance of the R*-tree, the Quadtree, and the Pyramid-Technique for low and medium dimensional point data. Using disk-based indices implemented in SHORE [5], we demonstrate that the Quadtree significantly outperforms the other two indexing methods. Consequently, we make the case that it is time to consider the unbalanced Quadtree index for efficient querying on low to medium dimensionality point datasets.

2. IMPLEMENTATION

In this section, we describe our implementation of the R* tree, Pyramid-Technique, and Quadtree indices as disk-based index structures in the SHORE [5] storage manager.

SHORE has an R*-tree implementation, but it only works for two dimensions. So, we implemented a general R*-tree in SHORE following the description in [2]. In addition, we use the following optimization for an efficient disk layout: Once the R*-tree is created using multiple inserts, the original index is traversed using a breadth first method, and each visited node is rewritten to disk sequentially. As a result, we create an R*-tree such that all non-leaf and leaf sibling nodes are clustered sequentially on disk. The clustering of non-leaf and leaf sibling nodes results in efficient disk access. Compared to the original R*-tree, the optimized R*-tree improves query performance by about 5-20%. (We have not implemented a bulkloading algorithm which can lead to faster loading times.)

To make sure that our R*-tree implementation is not slower compared to the 2D R*-tree implementation in SHORE, we compared the performance of window queries on 2D datasets for the two implementations, and found that our implementation is actually consistently slightly faster than the implementation in SHORE.

The Pyramid-Technique is based on transforming high dimensional points to one dimensional points, and then using the B^+ tree to index the one dimensional points. For window query processing, the query is transformed into a set of range searches on the B^+ tree. We implemented both naive

and extended Pyramid-Techniques as described in [3]. The naive Pyramid-Technique is used for uniform data and the extended Pyramid-Technique is used for skewed data. We simply use the SHORE B^+ tree implementation and the bulkload algorithm that is already implemented in SHORE.

The Quadtree is an index structure based on a hierarchical regular decomposition of space. We implemented the Quadtree in the following way: First, a Quadtree is created using records of a constant (known) size for the non-leaf nodes, and using the page size for the leaf node (technically we are implementing a bucket Quadtree [18]). Then, we traverse the original index starting from the root using a breadth first search. During the traversal, we copy each visited node and write it sequentially to disk, producing a new (final) Quadtree index file. When copying each node, both non-leaf nodes and leaf nodes are packed to variable size records, leaving only a small amount of space to allow for updates. Finally, at the end of the traversal of the original Quadtree, we simply point to the new file as the Quadtree index file, and delete the original file. Essentially, our method produces a compact disk layout for the Quadtree index and is a quick prototype for a bulkloading algorithm.

Note that in our Quadtree construction method, we are using a compact packing method. This is crucial for an effective Quadtree index since the internal Quadtree nodes can be much smaller than a disk page, and the occupancy of the leaf-level buckets is often very low. In addition, the breadth-first traversal method for packing is a natural order for packing as sibling nodes are frequently co-accessed during the processing of a query. (The impact of the packing technique is described in more detail in Section 4.4).

For concurrency control, our current Quadtree implementation simply uses the record-level locking in SHORE. For consistency update queries start by grabbing an exclusive lock on the root node. A more detailed discussion of highly efficient concurrency control method for Quadtree indices is beyond the scope of this work and is part of future work. However, it seems that the regular and *non-overlapping* decomposition structure of the Quadtree can potentially be effectively exploited using predicate-based locking methods or optimistic concurrency control methods.

3. ANALYTICAL COMPARISON

In this section, we will analytically compare the performance of R*-tree, Pyramid-Technique, and Quadtree indices. For this comparison, we use an analytical formula proposed in [3]. This analytical formula requires as input information about the minimum bounding rectangles (MBR) in an index structure, and then estimates the average number of page accesses for a range query Q based on this information.

We also considered another analytical methods. For instance, an analytical formula proposed in [7] estimates page accesses in R-tree variants only using data characteristics, without requiring information about the actual index structure. However, this formula works accurately only if the data dimensionality is fairly small, for the following reason. This formula is derived assuming square-shaped MBRs and a good index structure which has no overlap amongst MBRs. In the case of R-tree variants, this is a highly optimistic assumption even for low and medium dimensional data. As shown in [4], as data dimensionality increases, the ratio of MBR overlaps increases rapidly. The increased overlaps of

Table 1: Notations

Symbol	Definition
d	number of dimensions
P	number of pages in an index
$M_i = (L_i, H_i)$	MBR of a node i with low and high coordinates L_i and H_i
$L_i = (l_{i,1}, ..., l_{i,d})$	d-dimensional low coordinate of a MBR i
$H_i = (h_{i,1}, ..., h_{i,d})$	d-dimensional high coordinate of a MBR i
$Q = (q_1, ..., q_d)$	range query with side lengths $q_1,....,q_d$
$DA(i)$	probability of accessing a page i
PA	total number of pages accessed by a range query Q

MBRs in turn result in increased page accesses. Since the formula does not consider overlap among MBRs, the estimation error will increase with increasing data dimensionality. To check this observation, we experimentally compared the actual number of page accesses and predictions from [7]. For this experiment, we used 2, 4, and 8 dimensional uniform data sets with 2 million points and range queries of size 1% volume of the data space. The estimation errors from [7] for the R*-tree are 4%, 95%, and 99% for 2, 4, and 8 dimensional datasets.

We note that the formula in [3] can be directly used for the R*-tree. However, the formula needs to be adjusted for the Pyramid-Technique and our packed Quadtree. In the following discussion, we will present the original analytical formula presented in [3] and adapted formulas for the Pyramid-Technique and our packed Quadtree. We make the following assumptions:

1. The data space is a d-dimensional unit hypercube $[0..1]^d$.

2. Queries are hypercubes uniformly distributed over the unit hypercube and they are always placed completely inside the unit hypercube.

3.1 R*-tree

Given a query $Q = (q_1, ..., q_d)$ and the information about MBRs in an index, the expected number of page accesses is computed in the following way. First, for the query Q, the probability of accessing a page i, $DA(i)$, is calculated. Next, the expected number of page accesses is calculated as a sum of $DA(i)$ over all pages i in the index.

