Performance Prediction of High-Dimensional Index Structures using Sampling

Christian A. Lang
clang@cs.ucsb.edu

Ambuj K. Singh
ambuj@cs.ucsb.edu

Technical Report TRCS00-16

Department of Computer Science
University of California
Santa Barbara, CA 93106

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 has been proposed in previous years [BKSS90, BKK96, Gut84, KS97, WJ96b, LS90, Fre87, LJF94, CM99, BBK98a, WSB98]. 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?

The indexing scheme defines how the dataspace is partitioned into pages. The most commonly used schemes are data-partitioning schemes (e.g. R-trees [Gut84]), space-partitioning schemes (e.g. Quadtrees [FB74]), and hashing schemes (e.g. VA-file [WSB98]). We will concentrate on data-partitioning schemes (especially the VAMSplit R*-tree [WJ96a]) in this paper. However, some work on prediction was also done for space-partitioning [FG96] and hashing techniques [WSB98].

Equally important is the expected type of queries. Here we can mainly distinguish between range queries, nearest neighbor (NN) queries, and spatial joins. NN queries add the difficulty that — unlike range queries — the query shape depends on the location of the query. Our work concentrates on NN queries but can also be applied to the simpler problem of range queries. A recent work on spatial join prediction was done by Faloutsos et al. [FSTJ00].

Finally, we can model the dataset in different ways. The most common assumption is uniformity of the data which leads to great simplifications of the prediction formulas. The largest fraction of prediction research is based on this assumption. However, most real data is not uniform. More recent work on performance prediction tries to capture this fact by using global parameters (e.g. fractal dimensionality [FK94]), several local parameters (e.g. histograms [JAS00]), or sampling [LNS90] to model the data. We will discuss each approach and its advantages and disadvantages in more detail in Section 2.

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 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.

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. We end with some concluding remarks in Section 6.

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.

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. [FSR87] but was restricted to one-dimensional data. In a later paper, Kamel and Faloutsos [KF93] 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. [AMN95] 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. [BBKK97] 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. They account for boundary effects by cutting off these spheres at the boundary. In reality, however, the search radius would also need to grow with growing proximity to the boundary. The page prediction is computed again by using Minkowski sums. Weber et al. [WSB98] 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.

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. [FK94] 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 [PM97] extended it for 1-NN queries in R-trees. Korn et al. [KPF00] 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 that recursively splits the dataspace into smaller and smaller regions and counts the regions containing data points. The fractal dimensionality is basically the rate of change of these counts with shrinking regions. Since the number of regions grows exponentially with the dimensionality but the number of points does not, the number of regions containing points will stay nearly constant (equal to the number of points). Therefore, the fractal dimensionality will be close to 0 in high-dimensional settings, reflecting the fact that the data is very sparse.

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.

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. [CP98] 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 [TS96] 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. [APR99] show how the prediction error of histograms can be reduced by reducing the variance of densities within the histogram regions. Korn et al. [KJJ99] demonstrate how discontinuities at the histogram region edges can be avoided by using splines. Finally, Jin et al. [JAS00] 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.

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. [LNS90] show how to estimate selectivity by sampling using confidence intervals. Haas et al.

[HNSS93] give a similar analysis for join operations. 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. 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.

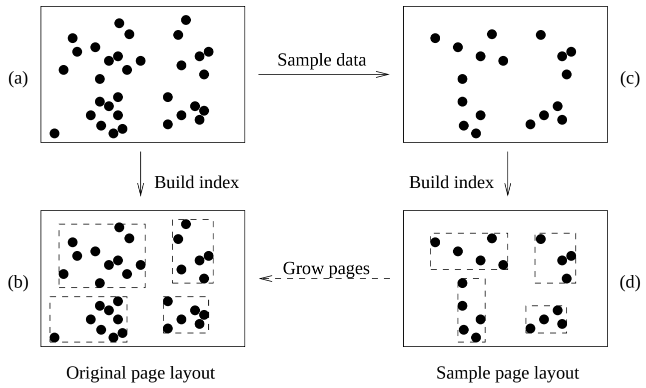


Figure 1: Connection between original and sample page layout

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 $0 < \zeta < 1$?

For $d = 1$, the bounding box degenerates to an interval. Once we solve the problem for this case, we can assume the original bounding box changes the same way for each single dimension and therefore have a solution for the original problem.

