Modeling High-Dimensional Index Structures using Sampling*

Christian A. Lang
Department of Computer Science
University of California
Santa Barbara, CA 93106
clang@cs.ucsb.edu

Ambuj K. Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
ambuj@cs.ucsb.edu

ABSTRACT

A large number of index structures for high-dimensional data have been proposed previously. In order to tune and compare such index structures, it is vital to have efficient cost prediction techniques for these structures. Previous techniques either assume uniformity of the data or are not applicable to high-dimensional data. We propose the use of sampling to predict the number of accessed index pages during a query execution. Sampling is independent of the dimensionality and preserves clusters which is important for representing skewed data. We present a general model for estimating the index page layout using sampling and show how to compensate for errors. We then give an implementation of our model under restricted memory assumptions and show that it performs well even under these constraints. Errors are minimal and the overall prediction time is up to two orders of magnitude below the time for building and probing the full index without sampling.

1. INTRODUCTION

A large number of index structures for high-dimensional data have been proposed in previous years [3, 7, 15, 20, 35, 26, 14, 24, 8, 6, 33]. These index structures have different characteristics and optimization strategies, making their analysis difficult. The indexed high-dimensional datasets are also difficult to analyze because of their non-trivial distribution in high dimensions. As a result of this complexity, the development of models to predict the performance of index structures has been slow and difficult. Such models are important because they allow us to analyze strengths and weaknesses of index structures. Another application area is the tuning of an index structure. A fast prediction model would allow us to evaluate different parameters (like splitting strategy, dimensions stored in the index, or page sizes) without really having to build the index on disk. Once the best parameter combination is determined, the full index can be built.

More generally, the problem we try to solve can be described as follows:

Given an indexing scheme, a query workload, and a dataset, what is the average number of index page accesses required to execute the queries?

We will concentrate on data-partitioning indexing schemes (especially the VAMSplit R*-tree [34]) and nearest neighbor (NN) queries in this paper. However, our work can also be applied to range queries and other indexing schemes.

We claim that performance prediction of high-dimensional index structures has to take specific properties of the dataset and the index structure into account. Previous approaches model dataset and indexing scheme too simplistically (e.g. by assuming data uniformity and squared index pages) and as a result, fail in a high-dimensional setting. We believe that sampling is the right approach when predicting high-dimensional index structures. The prediction model we present in this paper first samples the data, builds a miniature index in memory, and then uses this index to predict the performance of the original index structure. Since the index pages contain fewer points due to sampling, they shrink slightly. We develop formulas to compensate for this shrinkage.

Since we use a data sample, the real data distribution is captured quite well. Since we build the miniature index with the same algorithm used for the on-disk index, the indexing scheme is captured quite well. However, the smaller the main memory available, the smaller the sample has to be in order to avoid costly I/O operations. At the same time, the prediction accuracy decreases with decreasing sample size.

In the second part of the paper, we therefore show how memory restrictions can be overcome by building the miniature index in several phases. This can be seen as a compromise between a low-quality but fast prediction of a memory-based approach, and a high-quality but slow prediction of a disk-based approach. We present two techniques for predicting the original index pages in phases. For each technique, we develop cost formulas to estimate the I/O cost incurred for different memory sizes, dataset sizes, and dimensionalities. We show that both techniques are between one and two orders of magnitude faster than measuring the page accesses with a disk-based index. The experimental results show that our model provides high prediction accuracy while at the same time reducing the prediction cost from

*Work partially supported by grants CCR-9972571, IIS-9877142 from the National Science Foundation and by UC MICRO award 00-191.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

hours to seconds. Our comparison with prediction techniques based on the uniformity assumption and the fractal dimensionality shows that these approaches are not suited for high-dimensional data. To the best of our knowledge, there is no other technique that gives useful performance predictions for high-dimensional real datasets.

The paper is organized as follows. In Section 2 we present some recent work on query cost prediction, and different approaches of modeling the data. Section 3 discusses our basic prediction model without memory restrictions. Section 4 shows how restricted memory can be addressed and Section 5 gives some experimental results based on real high-dimensional datasets. Section 6 discusses some applications that show the usefulness of our technique. We end with some concluding remarks in Section 7.

2. RELATED WORK

This section discusses previous work in the area of performance prediction of index structures. We will organize this discussion according to the way the techniques model the data. The first set of approaches assumes uniformity of the data, the second set models the data using global parameters, the third set models the data more fine-grainedly with local parameters, and the fourth set uses a data sample as a model. We will see that these approaches progressively model a larger number of data distributions more accurately.

2.1 Uniform Data Models

Most of the work assuming uniformity concentrates on R-tree-like index structures. The basic idea is to first estimate the average shape of an index page and the average query region, and then calculate the average number of index pages intersected by the queries. The first known analysis of R-trees was given by Faloutsos et al. [13] but was restricted to one-dimensional data. In a later paper, Kamel and Faloutsos [19] present a cost model using a concept similar to Minkowski sums to predict the number of disk accesses for two-dimensional data. Arya et al. [2] give a detailed analysis of NN queries for bucketing and k-d-tree index structures including boundary effects. However, they assume that the number of data points grows exponentially with the dimensionality which may not hold for real datasets. Berchtold et al. [4] present a cost model for 1-NN queries in high-dimensional space. They estimate the page layout by assuming that the space is recursively split in the middle (due to the uniformity assumption) until the right number of pages is reached. The query shape is estimated by equating the probability that a neighboring point is within the NN-sphere of the query point with the volume of this sphere. The page prediction is computed again by using Minkowski sums. Weber et al. [33] give a similar analysis and extend the model for k-NN queries and more general page geometries.

The advantage of the above models is that they are simple, require only very few parameters (like the page capacity), and are relatively fast to compute. The disadvantage is that the uniformity assumption does not hold for real and high-dimensional data and leads to bad prediction results in these cases. This shortcoming led to parametric models discussed next.

2.2 Globally Parametric Models

These models try to account for the non-uniformity of the data by modeling it using a few global parameters, like the fractal dimensionality. Faloutsos et al. [12] presented the first cost model for R-trees based on the fractal dimensionality which is claimed to be the "inherent dimensionality" of a dataset. This first model was restricted to range queries but later work by Papadopoulos and Manolopoulos [28] extended it for 1-NN queries in R-trees. Korn et al. [22] give a version for k-NN queries. Common to all these approaches is that they estimate the average page geometry assuming square pages (which may not hold in high dimensions) and the average query shape in a similar way as the uniform techniques. However, instead of considering the embedding dimensionality of the dataset, they consider the fractal dimensionality which can be estimated by an $O(N \log N)$ algorithm.

The advantage of this technique is that it can model data better than the simple uniformity assumption and that it is moderately fast. A disadvantage is its restriction to lower dimensions. Experiments on high-dimensional real datasets showed that the fractal dimensionality tends to be close to 0, leading to an overestimation of the page accesses.