In the case of a point query Q, considering that the volume of the data space is 1, $DA(i)$ corresponds to the volume of a MBR, M_i, in a page i. For a range query Q, $DA(i)$ corresponds to the volume of the MBR enlarged by the size of Q, and the formula to compute $DA(i)$ for the range query Q is:

$$DA(i) = \prod_{j=1}^{d} (H_{i,j} - L_{i,j} + q_j) \quad (1)$$

We note that the above formula does not consider a boundary effect in which an enlarged MBR exceeds the boundary of the data space. Although the boundary effect can be negligible in low dimensional spaces, estimation errors in predicting page accesses increases as data dimensionality increases. To adapt the formula to the boundary effect, the following formula is proposed in [3].

$$DA(i) = \prod_{j=1}^{d} \frac{\min(H_{i,j}, 1 - q_j) - \max(L_{i,j} - q_j, 0)}{1 - q_j} \quad (2)$$

To consider the boundary effect, [3] assumes that a range query Q is always completely placed inside the data space. This assumption essentially reduces the data space volume to $\prod_{j=1}^{d} 1 - q_j$. Furthermore, with this assumption, one has to adjust the boundaries of the MBRs in the index to fall within the reduced area. This adjustment is accomplished in Equation 2 by ensuring that the minimum and the maximum values of the MBRs are within the reduced area. A detailed proof of the above formula is given in [3].

Then, the expected number of page accesses is the sum of $DA(i)$ over an entire index, and can be calculated as:

$$PA = \sum_{i=1}^{P} DA(i)$$

3.2 Quadtree

The formulas used for the R*-tree can also be used for a regular Quadtree, but not for our packed Quadtree. To estimate the number of page accesses accurately, it is necessary to adapt the formulas to our Quadtree packing scheme.

The formulas for the R*-tree assume that a node i in an index corresponds to a single page i. Therefore, $DA(i)$, is proportional to the volume of the MBR, M_i, enclosing the node i. However, in our packed Quadtree, several nodes can be packed into a single page (see Section 2). Note that since our packing strategy packs nodes in a breadth-first order, these nodes need not be adjacent in space, and some of these nodes could be at different levels in the index.

To calculate $DA(i)$, we first have to calculate the volume of the MBRs in a page enlarged by Q. Let $V'(i)$ denote this volume. For the MBRs, $M_1,...,M_m$, in a page i, $V'(i)$ is:

$$
\begin{aligned}
V'(i) = \quad & \sum_{i=1}^{m} V(M_i, Q) - \sum_{i<j=2}^{m} V(M_i \cap M_j, Q) \\
+ \quad & \sum_{i<j<k=3}^{m} V(M_i \cap M_j \cap M_k, Q) + ... \\
+ \quad & (-1)^{m-1} V(M_1 \cap M_2 \cap M_3 \cap ... \cap M_m, Q)
\end{aligned}
$$

where $V(M_i, Q) = \prod_{j=1}^{d} \min(H_{i,j}, 1-q_j) - \max(L_{i,j} - q_j, 0)$.

In the above formula, $V(M_i, Q)$ represents the volume of a MBR i enlarged by Q and it is calculated after removing the parts of the MBR outside the effective data space (for reasons discussed in Section 3.1).

$$DA(i) = \frac{V'(i)}{\prod_{j=1}^{d}(1 - q_j)} \quad (3)$$

Note that in the above formula the denominator is needed since the volume of the data space is not 1, but $\prod_{j=1}^{d}(1 - q_j)$ since we assume that a range query Q is completely positioned inside the data space.
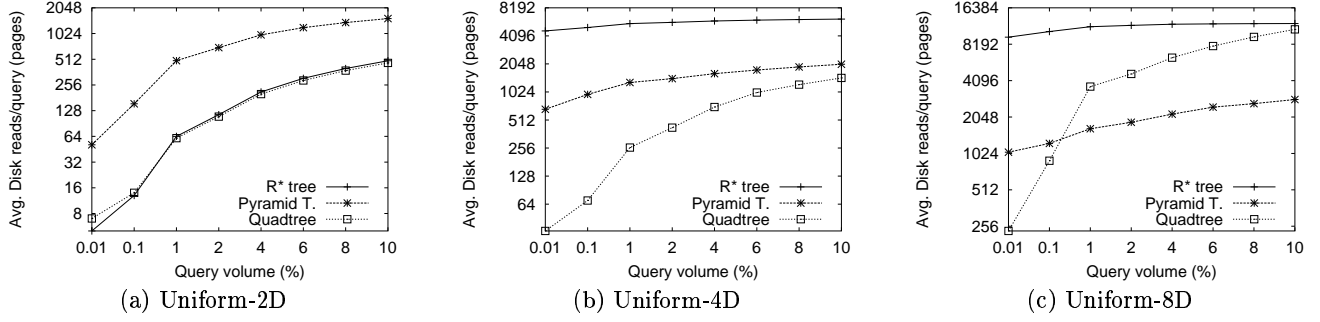
Figure 1: Analytical model prediction of disk accesses with uniformly distributed synthetic datasets

Finally, as in the case of the R*-tree, the expected number of page accesses for the packed Quadtree is calculated as:

$$PA = \sum_{i=1}^{P} DA(i)$$

3.3 Pyramid-Technique

The Pyramid-Technique [3] is based on an unbalanced space partitioning: The data space is split into $2d$ pyramids and each pyramid is further divided into several partitions, each of which corresponds to a page in the B^+-tree.

To index a d-dimensional point, the Pyramid-Technique transforms the point into a 1-dimensional pyramid value and indexes the pyramid value in the B^+-tree. The transformed pyramid value, p, is computed as: $p = v + h$, where v is an integer in the range of $[0, 2d - 1]$, representing the pyramid that encloses the d-dimensional point. h is a real number in the range $[0,0.5]$. h represents the height of the point inside the pyramid v from the center (at value 0.5) for the v or the $(v$-$d)$-th dimension.

A range query in the Pyramid-Technique is processed as follows: First, each of the $2d$ pyramids is checked to determine if it intersects with the query Q. If there is a point inside Q with a pyramid value in the range $[v, v{+}0.5]$, the pyramid v intersects with Q. Next, for each intersecting pyramid, the range of the pyramid values affected by Q, $[h_{low}, h_{high}]$, is determined. Finally, for each intersecting pyramid v, the B^+-tree is searched with the range query $[v{+}h_{low}, v{+}h_{high}]$. For more details on the Pyramid-Technique, please see [3].