The expected length $L(m)$ of the bounding interval of m uniformly distributed points whose values range from 0 to 1 can be calculated as follows: Let x_i denote the value of the ith point. There are $m!$ ways how the points can be ordered. Without loss of generality, we assume the points are ordered according to their index, i.e. $x_i < x_j$ for $i < j$. To get the expected length of the interval, we have to calculate the average of $x_m - x_1$ under all possible values for the x_i. Each x_i can only have values in the range $[0, x_{i+1})$. This leads to the following formula for the expected interval length:

$$L(m) = m! \int_0^1 \int_0^{x_m} \ldots \int_0^{x_2} (x_m - x_1) dx_1 \ldots dx_{m-1} dx_m$$

Simplifying the integrals, we get

$$L(m) = \frac{m-1}{m+1}$$

[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

Therefore, if we change the number of points from C to $C \cdot \zeta$, the length of the interval will change on average by

$$\frac{L(C \cdot \zeta)}{L(C)} = \frac{(C \cdot \zeta - 1)(C + 1)}{(C \cdot \zeta + 1)(C - 1)}$$

It follows

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.

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 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).

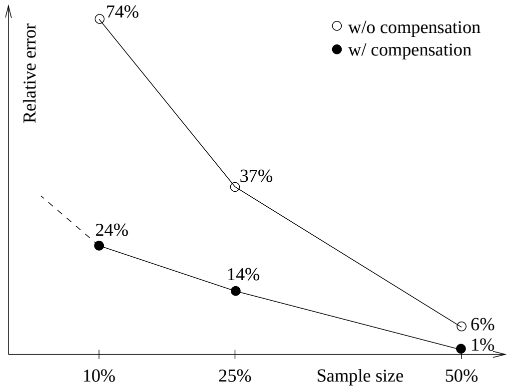


Figure 2: Relative error for different sample sizes

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. [BBK98b]. As a splitting strategy we chose the *maximum variance*-split. This results in a layout of pages similar to the VAMSplit R*-tree [WJ96a]. 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. In order to develop a cost formula for building the index, we will briefly describe each step of the algorithm (for a more detailed description, cf. [BBK98b]). Figure 3 gives the pseudo code.

The tree topology in line 1 is based on the maximum and the effective data page capacity (denoted by $C_{max,data}$ and $C_{eff,data}$ respectively) and the maximum and effective directory page capacity (denoted by

```
Determine the tree topology;
BuildTreeLevel( height, 0, N );
End;

BuildTreeLevel( int level, int start, int end ) {
   If level == 1 Then CreateLeafNode( start, end ); Return; Endif
   Determine fanout at this level;
   PartitionData( start, end, fanout );
   For each part (let it start at pstart and end at pend)
      BuildTreeLevel( level-1, pstart, pend );
   CreateInnerNode with pointers to nodes created by the recursive calls; }

PartitionData( int start, int end, int parts ) {
   If parts == 1 Then Return; Endif
   Find dimension with maximal variance, let it be d;
   SelectOnDimension( d, start, end, (start+end)/2 );
   PartitionData( start, (start+end)/2, parts/2 );
   PartitionData( (start+end)/2+1, end, parts/2 ); }

SelectOnDimension( int dim, int start, int end, int n ) {
   If end-start <= M Then
      Load data into memory;
      Perform select in memory;
      Write data back to disk;
   Else
      Read 3 chunks of size M/3 each from different locations into memory;
      Determine median of these points in dimension dim;
      Bipartition data according to median in dimension dim,
         let cut denote the location of the median after the bipartitioning;
      If cut <= n Then
         SelectOnDimension( dim, cut, end, n );
      Else
         SelectOnDimension( dim, start, cut, n );
      Endif;
   Endif; }
```

Figure 3: The top-down bulk loading algorithm with restricted memory

N	number of datapoints
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

$C_{max,dir}$ and $C_{eff,dir}$ respectively) which are provided by the user[3]. The tree height (denoted by $height$) can be calculated as

$$height = \left\lceil \log_{C_{eff,dir}}\left(\frac{N}{C_{eff,data}}\right)\right\rceil + 1.$$

Line 2 starts the process of building the index by calling the function `BuildTreeLevel`. Let us denote the cost of this function by $cost_{BuildTreeLevel}$. The cost of the whole on-disk bulk loading algorithm is then

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

We give a recursive definition of the latter cost function in the appendix. In developing this recurrence formula, we assume that the split during the bipartition step (line 25) happens exactly in the middle. In reality, these splits can be quite off the middle, leading to a higher actual cost than that predicted. Our experiments will show that such off-middle splits can introduce a five to ten fold increase in the I/O cost.