2.3 Locally Parametric Models

This group of techniques models a dataset by first splitting it into regions and then describing each region with a few parameters (e.g. its density). The regions can be created by data or space partitioning. The work by Ciaccia et al. [10] is an example of this technique for data partitioning. They extend the M-tree nodes by statistics (the distance distribution) in order to predict the CPU and I/O cost of range and NN queries. In case of space partitioning, this technique results essentially in histograms over the dataset. Theodoridis and Sellis [31] give a model for predicting the performance of range queries on R-trees. They generate a *density surface* which is basically a two-dimensional histogram representing the local densities of the dataset. Acharya et al. [1] show how the prediction error of histograms can be reduced by reducing the variance of densities within the histogram regions. Korn et al. [21] demonstrate how discontinuities at the histogram region edges can be avoided by using splines. Finally, Jin et al. [18] extend the histogram approach for spatial non-point data.

The advantage of these techniques is their increased accuracy in modeling data by considering local effects. However, this comes at the price of a higher storage requirement. Another disadvantage of the histogram approaches is that they are not applicable in high dimensions since either the number of histogram regions becomes too large, or these regions contain too much empty space and become inaccurate.

2.4 Sampling-based Models

All techniques discussed so far showed problems when modeling real data or when this data was high-dimensional. A very simple but effective technique that can handle high-dimensional non-uniform data is sampling. The basic idea is to pick a subset of the real dataset, examine the index behavior on this subset, and extrapolate from the subset behavior to the behavior on the original dataset. Previous work based on sampling focussed only on query estimation for relational models. Lipton et al. [25] show how to estimate selectivity by sampling using confidence intervals. Haas et al. [16] give a similar analysis for join operations. The limitations of random sampling for estimating the number of distinct values in a column of a table are discussed by Charikar et al. [9]. We are not aware of any previous work utilizing sampling to estimate the cost of query operations in spatial databases.

The advantage of sampling is that it is simple and independent of the dimensionality. Furthermore, it preserves clusters, and so it can be used to model non-uniform data. A disadvantage seems to be the high cost introduced by acquiring a large data sample, or the low accuracy when a small data sample is used.

The rest of this paper will show that it is possible to find a compromise between these two extremes that leads to low prediction costs at a high accuracy. We claim that sampling is a very effective tool to predict the performance of index structures for high-dimensional data.

3. SAMPLING-BASED PREDICTION MODEL

In this section, we show how sampling can be used to predict the cost of range and nearest neighbor queries. This discussion assumes an unlimited amount of memory. The next section will deal with the limited memory case.

3.1 The Basic Model

The basic idea of our model is to predict the leaf page geometry by building a miniature version of the index structure. This is achieved by first sampling the dataset, then building the index structure (efficiently in memory) on the sample, and finally predicting the query cost using the leaf pages of this "mini-index" and the query workload. For the sake of prediction accuracy, it is important to ensure that the mini-index has the same overall structure as the full index. This structure comprises the number of nodes at each level, the fanout at each node, and the height of the tree. By using the same tree construction algorithm for the mini-index and the full index, we can achieve this structural similarity. Of course, this assumes knowledge of the index structure examined. In the following, we focus on R-tree-like structures, more specifically VAMSplit R*-trees [34]. However, the ideas presented could be applied to other structures as well.

Since the mini-index is built on a subset of the dataset, we have to reduce the page capacity accordingly. For example, if we use as a sample 1/10 of the original data, each page of the mini-index will contain on average 1/10 of the original points. Or in other words, the page capacity is reduced by the factor 1/10. Since R-trees organize points in minimal bounding boxes, the reduced number of points per box will cause the box to shrink. This shows that even though we have structural similarity, the page layouts of the full index and the mini-index are not necessarily equal.

Figure 1 depicts the connection between the original and the sample page layout. Figure 1(a) shows the original dataset; Figure 1(b) depicts the index structure constructed on the original dataset; Figure 1(c) shows the sampled dataset; Figure 1(d) depicts the mini-index constructed on the sampled dataset. As seen in the example, the pages of the mini-index shrink on account of sampling. Therefore, we have to grow them all by a *compensation factor* to make the resulting page layout similar to the original index. Once the miniature index is established, we predict the expected I/O cost by counting the number of intersections between expected query regions and leaf pages of the mini-index.

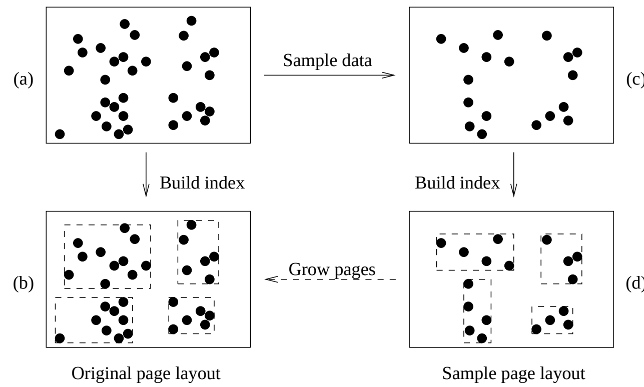


Figure 1: Connection between original and sample page layout

Two questions arise from the preceding discussion: what is the right compensation factor for growing pages of the mini-index? How does the sample size affect the performance and the prediction quality? We will answer these questions in the following subsections.

3.2 Calculation of the Compensation Factor

To allow for a mathematical analysis, let us assume that the points within the page are distributed uniformly[1]. Then the problem we want to solve can be stated as follows:

Given a set of C uniformly distributed points in d dimensions. Let B be the minimal bounding hyperrectangle of these points.
How does the volume of B change when the number of points is reduced from C to $C \cdot \zeta$ where $0 < \zeta < 1$?

The answer gives the following

THEOREM 1. *Under uniformity assumption, the volume of each index leaf page changes by*

$$\delta(C,\zeta)^{-1} = \left(\frac{(C\cdot\zeta - 1)(C+1)}{(C\cdot\zeta + 1)(C-1)}\right)^d$$

where C is the page capacity, ζ is the sampling fraction, and d is the dimensionality of the dataset.

PROOF. see [23]. □

3.3 Effect of Sample Size

Determining the appropriate sampling ratio (or the sample size) for a given dataset can be difficult. Large samples may not fit entirely into memory, and the resulting disk I/Os will result in higher prediction costs. Too small samples, on the other hand, will cause higher prediction errors. This can be seen in Figure 2, where we ran 500 21-NN queries on the COLOR64 dataset (cf. Table 1) and compared the actual page accesses with the predicted page accesses for different sample sizes. As expected, the prediction becomes more accurate when compensating for the page shrinkage. However, for sample sizes of less than 10%, the error becomes too large

[1]Note that this is very different from assuming uniformity throughout the whole dataspace since each page is created based on the (non-uniform) data sample. Only within a page, we assume uniformity.

COLOR64	112,361 64-dimensional points representing color histograms computed from a commercial CD-ROM (transformed using KLT).
TEXTURE48	26,697 48-dimensional points representing texture feature vectors obtained from the Corel Image Collection (transformed using KLT).
TEXTURE60	275,465 60-dimensional points representing texture feature vectors of Landsat images (transformed using KLT).
ISOLET617	7,800 617-dimensional points representing features such as spectral coefficients, contour features, and sonorant features obtained by recording the 52 letters of the alphabet as spoken by 150 speakers.
STOCK360	6,500 360-dimensional points representing the price of 6,500 stocks over one year (transformed using DFT).