Similar to the analytical formulas for the R*-tree and the Quadtree, given a B^+-tree index created using the Pyramid-Technique, we can derive a formula that estimates the expected number of page accesses. To compute this estimate, we need information from the B^+-tree regarding the range of pyramid values in a page i. Let the page i have pyramid values in the range $[p_{i,l}, p_{i,h}]$, where $p_{i,l} = v_{i,l} + h_{i,l}$ and $p_{i,h} = v_{i,h} + h_{i,h}$. Then, the probability of accessing the page i, $DA(i)$, is proportional to the range of the pyramid values enlarged by the size of the $v_{i,l}$-dimensional value of Q.

More formally, let x denote a dimension in Q that can affect the page i.

$$x = \begin{cases} v_{i,l} & \text{if } v_{i,l} < d \\ v_{i,l} - d & \text{if } v_{i,l} \geq d \end{cases}$$

Then, $DA(i)$ is defined as:

Indexes	Average estimation errors		
	2D	4D	8D
R*-tree	6.2%	15.7%	7.5%
Quadtree	4.9%	14.7%	4.4%
Pyramid T.	20.1%	2.7%	1.7%

$DA(i) =$

$$\begin{cases} \dfrac{\min(|h_{i,h} - 0.5|, 1 - q_x) - \max(|h_{i,l} - 0.5| - q_x, 0)}{1 - q_x} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } v_{i,l} < d \\[2ex] \dfrac{\min(|h_{i,h} + 0.5|, 1 - q_x) - \max(|h_{i,l} + 0.5| - q_x, 0)}{1 - q_x} \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } v_{i,l} \geq d \end{cases}$$

Note that the absolute values of $h_{i,h}$ and $h_{i,l}$ shifted by ± 0.5 are the values in the original data space $[0..1]$. Also, the minimum and maximum are used to remove the parts that exceed the data space. In addition, in contrast to the formulas for the R*-tree and Quadtree, note that 1-q_x is used as a denominator instead of $\prod_{j=1}^{d}(1 - q_j)$. This is because the probability of accessing a page i, $DA(i)$, is only affected by the range of values in a single dimension.

Finally, the overall expected number of page accesses is calculated as:

$$PA = \sum_{i=1}^{P} DA(i)$$

3.4 Comparison

In this section, we test the accuracy of the analytical models. For this test, we use 2, 4, and 8 dimensional uniform datasets with 2 million points, and compare the actual number of page accesses with the analytical predications. For this test, we used 1024 queries that are uniformly distributed and completely inside the data space. Since our analytical model does not capture the effects of using a buffer pool, for this comparison, we simply flushed the buffer pool after processing each query.

The Table 2 shows the average estimation errors of the prediction compared to actual measurements. As can be seen in this table, the analytical model is fairly accurate for

Table 3: Index construction time (minutes)

Dataset	R*tree	Pyramid-T.	Quadtree
Uniform-2D	90.8	2.0	3.6
Uniform-4D	61.6	2.0	3.7
Uniform-8D	69.8	11.8	14.0
F.Cover-2D	21.8	1.4	0.9
F.Cover-4D	20.8	1.3	0.8
F.Cover-8D	19.1	1.8	2.4
MAPS-2D	92.8	19.1	3.3
MAPS-4D	70.4	11.7	3.5
MAPS-8D	75.7	12.9	10.1

Table 4: Index size (pages)

Dataset	R*tree	Pyramid-T.	Quadtree	File
Uniform-2D	4,267	6,950	4,124	5,918
Uniform-4D	6,822	8,934	7,789	7,906
Uniform-8D	12,422	12,994	13,603	11,835
F.Cover-2D	1,268	2,020	1,413	1,719
F.Cover-4D	2,027	2,597	2,114	2,297
F.Cover-8D	3,592	3,776	4,423	3,483
MAPS-2D	5,387	7,058	7,352	6,009
MAPS-4D	7,468	9,072	13,170	8,027
MAPS-8D	13,221	13,194	16,079	12,017

all the index types.

The predictions of the analytical models are shown in Figures 1 (a), 1 (b), and 1 (c), for the 2, 4, and 8 dimensional datasets respectively. For each figure, these figures plot the predicted number of page accesses per query for increasing query sizes. As can be seen from these figures, the analytical models predict that the Quadtree will outperform the R*-tree in all cases. It also shows that the Quadtree will outperform the Pyramid-Technique on relatively low dimensional data and with small window queries in higher dimensions. In addition, the models predict that the Pyramid-Technique will outperform the Quadtree for high dimensional data with large window queries. As shown in the next section, these predictions agree well with the actual experimental results, which adds confidence that the experimental results are not due to implementation differences but rather due to intrinsic characteristics of index structures.

4. EXPERIMENTS

In this section, we present experimental results comparing R*-tree, Pyramid-Technique, Quadtree indices and a sequential file scan implemented in SHORE [5]. (The sequential file scan serves as a baseline). SHORE was configured to use 8KB page size, and in all experiments except the one in which we evaluate the effect of buffer pool size, the SHORE buffer pool size was set to 8MB. This small buffer pool size allows us to clearly see the effect of IOs. Note that we also present some results with settings when the indices are completely resident in memory. All experiments were performed on a machine with 2GHz Intel Xeon processor running Red Hat Linux version 2.4.20.

The Quadtree was implemented as described in Section 2, and pages were only packed to 70% of their capacity to mimic the scenario in which addition space is left to accommodate future index updates.

For our experiments, we used both synthetic and real datasets. For the synthetic dataset, we used a dataset containing 2 million uniformly distributed points in 2, 4, and 8 dimensional data space (labeled as Uniform-2D, Uniform-4D, and Uniform-8D respectively). In addition, we used the following two real datasets: the Forest Cover data containing the forest cover type for 30 x 30 meter cells from US Forest Service Region 2 Resource Information System (http://kdd.ics.uci.edu/databases/covertype/covertype.html), and the MAPS Catalog data containing photometric and astrometric information extracted from the Palomar Observatory Sky Survey (http://iparrizar.stcloudstate.edu/~juan/MAPS_Database/).