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

[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 $height$.

[3]These capacities provide a good example for tuning parameters that can be optimized using our prediction technique.

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 4). 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

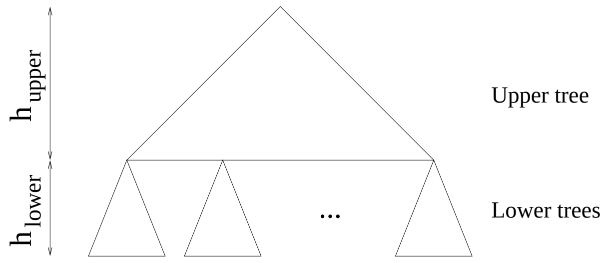


Figure 4: Partitioning of the index tree

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 (cf. Equation 6 in the appendix) at each level of the tree into account.

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

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

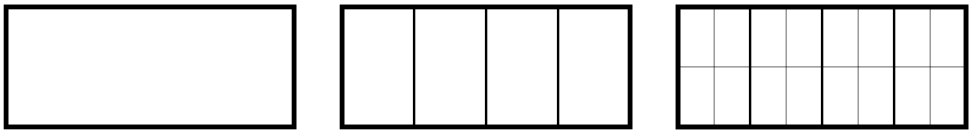
where q denotes the number of query points.

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 5 (a). If the points within the page are distributed uniformly, the dimension with



(a) Upper tree leaf page geometry (b) After partitioning into 4 parts (c) After partitioning each part into 4 subparts

Figure 5: Splitting of a page

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 5 (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. [BBKK97] 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 geometry 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 6. Note that only steps 2 and 3 introduce I/O cost. The scanning in step 3 causes a cost of

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

```
(1) Determine the tree topology;
(2) Read q query points randomly from dataset;
(3) Scan the whole dataset to determine the query spheres and
(4)    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 6: The cutoff tree prediction algorithm

Therefore, the overall cost of this approach is

$$cost_{Cutoff} = cost_{ReadQueryPoints} + cost_{ScanDataset}. \tag{3}$$

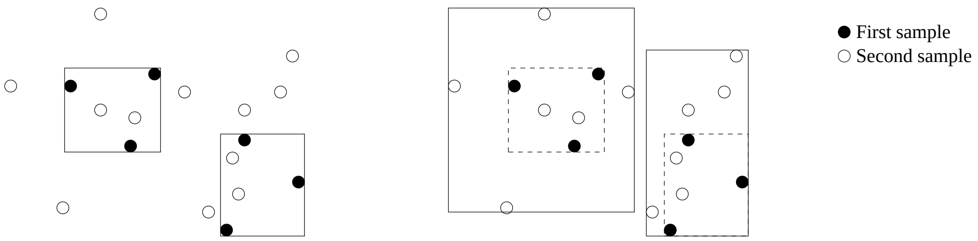
4.4 The Resampled Index Tree

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[4]. 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 7 (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 7 (b). Since all sampling is performed uniformly, we can expect each page to contain on average

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



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

Figure 7: Growing pages by resampling

```
 (1) Determine the tree topology;
 (2) Read q query points randomly from dataset;
 (3) Scan the whole dataset to determine the query spheres and
 (4)    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 and
 (7)    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 8: The resampled tree prediction algorithm

three times as many points as before.

In the general case, we have k upper tree leaf pages[5]. 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.

In order to establish a cost formula for this approach, we state the algorithm in pseudo code in Figure 8. Steps 2 and 3 cause the same cost as for the cutoff index tree. Step 6 is performed as follows (cf. Figure 9). We read $N \cdot \sigma_{lower}$ points from the original data file into memory. Since we have a memory size of M, we

[5]Note that k can be determined by the formula in Equation 6 in the appendix.