Table 1: Datasets used in the experiments

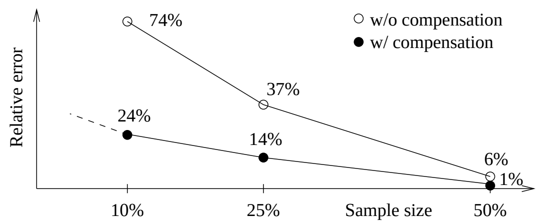


Figure 2: Relative error for different sample sizes

N	number of data points
M	number of points that fit in memory
B	number of points per disk page
$height$	height of the on-disk index tree
h_{upper}	height of the upper tree[2]
h_{lower}	height of the lower trees
σ_{upper}	sampling ratio for the upper tree
σ_{lower}	sampling ratio for the lower trees
$C_{max,data}$	maximum data page capacity (on-disk index)
$C_{max,dir}$	maximum directory page capacity (on-disk index)
$C_{eff,data}$	effective data page capacity (on-disk index)
$C_{eff,dir}$	effective directory page capacity (on-disk index)
t_{seek}	average seek and latency time of the disk
t_{xfer}	transfer time for a single 8 KByte disk page

Table 2: Notation used in the paper

for even this technique to be useful. Another constraint on the sample size is due to the page occupancy of the mini-index. Since a page of the mini-index must hold at least one point, the sample rate can never be smaller than $1/C$, where C is the page capacity of the full index.

Next, we present a compromise between the extremes of building the whole mini-index in memory (which is fast but induces a high error) and building the whole index on disk (which is slow but has no error). We will show how the mini-index can be built in phases, thereby allowing for a fast prediction (up to two orders of magnitude faster than building the index on disk), and at the same time ensuring high accuracy (typically less than 5% relative error).

4. IMPLEMENTATION UNDER RESTRICTED MEMORY ASSUMPTIONS

In this section we show how the sampling technique of the previous section can be implemented under restricted memory assumptions. The basic idea is to make use of the available memory in order to reduce the amount of disk I/O during the prediction. This is achieved by building the mini-index in a piecemeal fashion. We show how the technique can be easily adjusted to reach different tradeoffs between disk I/O and prediction accuracy.

The following sections discuss the new approach in greater depth and give an analysis of its I/O cost. This discussion is preceded by the description of a bulk-loading algorithm for R-trees which serves as a basis for our comparisons. We will refer to this approach as *on-disk index tree*. Both approaches have a main memory size of M data items available.

Throughout this discussion we will use the symbols listed in Table 2.

4.1 The On-Disk Index Tree

The on-disk index tree is an index structure generated by the bulk loading algorithm described by Berchtold et al. [5]. This algorithm recursively partitions the dataspace to generate the index tree pages level-wise. For each level, the required fanout is determined and the data is partitioned according to some splitting strategy. Each partition is then further subpartitioned in the same fashion until the leaf level is reached. The pseudo code of this algorithm can be found in the full version of this paper [23] (for a more detailed description, cf. [5]).

As a splitting strategy we chose the *maximum variance-split*. This results in a layout of pages similar to the VAM-Split R*-tree [34]. We will use this on-disk index as a basis for comparing the I/O costs of the different approaches, since it is always possible to simply build an index on disk via bulk loading and then run some sample queries on it in order to measure their cost. The I/O cost of this approach comprises the cost for building the index and the cost for running selected queries on it.

Let $cost_{BuildTreeLevel}(level, start, end)$ denote the cost for building a single index subtree with height $level$ storing the data points numbered $start$ through end (this numbering can be obtained from the location of the data points on disk). The cost of the whole on-disk bulk loading algorithm is then

$$cost_{OnDisk} = cost_{BuildTreeLevel}(height, 0, N). \qquad (1)$$

We give a recursive definition of the latter cost function in [23]. Note that the data partitioning is based on the *find* algorithm introduced by Hoare [17]. The complexity of this algorithm depends on the data distribution. In the worst case, its complexity is $O(N^2)$, and in the best case it is $O(N)$. In our cost formula we therefore assume the best

[2]Note that an empty tree has height 0, and a tree consisting of only one node has height 1. Leaf nodes are at level 1 and the root node is at level $height$.

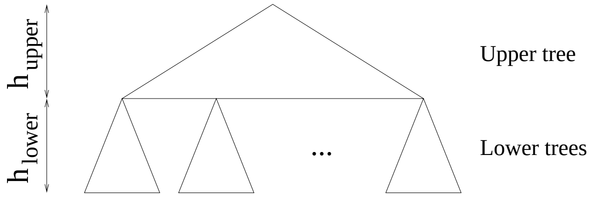


Figure 3: Partitioning of the index tree

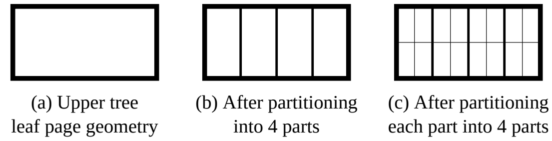


Figure 4: Splitting of a page

case which is in favor of the on-disk index. Our experiments will show that the cost on real data is five to ten times higher.

4.2 Predicting in Phases

Because of the high I/O cost for building the on-disk index, it is not a good choice for estimating the index performance. In this section, we give a compromise between our sampling approach in Section 3 (not accurate enough under memory restrictions) and the on-disk approach (not fast enough under memory restrictions). We use the same bulkloading algorithm as the on-disk index but apply it to a sample of the data. In order to increase the sample rate, we build the index tree in several phases with each phase using the whole memory. Therefore, we split the index tree into two parts: an upper part and a lower part. The upper part is a single tree (denoted by *upper tree*) and the lower part is a forest of subtrees (denoted by *lower trees*, cf. Figure 3). The subtrees are constructed one-by-one in a piecemeal fashion thereby allowing each subtree to use the whole memory. Since the upper tree has much fewer leaf pages than the whole tree, this results in a higher upper tree leaf page occupancy. This occupancy depends on the sampling rate and the height of the upper tree. Clearly, we want to choose the largest possible sample that still fits into memory. The sampling rate σ_{upper} for the upper tree is therefore chosen to be

$$\sigma_{upper} = \min(\frac{M}{N}, 1).$$

Note also that a small upper tree leads to few but large lower trees, whereas a large upper tree leads to many but small lower trees. This becomes an important issue when choosing the right value for h_{upper}. We will return to this issue in Section 4.5.

After obtaining the leaf pages of the upper tree, these are grown by the compensation factor $\delta(pts(height - h_{upper} + 1), \sigma_{upper})$ (cf. Section 3), where $pts(h)$ denotes the number of points in a subtree with root in height h. For example, $pts(height) = N$ and $pts(1) = C_{eff,data}$. These values can be calculated by taking the fanout at each level of the tree into account. More details can be found in the full version of this paper [23].

In order to build the lower trees, we examined two approaches. The first approach, which we will call *cutoff index tree*, predicts each lower subtree only by looking at the geometry of the grown leaves of the upper tree and assuming uniformity, thus not causing additional I/O. The second approach, denoted by *resampled index tree*, constructs each of the lower trees one after the other in a piecemeal fashion using the same algorithm as for the upper tree. Since for this approach, additional data points have to be inspected, additional disk I/O is required. However, as we will see later, this cost is well below the cost for building the entire index on disk.