The Forest Cover dataset contains 581,014 entries with 54

attributes. Of these 54 attributes, there are 10 quantitative attributes. To measure the performance with varying dimensionality, we ran Principle Component Analysis (PCA) on the dataset over the 10 quantitative attributes to generate 2, 4, and 8 dimensional datasets, which are labeled as F.Cover-2D, F.Cover-4D, F.Cover-8D, respectively. The use of PCA reduces the dimensionality of data without a large loss of information, and this method is used to mimic the use of dimensionality reduction methods in many high-dimensional applications.

The MAPS dataset contains about 90 million objects with 39 attributes. To keep our experiments manageable, we used the first 2 million objects. To provide some variety, we did not use PCA on this dataset. Instead, to produce data with varying dimensionality, we used the first 2, 4, and 8 attributes from the original dataset to get 2, 4, and 8 dimensional datasets, which are labeled as MAPS-2D, MAPS-4D, MAPS-8D, respectively.

For our query workload, we generated hypercube shaped range queries with varying query volumes. Query volume is defined as the percentage of the query hypercube volume over the data space volume. For the synthetic datasets, queries are uniformly randomly positioned in the underlying data space. For real data, we used both uniform random queries, and skewed queries that follows the distribution of the underlying data. In all experiments, for each point shown on the graph we have a corresponding query workload of 1024 queries and we report the average per query execution times and the average per query disk page accesses.

4.1 Index size and creation times

Table 4 shows index sizes for 2, 4, and 8 dimensional datasets. This table indicates that on average Quadtrees are about 22% larger than R*-tree and Pyramid-Technique indices, while R*-tree and Pyramid-Technique indices require almost the same amount of disk space. This is because Quadtrees have more non-leaf and leaf nodes than the other indices, and there is an extra storage overhead for saving meta-information for each node.

Table 3 shows index creation times. For index creation, first multiple inserts are used for creating the R*-trees and Quadtrees. Then, the index is traversed to produced a disk-efficient layout. The Pyramid-Technique uses the native SHORE B^+-tree bulkloading mechanism. The main observation here is that our naive bulkloading method for the Quadtree compares very favorably compared to the Pyramid-Technique. (Our R*-tree bulkloading is quite naive and a true bulkloading method is likely to be significantly faster.)
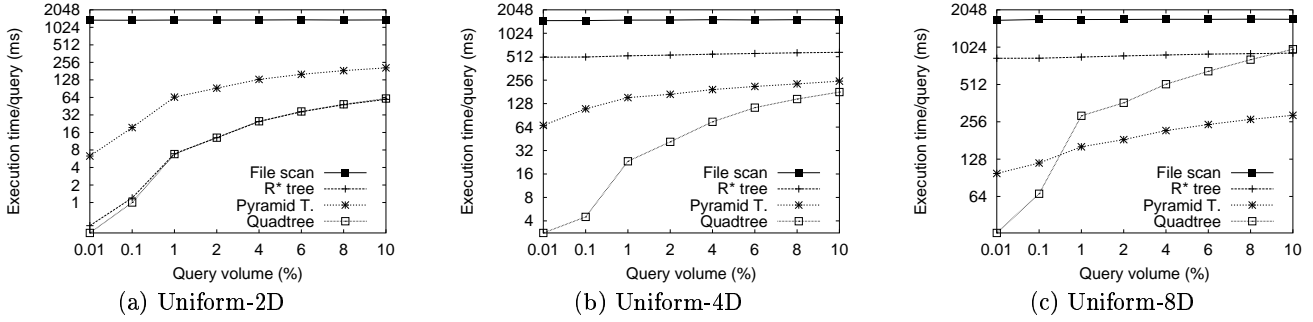
(a) Uniform-2D   (b) Uniform-4D   (c) Uniform-8D

Figure 2: Total execution time for varying data dimensionality with uniform synthetic datasets

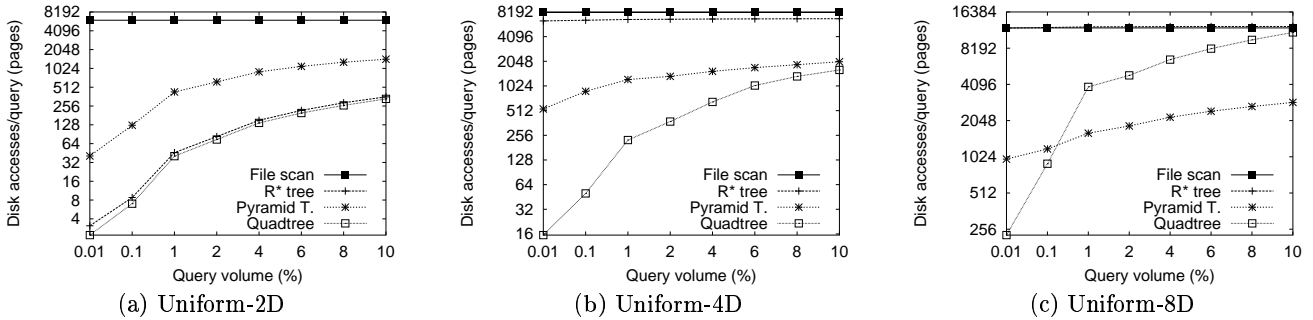(a) Uniform-2D   (b) Uniform-4D   (c) Uniform-8D

Figure 3: Number of disk page accesses for varying data dimensionality with uniform synthetic datasets

4.2 Evaluation with synthetic datasets

In this section, we compare the performance of indices with uniform data. Figures 2 and 3 present average query execution times and disk accesses for query volumes ranging from 0.01% to 10% for 2D, 4D, and 8D datasets. As shown in these Figures, the query execution times are proportional to the number of disk accesses for all three indices.

With Uniform-2D, the Quadtree and R*-tree show almost the same performance although the number of disk accesses for the Quadtree is slightly less than that of the R*-tree. In addition, both indices outperform the Pyramid-Technique by 3-21 times in all queries.