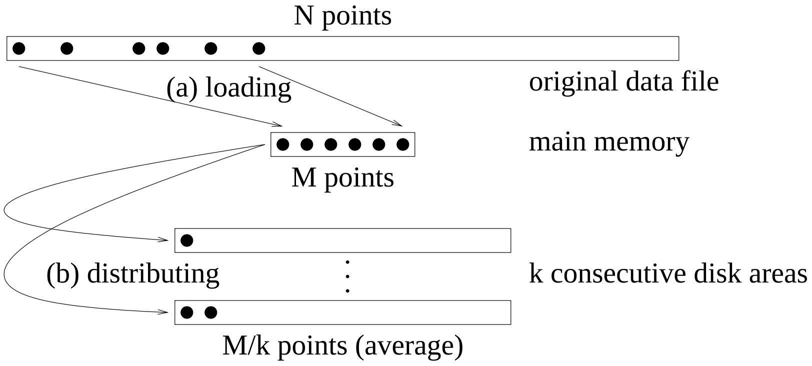


Figure 9: Read and write operations during the resampling step

read these sample points in chunks of size M (step a). Therefore, the number of chunks is $\left\lceil \frac{N}{M} \cdot \sigma_{lower} \right\rceil$. 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 blocks are transfered from memory to disk, this distributing step costs $k \cdot t_{seek} + \left\lceil \frac{M}{B} \right\rceil \cdot t_{xfer}$.

The whole step 6 therefore costs

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

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

$$cost_{Resampled} = cost_{ReadQueryPoints} + cost_{ScanDataset} + cost_{Resampling} + cost_{BuildLowerSubtrees}. \tag{5}$$

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.

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:

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

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 the appendix. 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.

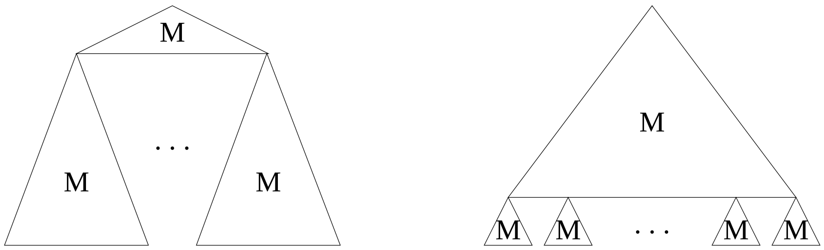
Upper Bound on h_{upper} In order to ensure that the capacity of the upper tree leaf pages is at least 2, we can equivalently ensure that a tree of full height built on $N \cdot \sigma_{upper}$ $(= M)$ points stores at least 2 points in its pages in level $height - h_{upper} + 1$. Therefore, the upper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

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

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 accomodate all points of one lower tree in memory (cf. Figure 10). This in turn leads to too small lower



(a) small upper tree / large lower trees (b) large upper tree / small lower trees

Figure 10: Small and large h_{upper}

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.

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.

Figure 11 depicts qualitatively the tradeoff between prediction accuracy and I/O cost for changing h_{upper} values. For small h_{upper} values, the I/O cost is small but the prediction error is high due to ill-defined

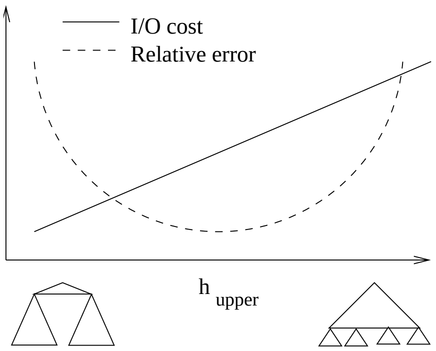


Figure 11: I/O cost and accuracy vs. h_{upper}

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[6]. Figure 12 compares the I/O cost of all three approaches for different memory sizes M. We assume a dataset with one million points

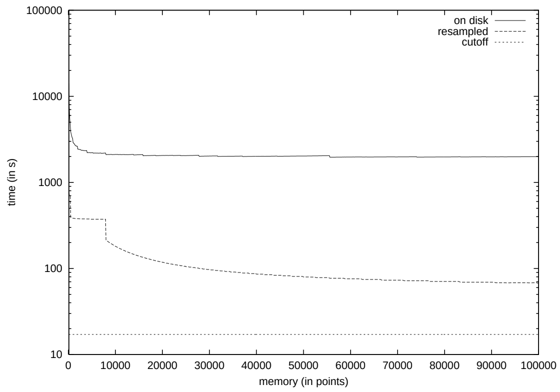


Figure 12: I/O cost for different memory sizes M

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 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 13 compares the I/O cost for different dataset sizes. In this case, we fixed the memory size at $M = 10,000$ and the dimensionality at 60. 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.