Once the lower trees have been estimated (using one of the above two techniques), the number of intersections between the query regions and the lower tree leaf pages are computed in order to calculate the average number of leaf page accesses. The determination of the query regions (the *workload*) depends on the type of query to be examined. In the following, we will concentrate on density-biased k-NN queries. For this type of queries, the k-NN spheres have to be determined for some number of query points. Since we consider biased queries, we have to place more queries in regions of higher density. This can be achieved by first randomly reading some query points from the dataset and then performing a full scan of the data to compute the query shapes (this scan is necessary for the first sampling step anyway). Since we read the sample query points randomly from the dataset, this results in a cost of

$$cost_{ReadQueryPoints} = q \cdot (t_{seek} + t_{xfer}), \qquad (2)$$

where q denotes the number of query points.

Alternatively, the search radii could be obtained from the sample. First experiments on the COLOR64 dataset showed that the search radius does not seem to be affected much by the sample ratio. In our experiments, however, we used the full dataset to compute the search radii.

In the following subsections, we discuss the two variants for building the lower trees.

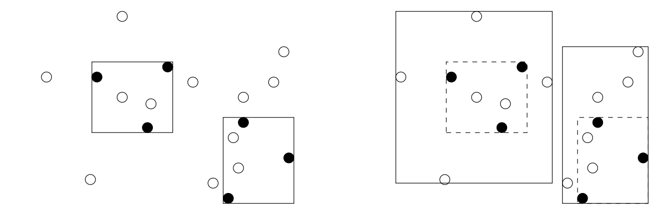
4.3 The Cutoff Index Tree

In order to estimate the structure of the lower trees, the cutoff approach takes only the geometry of the upper tree leaf pages and knowledge about the original bulk loading algorithm into account. To construct the lower tree beginning at a certain upper tree leaf page, we assume uniform data distribution inside this page and perform the page splits the original bulk loading algorithm would perform. Assume the page geometry in Figure 4 (a). If the points within the page are distributed uniformly, the dimension with the highest variance will be the dimension with the largest page extent. So, if we perform consecutive maximum variance splits, we will get the new page layout depicted in Figure 4 (b) (assuming a fanout of 4). The same procedure is repeated for the next levels until the leaf level is reached.

Note that we assume uniformity only within a page. Moreover, we know already the approximate upper tree leaf page geometries from building the upper tree. Finally, we use the fanout calculation and splitting strategy of the on-disk index to obtain the page layouts. In contrast, Berchtold et al. [4] assume uniformity in the whole dataspace. Furthermore, they cannot base their predictions on intermediate page geometries. Finally, they assume that a dimension is simply split in the middle in order to obtain the leaf page layout. Therefore, our approach results in more accurate page ge-

```
(1) Determine the tree topology;
(2) Read q query points randomly from dataset;
(3) Scan the whole dataset to determine the query spheres
(4)     and to read a sample of size M into memory;
(5) Build the upper tree for the sample;
(6) For each upper tree leaf page p
(7)     Build the lower tree starting at p by inspecting
           p's geometry;
(8)     Count the intersections between query spheres and
           the leaves of the lower tree;
(9) Report the average number of leaf page intersections
        per query;
```

Figure 5: The cutoff tree prediction algorithm



(a) Upper tree leaf pages before growing (b) Upper tree leaf pages after growing

● First sample ○ Second sample

Figure 6: Growing pages by resampling

ometry predictions. However, as we will see in the experiments, even when making all these assumptions, the prediction accuracy deteriorates for high dimensional datasets. This leads to the conclusion that a purely uniformity-based prediction model is not useful in high-dimensional space.

In order to establish a cost formula for this approach, we state the whole algorithm in pseudo code in Figure 5. Note that only steps 2 and 3 introduce I/O cost. The scanning in step 3 causes a cost of

$$cost_{ScanDataset} = t_{seek} + \left\lceil \frac{N}{B} \right\rceil \cdot t_{xfer}.$$

Therefore, the overall cost of this approach is

$$cost_{Cutoff} = cost_{ReadQueryPoints} + cost_{ScanDataset}. \quad (3)$$

4.4 The Resampled Index Tree

The resampled index tree samples additional data points for each upper tree leaf page and uses them to build the corresponding lower tree. The idea is to sample M points per lower tree, resulting in a higher sampling rate than for the upper tree. For example, if we have 50 upper tree leaf pages, we would have only $M/50$ points per lower tree. If we would increase the sample size 50 times, each lower tree would have approximately M points. The effective sampling rate for the lower trees is then 50-fold, resulting in more accurate predictions. Once the resampling is done, we can build each lower tree one at a time on M points (using the full memory for each of them) and counting again the number of leaf page intersections.

However, one problem remains: how can we resample the additional points per page? Or more precisely: how can we read the additional points and assign them to the lower trees efficiently? Since the upper tree leaf pages were obtained using a small sample, the page extents are expected to be smaller than the original pages at the same level of the tree[3]. This means that some of the newly sampled points will not fall into any page. Additionally, a naïve solution would require as many dataset scans as there are upper tree leaf pages which is obviously prohibitive.

The solution we present in the following requires each point in the sample to be read twice and written once, and is otherwise independent of the number of upper tree leaf pages. Consider the example in Figure 6 (a). The solid dots mark data points in the upper tree sample. The boxes around these points are the upper tree leaf pages. Let us assume we have three such upper tree leaf pages (only two of which are shown here) and therefore want to increase the sampling rate for each lower tree by a factor of 3. By sampling the dataset with this new sampling rate, we read the points marked by white dots. If a new point falls into an existing box, it is included in it. Otherwise, we include this point in the closest box according to the Euclidean distance and adjust the box accordingly. The resulting box layouts are depicted in Figure 6 (b). Since all sampling is performed uniformly, we can expect each page to contain on average three times as many points as before.

In the general case, we have k upper tree leaf pages[4]. To obtain M points per lower tree, we have to sample $k \cdot M$ points. The lower tree sampling rate is therefore

$$\sigma_{lower} = \min(\frac{k \cdot M}{N}, 1).$$

By scanning the dataset once, we can determine each sample point and — since we still have the upper tree leaf pages in memory — the lower tree to which it is assigned. However, since we cannot store all $k \cdot M$ points in memory, we store all points assigned to one box in a consecutive area on disk in order to reduce the number of random seeks. That means, we need as many such consecutive areas as we have upper tree leaf pages. In a second phase, we read one such consecutive area into memory at a time and construct the corresponding lower tree the same way we constructed the upper subtree. Since each consecutive area contains approximately M points, we make full use of the available memory[5].

In order to establish a cost formula for this approach, we state the algorithm in pseudo code in Figure 7. Steps 2 and 3 cause the same cost as for the cutoff index tree. Step 6 is performed as follows (cf. Figure 8). We read $N \cdot \sigma_{lower}$ points from the original data file into memory. Since we have a memory size of M, we read these sample points in chunks of size M (step a). Therefore, the number of chunks is $\lceil \frac{N}{M} \cdot \sigma_{lower} \rceil$. To read one chunk, we have to scan over M/σ_{lower} points, resulting in a cost of $t_{seek} + \left\lceil \frac{M}{B \cdot \sigma_{lower}} \right\rceil \cdot t_{xfer}$ per chunk for the loading step.