With Uniform-4D, the Quadtree performs the best while the R*-tree performs the worst among three indices. The Quadtree still outperforms the Pyramid-Technique by 1.5 to 8 times in all queries, although the Quadtree speedup over the Pyramid-Technique decreases with increasing query volumes. The reason for the poor performance of the R*-tree is due to the increased overlap amongst MBRs with increasing dimensionality, which results in multiple path traversals during query processing.

For the Uniform-8D dataset, the Quadtree still outperforms the Pyramid-Technique for small window queries, but the Pyramid-Technique begins to outperform the Quadtree for large window queries. In fact, the Pyramid-Technique does better than the Quadtree on high dimensional data with large window queries.

Figure 3 shows that with small window queries the number of disk accesses with the Pyramid-Technique is significantly higher than that with the Quadtree. This figure also shows that the number of disk accesses incurred by the

Pyramid-Technique increases more gradually compared to the Quadtree. This is due to the *unbalanced* space split strategy used in the Pyramid-Technique. Compared to the balanced space split used in the Quadtree, the unbalanced space split incurs more page accesses for small window queries, especially when window queries are further away from the center of data space. However, the unbalanced space split incurs less page accesses than the balanced space split for large window queries.

In summary, based on this experiment with uniform datasets, we make the following observations:

1. The Quadtree index outperforms the R*-tree in all cases, although the performance of these two index structures is comparable in 2 dimension.

2. The Quadtree outperforms the Pyramid-Technique with relatively low dimensional data and with small window queries in higher dimensions.

3. The Pyramid-Technique outperforms the Quadtree in high dimensional data with large window queries.

Finally, we note that the analytical results in Figure 1 shows slightly higher number of page accesses than the one in Figure 3. This is because Figure 3 shows the number of page accesses using a buffer pool, while Figure 1 shows the number of page accesses without any buffering.

4.3 Evaluation with real datasets

In this section, we compare the performance of the three indices using real datasets. For this experiment, we used skewed queries that follow the underlying data distribution,
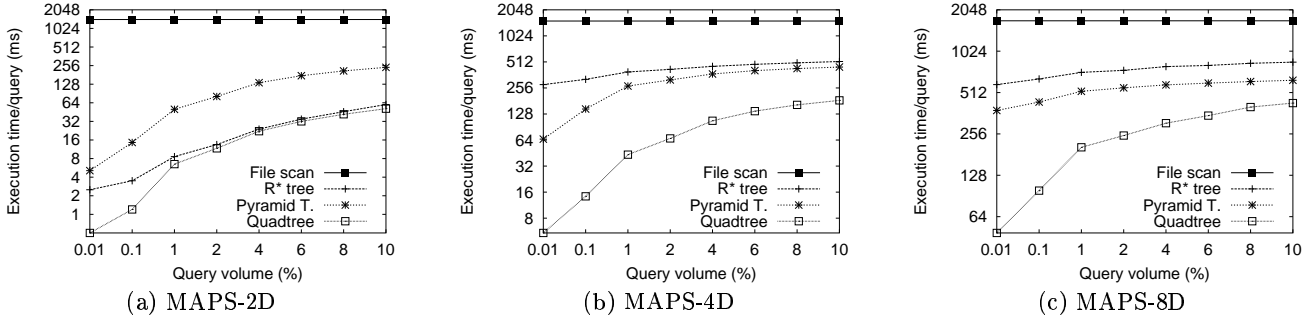
(a) MAPS-2D   (b) MAPS-4D   (c) MAPS-8D

Figure 4: Total execution time for varying data dimensionality with the MAPS Catalog dataset

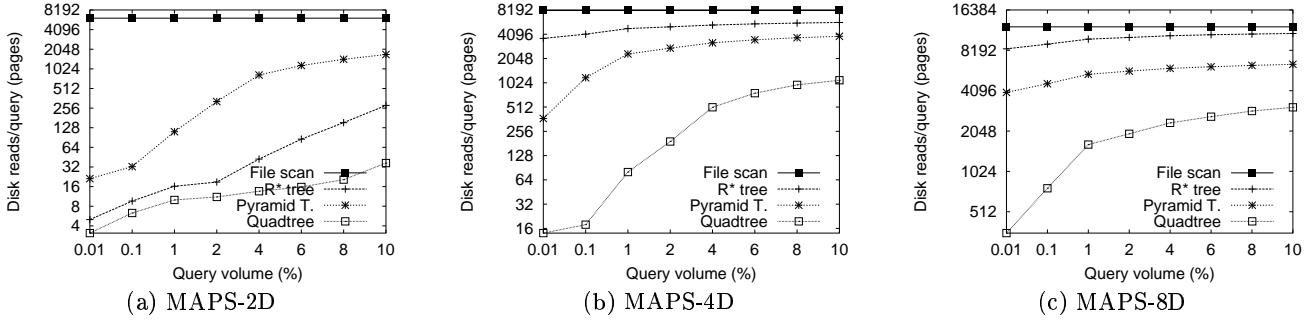(a) MAPS-2D   (b) MAPS-4D   (c) MAPS-8D

Figure 5: Number of disk page accesses for varying data dimensionality with the MAPS Catalog dataset

since such queries are more representative of actual queries with real datasets. (We also present results with random queries in Section 4.5.) In addition, we used the extended Pyramid-Technique designed for handling skewed data [3]. Using the extended Pyramid-Technique for skewed data improves query performance by up to 59% compared to using the naive Pyramid-Technique.

Figures 4 and 6 compare average query execution times, and Figures 5 and 7 compare disk accesses for the three index structures on the real datasets. As shown in these Figures, Quadtrees significantly outperform R*-trees and Pyramid-Technique indices in all cases. Furthermore, compared to the results with uniform data, Quadtrees outperform Pyramid-Technique indices even for the 8 dimensional datasets, and the Quadtree speedup over other indices is more significant. For example, Quadtrees are faster than Pyramid-Technique indices by 5–11, 2–12, 2–7 times on the MAPS-2D, MAPS-4D, and MAPS-8D datasets respectively, and by 3–10, 5–29, and 3–68 times on the F.Cover-2D, F.Cover-4D, and F-Cover-8D datasets respectively.

One reason for the higher Quadtree speedup is the relatively poor performance of the other indices when handling skewed data. For the R*-tree, since there is high overlap amongst MBRs and skewed points can be spread under several non-leaf nodes, the R*-tree suffers from traversing multiple paths. For example, the average number of overlapping MBRs per a random point query is 54, 2845, and 4353 with the MAPS-2D, MAPS-4D, and MAPS-8D datasets respectively, which constitute 1%, 38%, 34% of the total MBRs in the R*-trees respectively.