A similar observation can be made when varying the dimensionality of the data. Figure 14 plots the I/O cost curves for this case. 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

[6]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.

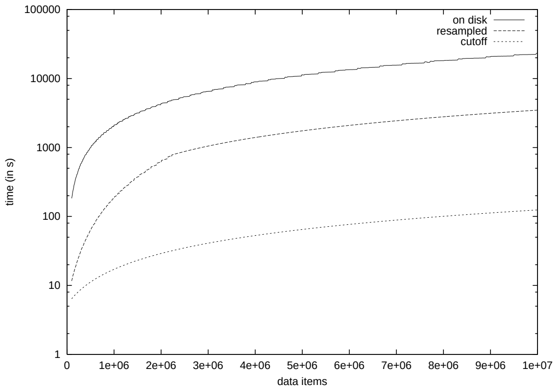


Figure 13: I/O cost for different dataset sizes N

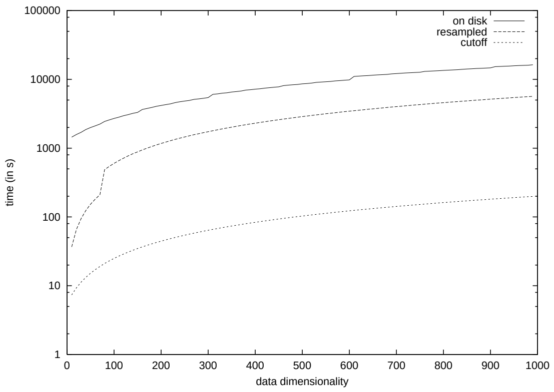


Figure 14: I/O cost for different data dimensionalities

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.

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 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. The next section presents the measured I/O cost and prediction accuracy for real datasets.

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. Table 7 summarizes

Method	Rel. error	Page seeks	Page transfers	I/O cost (in s)
On-disk	0%	$82,896 + 345,298$	$600,097 + 346,666$	$4,660.645$
Resampled ($h_{upper} = 3$, $\sigma_{upper} = 0.00363$, $\sigma_{lower} = 0.13068$)	-25%	$1,064$	$19,404$	18.402
Resampled ($h_{upper} = 4$, $\sigma_{upper} = 0.00363$, $\sigma_{lower} = 1$)	-1%	$16,433$	$47,259$	183.234
Cutoff ($h_{upper} = 3$, $\sigma_{upper} = 0.00363$)	-24%	501	$8,705$	8.492
Cutoff ($h_{upper} = 4$, $\sigma_{upper} = 0.00363$)	-93%	501	$8,705$	8.492

Table 3: Relative error and I/O cost for $M = 1,000$ (TEXTURE60 dataset)

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

the results for all datasets.

The height of the index tree in the TEXTURE60 example is 5. Table 3 lists the results for $M = 1,000$ and Table 4 lists the results for $M = 10,000$. 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

Method	Rel. error
Resampled ($h_{upper} = 2$, $\sigma_{upper} = 0.01$, $\sigma_{lower} = 0.04$)	-3%
Cutoff ($h_{upper} = 2$, $\sigma_{upper} = 0.01$)	-1%

Table 5: Prediction accuracy for $M = 1,000$ (uniform dataset)

Method	Rel. error
Resampled ($h_{upper} = 2$, $\sigma_{upper} = 0.1$, $\sigma_{lower} = 0.4$)	-1%
Cutoff ($h_{upper} = 2$, $\sigma_{upper} = 0.1$)	-0.5%

Table 6: Prediction accuracy for $M = 10,000$ (uniform dataset)

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

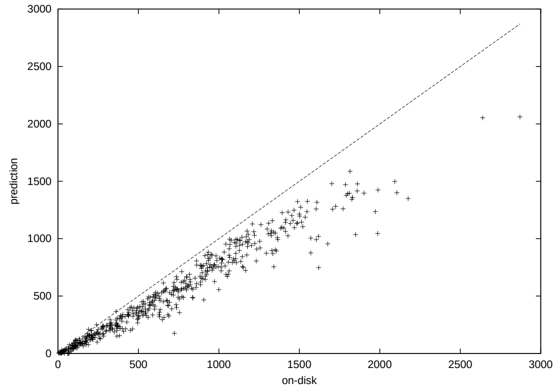
In this section we examine the accuracy of the predictions compared to the on-disk index. Tables 3 and 4 list the relative errors as a signed number to indicate under and overestimations. As discussed in Section 4.5 for the resampled approach, we underestimate the number of page accesses for small upper 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 memory sizes[7]. 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.