For each chunk, the M points have to be distributed among the k boxes. To reduce random seeks, the points are sorted by the box they fall into and then all points of one box are written to disk in one chunk of size M/k on average (step b). Since we have k such block writes and M/B

[3]This is true even after growing the pages with our compensation factor δ if the data is non-uniform.

[4]In [23], we give a formula for determining the fanout at each tree level.

[5]If an area contains more than M points, we discard the additional points in our current implementation.

```
 (1) Determine the tree topology;
 (2) Read q query points randomly from dataset;
 (3) Scan the whole dataset to determine the query spheres
 (4)    and to read a sample of size M into memory;
 (5) Build the upper tree for the sample, let k be the
        number of upper tree leaf pages;
 (6) Scan the whole dataset to determine k*M data points
 (7)    and write each of them in one of k consecutive disk
           areas depending on which box it falls into;
 (8) For each upper tree leaf page p
 (9)    Read the points from the corresponding disk area
           into memory;
(10)    Build the lower tree starting at p by inspecting
           the resampled points;
(11)    Count the intersections between query spheres and
           the leaves of the lower tree;
(12) Report the average number of leaf page intersections
        per query;
```

Figure 7: The resampled tree prediction algorithm

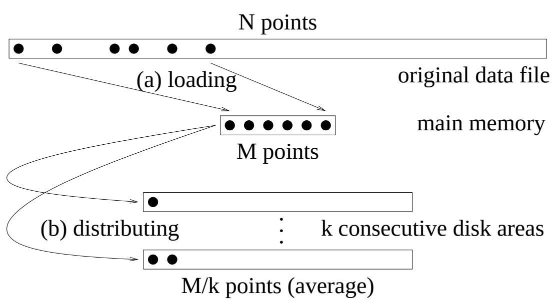


Figure 8: Read and write operations during the resampling step

blocks are transfered from memory to disk, this distributing step costs $k \cdot t_{seek} + \lceil \frac{M}{B} \rceil \cdot t_{xfer}$.

The whole step 6 therefore costs

$$cost_{Resampling} = \left\lceil \frac{N}{M} \cdot \sigma_{lower} \right\rceil \cdot \left(t_{seek} + \left\lceil \frac{M}{B \cdot \sigma_{lower}} \right\rceil \cdot t_{xfer} + k \cdot t_{seek} + \left\lceil \frac{M}{B} \right\rceil \cdot t_{xfer}\right). \quad (4)$$

The loop in line 8 reads k chunks of size approximately M into memory, resulting in a cost of

$$cost_{BuildLowerSubtrees} = k \cdot \left(t_{seek} + \left\lceil \frac{M}{B} \right\rceil \cdot t_{xfer}\right).$$

Building the lower tree in line 10 is done entirely in memory with the bulk loading algorithm discussed in Section 4.1.

The overall cost of the resampled index tree is therefore

$$\begin{aligned} cost_{Resampled} = \; & cost_{ReadQueryPoints} + \\ & cost_{ScanDataset} + \\ & cost_{Resampling} + \\ & cost_{BuildLowerSubtrees}. \end{aligned} \quad (5)$$

4.5 The Choice of h_{upper}

Constraints on the choice of h_{upper} arise due to two reasons: first, due to ensuring a minimum capacity of 2 for the leaf level pages (in order to get a non-trivial page), and second, due to prediction error considerations. We discuss both issues in the following sections and show how the I/O cost is affected by the choice of h_{upper}.

4.5.1 Constraints due to Leaf Page Capacity

Each upper (or lower) tree contains M points. The taller such a tree is, the less number of points are stored in the leaf nodes (due to the constant branching factor). Since each leaf page needs to store at least 2 points (otherwise it would have no volume), this implies an upper bound on the height of a tree. If the height of the upper tree is too small, the lower trees are too large causing sparsely populated lower tree leaf pages. On the other hand, if it is too large, the leaf pages of the upper tree are too sparse.

4.5.1.1 Lower Bound on h_{upper}.

In order to ensure that a lower tree in the resampled index stores at least 2 points in its leaf pages, we can equivalently ensure that a tree of full height built on $N \cdot \sigma_{lower}$ $(= k \cdot M)$ points (this corresponds to the situation of the resampled index), stores at least 2 points in its leaf pages. Therefore, we get the following lower bound on the height of the upper tree:

$$h_{min,upper} = \min_{2 \le h \le height-1} \{h : capacity(height, 2, N \cdot \sigma_{lower}, 0) \ge 2\},$$

where $capacity(h, level, items, 0)$ defines the number of data points contained in a subtree starting at level $level - 1$ in a tree of height h containing $items$ data items. It is defined in [23]. Note further that σ_{lower} depends on h.

In case of the cutoff index, only the upper tree is constructed using data points. Therefore, the height of the lower trees is arbitrary and we have no lower bound on the height of the upper tree.

4.5.1.2 Upper Bound on h_{upper}.

In order to ensure that the capacity of the upper tree leaf pages is at least 2, we can equivalently ensure that a tree of full height built on $N \cdot \sigma_{upper}$ $(= M)$ points stores at least 2 points in its pages in level $height - h_{upper} + 1$. Therefore, the upper bound on the height of the upper tree is

$$h_{max,upper} = \max_{2 \le h \le height-1} \{h : capacity(height, height - h + 2, N \cdot \sigma_{upper}, 0) \ge 2\}.$$

Depending on the number of data items, the dimensionality, and the memory size, these bounds may still leave several values to choose from.

4.5.2 Constraints due to Prediction Error

If h_{upper} is small, the lower trees contain many points, and therefore σ_{lower} has to be small as well in order to accommodate all points of one lower tree in memory. This in turn leads to too small lower tree leaf pages and therefore to an underestimation of the page accesses. On the other hand, if h_{upper} is large, there will be a large number of sparse upper tree leaf pages. Since the geometry of these leaf pages will likely deviate a lot from the original pages, many points distributed later based on distance will end up in "wrong" pages. This leads to many overlaps between the upper tree leaf pages, and therefore an overestimation of the page accesses. According to our experiments, the best choice for h_{upper} seems to be when the unsampled size of the lower trees becomes M.[6]

4.5.3 Effect of h_{upper} on I/O Cost

The further up in the tree the resampling is performed (smaller h_{upper}), the fewer lower trees have to be built, resulting in lower disk I/O. For larger h_{upper}, the disk I/O will be higher. This can be seen from Equation 4, where the amount of seeks is

$$\left\lceil \frac{N}{M} \cdot \sigma_{lower} \right\rceil \cdot (1 + k).$$

With increasing h_{upper}, more points can be sampled for the lower trees and therefore σ_{lower} increases. At the same time, k increases leading to an overall increase of the amount of seeks.

The amount of page transfers from Equation 4 is

$$\left\lceil \frac{N}{M} \cdot \sigma_{lower} \right\rceil \cdot \left(\left\lceil \frac{M}{B \cdot \sigma_{lower}} \right\rceil + \left\lceil \frac{M}{B} \right\rceil \right),$$

which is roughly

$$\frac{N}{B} + \frac{N}{B} \cdot \sigma_{lower}.$$

As explained above, with increasing h_{upper}, the sampling ratio σ_{lower} increases, causing an overall increase in the amount of page transfers.