For the Pyramid-Technique, its unbalanced space split strategy can be adversarial for skewed data. Figure 8 il-
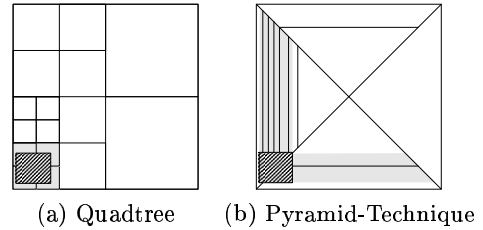

(a) Quadtree   (b) Pyramid-Technique

Figure 8: Interaction between query and partitioning boundaries

lustrates this point with an example. This figure shows the Quadtree and the corresponding Pyramid-Technique index built on a skewed 2 dimensional dataset which is clustered in the bottom-left region. Each square in the Quadtree and each region in the Pyramid-Technique corresponds to a leaf page. A striped rectangle represents a query window, and gray regions are the ones affected by the window query. As shown in this figure, with a skewed query, there are more data regions that intersect with regions in the Pyramid-Technique compared to quads in the Quadtree, which in turns leads to a larger number of page accesses for the Pyramid-Technique.

To experimentally confirm the adversarial effect of the unbalanced space split for skewed data, we measured the *filtration ratio*, which is defined as the number of points that are *not* contained within the window query over the number of points retrieved in accessed pages. As more false hits are retrieved, the filtration ratio increases. With a 10% query volume and the MAPS-8D dataset, the filtration ratio of the
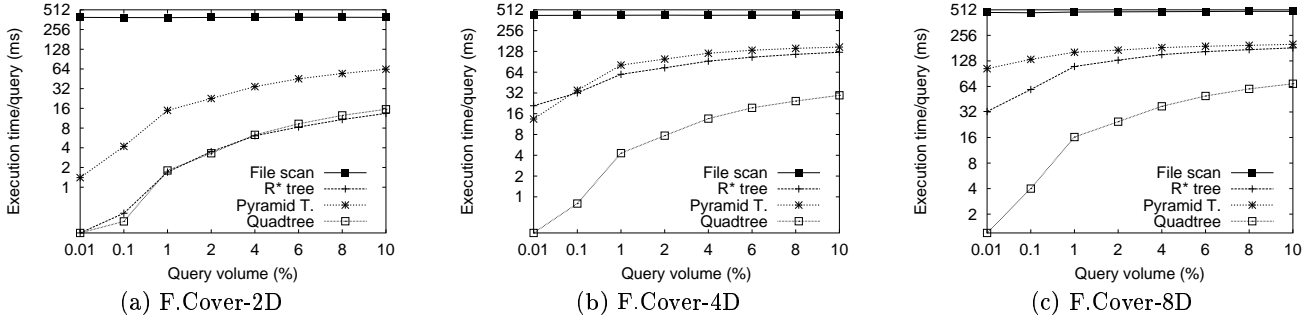
(a) F.Cover-2D   (b) F.Cover-4D   (c) F.Cover-8D

Figure 6: Total execution time for varying data dimensionality with the Forest Cover dataset

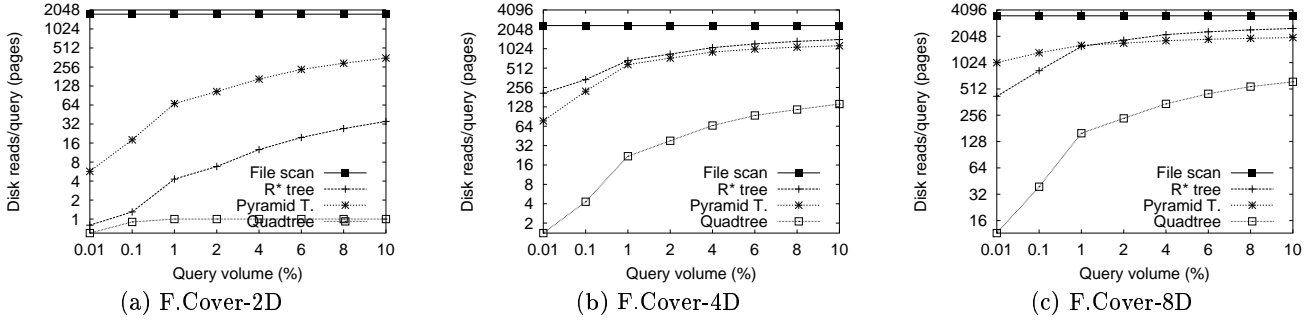(a) F.Cover-2D   (b) F.Cover-4D   (c) F.Cover-8D

Figure 7: Number of disk page accesses for varying data dimensionality with the Forest Cover dataset

Quadtree using a balanced space split strategy is 26.1%. For this same setting, the filtration ratio of the Pyramid Technique using the unbalanced space split strategy is 71.5%.

Another reason for the better performance of the Quadtree with skewed data is that the Quadtree uses the buffer pool more effectively. The explanation for this behavior is as follows: When indexing skewed data, index structures such as the R*-tree and the Pyramid-Technique build a balanced tree structure, often distributing even highly clustered points into multiple paths in order to maintain the height-balanced requirements. On the other hand, the Quadtree builds an unbalanced tree structure, which reflects the distribution of the underlying dataset. When processing skewed queries which follow the underlying data distribution (realistic types of queries in real applications), the difference in these index structures leads to very different buffer pool usage behaviors.

In a balanced index, non-leaf nodes close to a root node are more likely to be cached in the buffer pool. This is because many clustered points are spread under several non-leaf nodes, thus different parts of the index are likely to be traversed, accessing non-leaf nodes more often. During query processing, fewer disk IOs are incurred for accessing non-leaf nodes, but since many leaf nodes are not resident in the buffer, disk IOs are frequently incurred for accessing leaf nodes.

On the other hand, in an unbalanced index such as the Quadtree, when processing skewed queries, the dense and deeper index structure is traversed more frequently than other regions of the index. As a result, non-leaf and leaf pages in the dense region are likely to be cached in the buffer pool. For skewed queries that are likely to traverse

Table 5: Speedup with 128MB over 8MB buffer pool

Dataset	R*-tree	Pyramid-T.	Quadtree	File
Uniform-8D	2.7–2.9	1.6–1.7	**2.2–3.3**	1.4–1.5
MAPS-8D	2.7–3.2	1.8	**1.4–1.6**	1.5

the dense regions repeatedly, fewer disk IOs are incurred. In other words, the Quadtree has *better spatial locality* than the balanced index structures and this results in efficient buffer management, improving overall performance.

To validate this observation, we measured the performance speedup of the three index structures with a large buffer pool. For this experiment, we set the SHORE buffer pool size to 128MB, which is large enough to cache each index entirely. Then we calculated the speedup over an 8MB buffer pool size. The result of this experiment are presented in Table 5. These results represent the range of speedups with all query volumes (0.01%–10%).

As shown in Table 5, in all cases, a large buffer size benefits all indices, since disk IOs are eliminated. However, more importantly, we observe that the speedups for the Quadtrees change significantly depending on the dataset characteristics, whereas the speedups for the R*-tree and Pyramid-Technique indices almost remain constant regardless of the dataset characteristics. Specifically, the larger buffer pool has a relatively smaller impact on the Quadtrees with the skewed dataset. The reason for this behavior is because for skewed data the Quadtrees already effectively caches many of the frequently accessed pages with a small buffer pool. Thus, the increased buffer pool has a relatively smaller impact on the number of disk accesses.
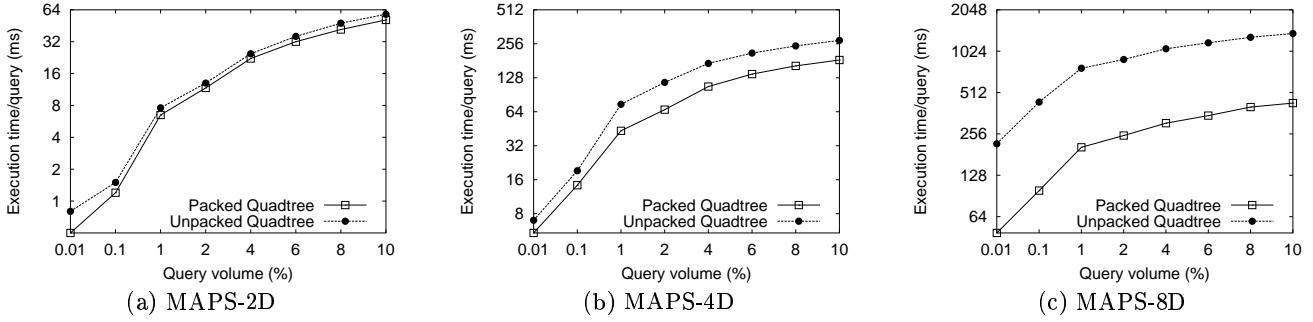
288

(a) MAPS-2D   (b) MAPS-4D   (c) MAPS-8D

Figure 9: Total execution time of packed and unpacked Quadtree with the MAPS Catalog dataset

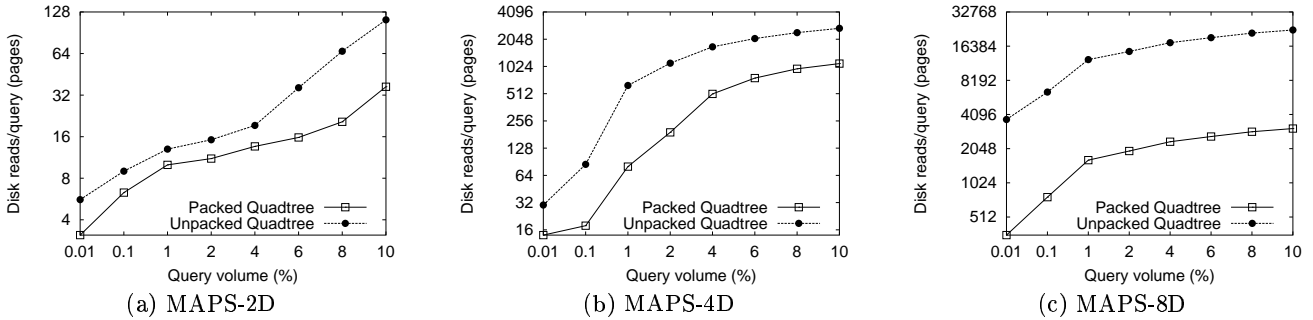(a) MAPS-2D   (b) MAPS-4D   (c) MAPS-8D

Figure 10: Number of disk page accesses of packed and unpacked Quadtree with the MAPS Catalog dataset

Related to the results in Table 5, we note that the Quadtree still outperform the R*-tree and the Pyramid-Technique with a large buffer pool for all the datasets. For example, with MAPS-8D, the Quadtree speedup over the R*-tree ranges from 1.2–5.1 times and the Quadtree speedup over the Pyramid-Technique ranges from 1.3–6.1.

In summary, we make the following observations for real datasets:

1. The Quadtree outperforms the other indices for the range of dimensions from 2 to 8. However, the relative performance improvements are smaller as the dimensionality increases and as the query volume increases.

2. The higher performance of the Quadtree is due to the relatively poor performance of the R*-tree and Pyramid-Technique in handling skewed data, and a better buffering behavior of the Quadtree with skewed data.

4.4 The effect of packing on the Quadtree

In this section, we investigate the effect of the packing technique used for the Quadtree. To examine this effect, we compared index sizes and query processing times for the Quadtrees with and without using the packing technique. The Quadtree without packing is the initial Quadtree built using a page for a leaf node. The Quadtree with packing is the final Quadtree generated after rewriting the initial index using the packing technique described in Section 2.

With MAPS-4D, the Quadtree without packing uses 13,335 pages and results in 39% leaf node occupancy. The Quadtree with packing uses 7,352 pages and has a 74% leaf node occupancy. The packing technique reduces index size by 45%

and improves execution times by 1.5–3.7X. We note that the Quadtree without packing still outperforms the R*-tree and Pyramid Technique indices.