Summarizing, for small h_{upper} values, the I/O cost is small but the prediction error is high due to ill-defined lower tree leaf pages. When h_{upper} increases, the I/O cost increases but the prediction error decreases until it reaches a minimum (approximately when the lower trees contain M points). After this point, both the I/O cost and prediction error increase. The latter is due to increasingly ill-defined upper tree leaf pages.

4.6 I/O Cost of the Predictions

In this section, we will analytically compare the I/O cost of all approaches presented in the previous sections for different dataset and memory sizes and data dimensionalities. This analysis is based on the cost formulas developed earlier. Later in Section 5, we will present experimental results on real datasets that validate these analytical results. As we will see there, the gap in I/O cost between the on-disk algorithm and our prediction schemes is in fact much larger than that predicted analytically.

For the theoretical analysis, we chose $t_{seek} = 10$ ms and $t_{xfer} = 0.4$ ms[7]. Figure 9 compares the I/O cost of all three approaches for different memory sizes M. We assume a dataset with one million points in 60 dimensions. Note that the y-axis is plotted on a log scale. For the resampled approach, we chose h_{upper} so that the unsampled size of the lower trees is approximately M (cf. Section 4.5.2). This explains the jumps in the graph. The I/O cost of the cutoff approach is independent of the choice of h_{upper}. As

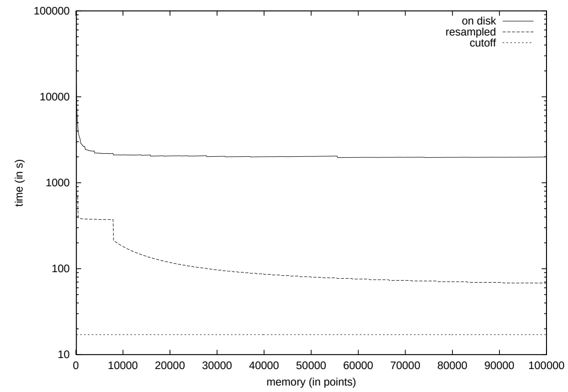


Figure 9: I/O cost for different memory sizes M

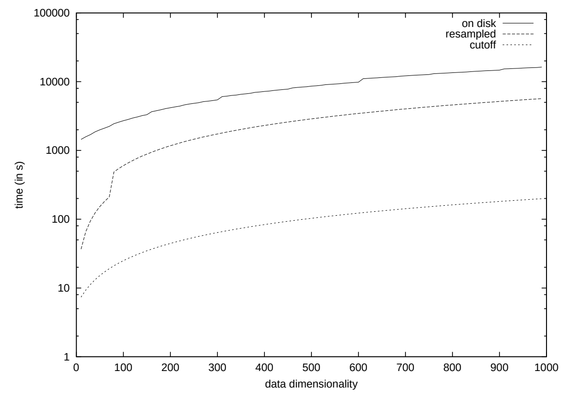


Figure 10: I/O cost for different data dimensionalities

expected, in all cases the I/O cost decreases monotonically with increasing memory size. However, the resampled approach is always one order of magnitude faster than the on-disk approach and the cutoff approach is up to two orders of magnitude faster.

The graph in Figure 10 compares the I/O cost for different data dimensionalities. Here, we fixed the dataset size at one million points and chose the memory size as $M = \frac{600{,}000}{dim}$ since the number of points we can store in memory depends inversely on the dimensionality. By using this formula, M will be 10,000 again for 60 dimensions.

The I/O cost depends linearly on the dimensionality in all three approaches. The jump at about 80 dimensions in the resampled approach is again due to a different choice of h_{upper} to achieve a better prediction. If we kept the same upper tree height, the resampling approach would stay about 10 times faster than the on-disk approach at the cost of a worse prediction quality. The cutoff approach is about 100 times faster for all dimensionalities.

A similar observation can be made when varying the dataset size. The resampled approach is again one order of magnitude faster than the on-disk approach and the cutoff approach is two orders of magnitude faster. Instead of hours, the new approaches take minutes or seconds.

It is important to note that in the case of the on-disk index we take only the I/O cost for building the index into account. The cost for running sample queries on the index can add a significant amount of I/O as we will show in our

[6]For smaller sizes, not the whole memory is used and the prediction quality is lower. For larger sizes, sampling has to be applied, leading again to a lower prediction quality.

[7]We assume a hard disk with an average seek (plus latency) time of 10 ms, and a disk bandwidth of 20 MB/s giving an average transfer time of 0.4 ms for 8 KByte pages.

experiments. Furthermore, as mentioned earlier, we underestimate the cost for the on-disk approach since we assume perfect splits during the data partitioning. This may not hold for real data. Section 5 presents the measured I/O cost and prediction accuracy for real datasets.

4.7 Other Index Structures

Our prediction technique is not restricted to the VAM-Split R*-tree. It is in fact applicable to a large group of index structures. This group comprises all index structures that organize the data in fixed-capacity pages with a given storage utilization. Prominent members of this group are the R-tree [15] and its variants (R^+-tree, R^*-tree), the X-tree [7], the SS-tree [35], the SR-tree [20], the M-tree [11], the k-d-B-tree [29], and the grid file [27].

An example for an index structure not contained in this group is the VA-file [32], since it does not organize points in pages of fixed capacity.

5. EXPERIMENTAL RESULTS

In order to get an impression of the real running times of our algorithms, we implemented all of them and counted the actual number of seeks and data transfers. Our experiments were conducted on the datasets listed in Table 1. For each of them, we built an index on disk using the bulk loading algorithm and ran 500 21-NN sample queries to calculate the average number of leaf page accesses per query. This is compared to our prediction techniques which also use 500 sample queries. The following discussion focuses on the TEXTURE60 dataset but similar observations can be made for the other datasets. Detailed results can be found in [23].

The height of the index tree in the TEXTURE60 example is 5. Table 3 lists the results for $M = 10{,}000$. The results for $M = 1{,}000$ look very similar and we will omit them due to space restrictions. The first column indicates the prediction method used along with the parameters. The relative error is measured based on the on-disk index as the ground truth. Negative numbers are underestimations, positive numbers are overestimations. The page seeks and page transfers columns list the total number of disk seeks (caused by reading a page not adjacent to the previously read page) and 8K-page transfers respectively. Note that the sum in the on-disk costs stands for "*building cost + query cost*". The I/O cost is again obtained assuming a disk with a seek time of 10 ms and a transfer rate of 20 MB/s.

5.1 Runtime of the Predictions

As expected from our cost formulas, the cutoff approach is the fastest, followed by the resampled approach. In this setting, the resampled approach was between 25 and 318 times faster than the on-disk approach. For the cutoff index, the speedup is even higher: between 525 and 548 times improvement. Another interesting observation is that the ratio between the seeks and page transfers for the on-disk queries is close to 1 meaning that nearly all page accesses during queries were random.

5.2 Accuracy of the Predictions

In this section we examine the accuracy of the predictions compared to the on-disk index. Table 3 lists the relative errors as a signed number to indicate under and overestimations. As discussed in Section 4.5 for the resampled approach, we underestimate the number of page accesses for small upper trees and overestimate it for large upper trees. At the point when σ_{lower} becomes 1 (resp. when the lower trees contain M points), the relative error is less than 5% for both $M = 1{,}000$ and $M = 10{,}000$[8]. The cutoff approach yielded relative errors below 25% at a much lower I/O cost. As we will see, the consistency of the cutoff predictions is much worse however.

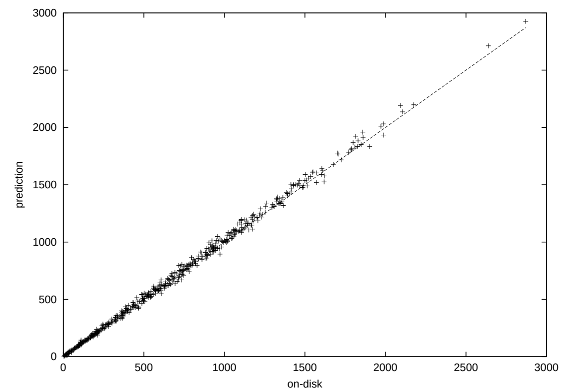


Figure 11: Correlation diagram for resampled index (TEXTURE60 dataset, M = 10,000 and $h_{upper} = 3$)

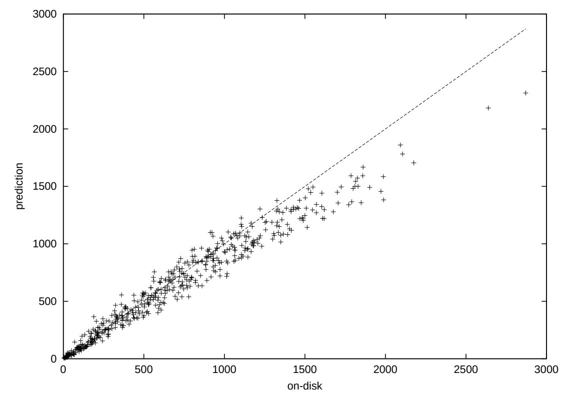


Figure 12: Correlation diagram for resampled index (TEXTURE60 dataset, M = 1,000 and $h_{upper} = 4$)

In order to examine the consistency in prediction quality, we use a correlation diagram which correlates predicted and estimated number of page accesses for each of the 500 sample queries. In the ideal case, all points would lie on the main diagonal meaning that measurement and prediction match perfectly every time. Figure 11 shows how well measurement and prediction are correlated for the resampled approach with $M = 10{,}000$ and $h_{upper} = 3$. The corresponding diagram for the cutoff approach showed no correlation at all. This demonstrates that the overall relative error alone is not a good enough measure for the quality of a prediction. Figure 12 shows the resampled approach with $M = 1{,}000$ and $h_{upper} = 4$. As we can see from this diagram, the amount of correlation decreases slightly with decreasing memory.

Since we assume uniformity of the data when inflating

[8] The fact that there is any error at all when the lower sample ratio approaches 100% can be explained by the upper sample ratio which is still well below 100%.

Method	Rel. error	Page seeks	Page transfers	I/O cost (in s)
On-disk	0%	61,798 + 350,152	500,232 + 351,500	4,460.193
Resampled ($h_{upper} = 2$, $\sigma_{upper} = 0.0363$, $\sigma_{lower} = 0.1089$)	−32%	688	18,528	14.291
Resampled ($h_{upper} = 3$, $\sigma_{upper} = 0.0363$, $\sigma_{lower} = 1$)	+3%	1,035	33,805	23.872
Resampled ($h_{upper} = 4$, $\sigma_{upper} = 0.0363$, $\sigma_{lower} = 1$)	+17%	5,071	37,244	65.608
Cutoff ($h_{upper} = 2$, $\sigma_{upper} = 0.0363$)	−64%	501	8,705	8.492
Cutoff ($h_{upper} = 3$, $\sigma_{upper} = 0.0363$)	−27%	501	8,705	8.492
Cutoff ($h_{upper} = 4$, $\sigma_{upper} = 0.0363$)	−16%	501	8,705	8.492

Table 3: Relative error and I/O cost for M = 10,000 (TEXTURE60 dataset)

the upper tree leaf pages and during the page splits in the cutoff approach, we examined how well the predictions turn out for uniform data. We generated a dataset of 100,000 uniformly distributed points in 8 dimensions[9] and ran the same experiments as before (the index tree has a height of 3 in this example). The relative errors for the resampled and the cutoff approach were between −0.5% and −3%, thus confirming the assumption of our model.

5.3 Comparison with Other Models

In this section, we give a comparison between three different prediction techniques. The first technique is the latest work based on the uniformity assumption by Weber et al. [33], the second technique is the latest work based on the fractal dimensionality by Korn et al. [22], and the third technique is the resampled index presented in this paper. Since locally parametric techniques are either restricted to other index structures (like the M-tree) or not applicable to high dimensions, we cannot include this category in our comparison. The same holds for sampling techniques which have been restricted to the relational model so far.

Again, we ran 500 21-NN queries on the TEXTURE60 dataset and a bulkloaded VAMSplit R*-tree with 8,641 leaf pages as the basis of our comparison. The measured average number of leaf page accesses was 681.

In case of the uniform model, Weber et al. [33] predict that for 14 split dimensions, from dataset dimensionality 40 onwards all pages will be accessed. In the TEXTURE60 dataset, we have only 13 split dimensions (and therefore a larger average page extent), so the point where all pages are accessed will be reached even earlier. Therefore, we obtain 8,641 pages as the prediction from the uniform model. The reason for this high overestimation is the wrong assumption that pages are created by splitting the dataspace in the middle and the usage of the embedding rather than the inherent data dimensionality.

In case of the fractal dimensionality, we first measured the fractal dimension D_0 and the correlation fractal dimension D_2 of the TEXTURE60 dataset which is 0.093914 and 0.003623, respectively. Plugging these values into the formulas developed in [22], we got a predicted number of page accesses of 5,892. This value is lower than the uniform prediction but still way above the measured number of page accesses.

[9]Since the number of data points would have to grow exponentially with the dimensionality to keep the distribution uniform, higher data dimensionalities are practically infeasible.

Method	Pages accessed	Rel. error
Uniform	8,641	1,169%
Fractal	5,892	765%
Resampled	701	3%

Table 4: Prediction accuracy for different models (TEXTURE60 dataset)

Only the resampling technique performed well in this high-dimensional setting, resulting in a prediction of 701 pages. Table 4 summarizes these results. Note that for our 360 and 617-dimensional datasets, the fractal dimensionality approach is not applicable anymore, since the number of points is too small compared to the number of dimensions. Our approach, on the other hand, gave reasonable predictions even for these datasets with a relative error between −8% and +0.7%.

6. APPLICATIONS OF THE PREDICTION MODEL

In this section, we show how our prediction model can be used in order to improve the performance of index structures. Two parameters are important in this respect: the index page size and the number of dimensions stored in the index. The next two sections explain how our model aids the determination of these parameters.