With MAPS-8D, the Quadtree with no packing uses 69,442 pages with 14% leaf node occupancy. The Quadtree with packing uses 16,079 pages with 74% leaf node occupancy. The packing technique reduces an index size by 77% and improves execution times by 3.3–8.7X. We note that in this case the Quadtree performance without packing can be worse than the R*-tree and the Pyramid-Technique.

Figures 9 and 10 compare average query execution time and disk accesses for the packed and unpacked Quadtrees. As seen from these figures, the packing technique is an influential factor in achieving the high Quadtree speedup over the other index structures. Without the packing technique, the Quadtree might not outperform the R*-tree and the Pyramid-Technique in some cases. In addition, the effect of the packing technique is especially evident for high dimensional data due to a higher reduction factor in index size. The key to the improved performance of the packing technique comes from the reduced disk accesses due to the smaller index size.

4.5 Effect of query distribution

For the experiments in Section 4.3, we used queries that followed the underlying data distribution. In this section, we evaluate the effect of the query distribution and consider the effect of using random queries on the real datasets. Figure 11 compares the effect of query types on the MAPS-8D dataset. Random queries are uniformly distributed in the data space and skewed queries are the ones that follow the distribution of the underlying data (as in Section 4.3).
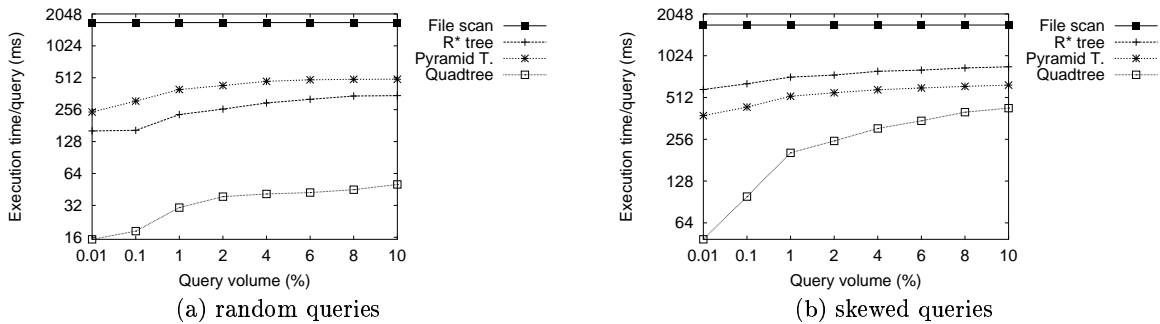
289

Figure 11: Random vs. Skewed queries with MAPS-8D

As shown in this figure, with random queries, the Quadtree speedup over the other indices is even more significant than with skewed queries. While skewed queries traverse deep and denser regions of an index, in most cases, many random queries are likely to traverse sparse and low-depth regions of the index. The traversal of those sparse and low-depth regions costs much less and lowers the average execution time for the Quadtree. In balanced index structures such as the R*-tree and the Pyramid Technique, random queries that traverse sparse regions still require accessing at least as many nodes as the height of the index. Consequently, the performance of these balanced index structures is worse compared to the Quadtree.

5. CONCLUSIONS

In this paper, we have compared the performance of the R*-tree, the Pyramid-Technique, and the Quadtree for indexing low to medium dimensional datasets (datasets with less than eight dimensions). Although the R*-tree is widely used for indexing such datasets, our results show that the Quadtree significantly outperforms the R*-tree and the Pyramid-Technique, and that the speedup of the Quadtree over these other indices is more significant for skewed datasets. Based on experimental results, we conclude that the efficiency of the Quadtree comes from: 1) the packing technique, 2) the regular and disjoint partitioning method used by the Quadtree, and 3) the better spatial locality leading to a more efficient utilization of the buffer pool.

As part of our work, we also developed analytical models for the performance of these three indices. Our analytical results are in agreement with the empirical results, and add confidence that the claims that we make are not due to implementation differences or dataset peculiarities.

As part of future work, we plan on expanding our investigation to consider more complex operations, such as the k nearest neighbor and the distance join operations.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search In Sequence Databases. In *FODO*, pages 69–84, 1993.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles[+]. In *SIGMOD*, pages 322–331, 1990.

[3] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyrmaid-Technique: Towards Breaking the Curse of Dimensionality. In *SIGMOD*, pages 142–153, 1998.

[4] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, 1996.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring Up Persistent Applications. In *SIGMOD*, pages 383–444, 1994.

[6] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *J. Intell. Inf. Syst.*, 3:231–262, 1994.

[7] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *PODS*, pages 4–13, 1994.

[8] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1997.

[9] I. Gargantini. An effective way to represent quadtrees. *CACM*, 25(12):905–910, 1982.

[10] A. Guttman. R-trees: a Dyanmic Index Structure for Spatial Indexing. In *SIGMOD*, pages 44–57, 1984.

[11] G. R. Hjaltason and H. Samet. Speeding up construction of pmr quadtree-based spatial indexes. In *VLDB Journal*, pages 109–137, 2002.

[12] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. In *VLDB*, pages 500–509, 1994.

[13] R. Kothuri, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparision using GIS Data. In *SIGMOD*, pages 546–556, 2002.

[14] K.-I. Lin, H. Jagadish, and C. Faloutsos. The TV-tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3:517–542, 1994.

[15] Y. S. Moon, K. Y. Whang, and W. S. Han. General Match: A Subsequence Matching Method in

Time-Series Databases Based on Generalized Windows. In *SIGMOD*, pages 382–393, 2002.

[16] Y. S. Moon, K. Y. Whang, and W. K. Loh. Duality-Based Subsequence Matching in Timeseries Databases. In *ICDE*, pages 263–272, 2001.

[17] W. Niblack, R. Barber, W. Equitz, M. D. Flickner, E. H. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC Project: Query Images by Content Using Color Texture, and Shape. In *SPIE*, pages 173–187, 1993.

[18] H. Samet. The Quadtree and Related Hierarchical Data Structures. *Computing Surveys*, 16(2):187–260, 1984.

[19] N. Weskamp, D. Kuhn, E. Hüllermeier, and G. Klebe. Efficient similarity search in protein structure database by k-clique hashing. *Bioinformatics*, 20(10):1522–1526, 2004.