6.1 Determining the Optimal Page Size

The size of the index pages has a big impact on the overall performance. If the pages are too small, more pages have to be read resulting in more expensive seek operations. If the pages are too large, more unnecessary points are loaded during queries resulting in higher transfer costs. The optimum lies somewhere in the middle.

Our model allows the determination of this optimum within minutes rather than hours. Figure 13 shows the result for the LANDSAT dataset. We compute the number of leaf page accesses for 500 21-NN queries using our model and an on-disk index. The I/O cost is then determined by assuming a seek time of 10 ms and a transfer rate of 20 MB/s. All page accesses are assumed to be random (which was confirmed to be true for the on-disk index).

As can be seen, our model resembles the measured cost very closely. Furthermore, it correctly predicts that the lowest query cost is reached for a page size of 64 KBytes. Note that this conclusion is reached within minutes instead of constructing the entire index structure for different page sizes

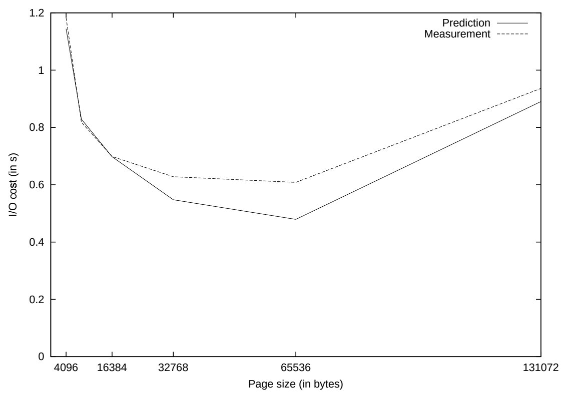


Figure 13: Determining the optimal page size (LANDSAT dataset)

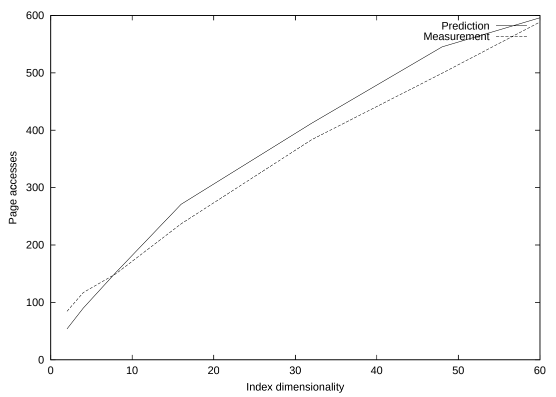


Figure 14: Feature page accesses for 21-NN queries (LANDSAT dataset)

which could take hours.

6.2 Determining the Optimal Data Dimensionality

Instead of building an index structure on all dimensions, it is possible to build the index structure on a reduced number of dimensions and store the remaining dimensions in a so-called *object server*. Seidl et al. [30] give an optimal NN search algorithm for this case. Their algorithm accesses index pages and object server pages in an interleaved way. With a minor modification, our prediction scheme can be used to predict both of these access types. The interested reader is referred to [23] for more details. Here, we only present the result for the index page prediction.

Figure 14 compares prediction and measurement of the number of index page accesses for 21-NN queries and different dimensionalities (number of dimensions stored in the index structure). The number of page accesses increases with increasing dimensionality because the page capacity is reduced. Both the measurement and the prediction show this trend with the prediction resembling the measurement very closely.

7. CONCLUSIONS

In this paper, we presented a new sampling-based technique for predicting the performance of index structures. To the best of our knowledge, this is the first approach that uses sampling to predict the page accesses for high-dimensional data.

Our technique has the following advantages:

- it is *simple* since it uses the same partitioning algorithms as the original index structure only applied to a subset of the data. This means that the bulk-loading algorithm of a given index structure can be simply reused to predict the page layout in memory. Nearly no code has to be generated or changed.
- It *works for high-dimensional clustered data* since sampling is independent of the dimensionality and preserves clusters.
- It is *reasonably fast*. Compared to building the index on disk, we achieved up to two orders of magnitude speedup. This allows to use this technique for tuning of index structures. Instead of spending days building index structures with different parameters, our model allows this evaluation to be done within minutes.
- It is *very accurate*. This holds not only for the average relative error (which was typically below 5%) but also for the correlation between measurement and prediction.

8. ACKNOWLEDGEMENTS

We would like to thank Mirek Riedewald, Hakan Ferhatosmanoğlu, and Ömer Eğecioğlu for many fruitful discussions. We also thank the anonymous referees for their comments.

9. REFERENCES

[1] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 13–24, 1999.

[2] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest neighbor searching. In *Symposium on Computational Geometry*, pages 336–344, 1995.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[4] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM Symp. on Principles of Database Systems*, 1997.

[5] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures. In *Proceedings of the Int. Conf. on Extending Database Technology, 1998, Valencia, Spain*, pages 216–230, 1998.

[6] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid technique: Towards breaking the curse of dimensionality. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, 1998.

[7] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 28–39, 1996.

[8] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proc. Int. Conf. on Data Engineering*, pages 440–447, February 1999.

[9] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. ACM Symp. on Principles of Database Systems*, pages 268–279, 2000.

[10] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *9th Australasian Database Conference (ADC'98)*, pages 15–26, 1998.

[11] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 426–435, 1997.

[12] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. ACM Symp. on Principles of Database Systems*, pages 4–13, 1994.

[13] C. Faloutsos, T. K. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 426–439, 1987.

[14] M. Freeston. The BANG file: A new kind of grid file. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 260–269, 1987.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[16] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proc. ACM Symp. on Principles of Database Systems*, pages 190–201, 1993.

[17] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7), July 1961.

[18] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *Proc. Int. Conf. on Data Engineering*, 2000.

[19] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. Conf. on Information and Knowledge Management*, pages 490–499, 1993.

[20] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, 1997.

[21] F. Korn, T. Johnson, and H. V. Jagadish. Range selectivity estimation for continuous attributes. In *Proc. International Conference on Scientific and Statistical Database Management*, pages 244–253, 1999.

[22] F. Korn, B.-U. Pagel, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. Int. Conf. on Data Engineering*, 2000.

[23] C. A. Lang and A. K. Singh. Performance prediction of high-dimensional index structures using sampling. Technical Report TRCS00-16, Univ. of California at Santa Barbara, 2000.

[24] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[25] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–11, 1990.

[26] D. B. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *TODS*, 15(4):625–658, 1990.

[27] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric, multikey file structure. *ACMTrans. Database Systems*, Vol. 9(1):38–71, 1984.

[28] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in R-trees. In *Proc. Int. Conf. Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 394–408, 1997.

[29] J. T. Robinson. The kdb-tree: A search structure for large multi-dimensional dynamic indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[30] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1998.

[31] Y. Theodoridis and T. K. Sellis. A model for the prediction of R-tree performance. In *Proc. ACM Symp. on Principles of Database Systems*, pages 161–171, 1996.

[32] R. Weber and S. Blott. An approximation based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no. 9141), October 1997.

[33] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 194–205, 1998.

[34] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. In *Proc. Storage and Retrieval for Image and Video Databases (SPIE)*, pages 62–73, 1996.

[35] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. Int. Conf. on Data Engineering*, pages 516–523, 1996.