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. Figures 15(a), 15(b), and 15(c) show the correlation diagrams for the resampled approach with $M = 10,000$ and $h_{upper} = 2$, 3, and 4, respectively. As expected, these diagrams again show that underestimations occur for small values of h_{upper} and overestimations occur for large values of h_{upper}. On the whole, these diagrams also show how well measurement and prediction are correlated. For the cutoff approach in Figure 16, this correlation is nearly lost showing that the overall relative error alone is not a good enough measure for the quality of a prediction. Finally, Figure 15(d) 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.

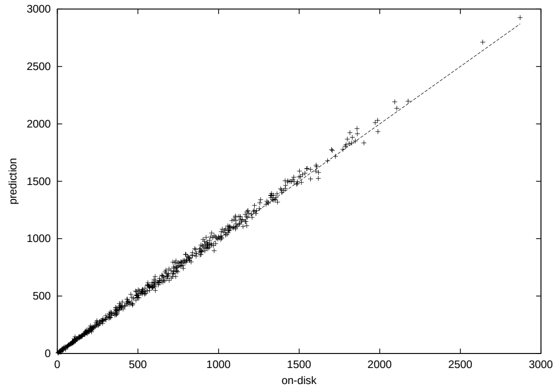
Since we assume uniformity of the data when inflating the upper tree leaf pages and during the page splits in the cutoff approach, we examined how well the predictions turn out for uniform data. We generated a dataset of 100,000 uniformly distributed points in 8 dimensions[8] and ran the same experiments as before. The index tree has a height of 3 in this example. Tables 5 and 6 show the results. As expected, the relative errors are negligible, thus confirming the assumption of our model.

[7]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%.

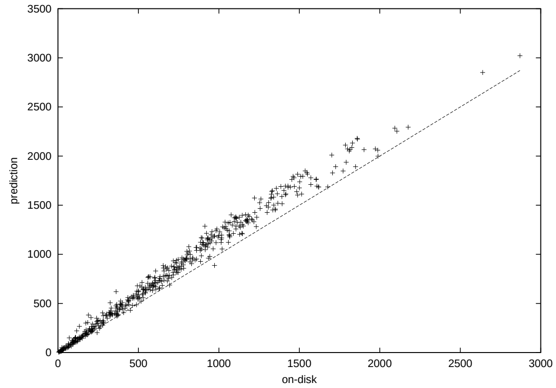
[8]Since the number of datapoints would have to grow exponentially with the dimensionality to keep the distribution uniform, higher data dimensionalities are practically infeasible.



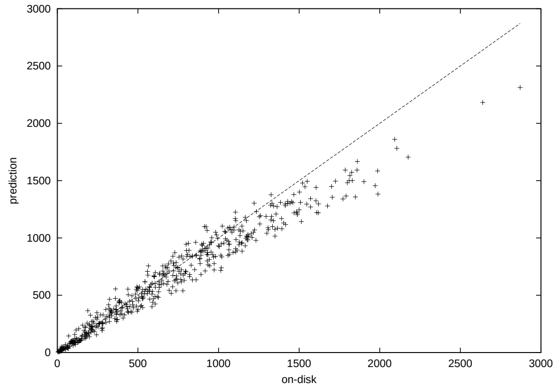
(a) $M = 10,000$ and $h_{upper} = 2$



(b) $M = 10,000$ and $h_{upper} = 3$



(c) $M = 10,000$ and $h_{upper} = 4$



(d) $M = 1,000$ and $h_{upper} = 4$

Figure 15: Correlation diagrams for resampled index (TEXTURE60 dataset)

Dataset	Resampling parameters	Rel. error
Uniform (8 dimensions)	$h_{upper} = 2$, $\sigma_{upper} = 0.01$, $\sigma_{lower} = 0.04$	-3%
TEXTURE48	$h_{upper} = 3$, $\sigma_{upper} = 0.0375$, $\sigma_{lower} = 1.0$	$+13\%$
TEXTURE60	$h_{upper} = 4$, $\sigma_{upper} = 0.00363$, $\sigma_{lower} = 1.0$	-1%
ISOLET617	$h_{upper} = 3$, $\sigma_{upper} = 0.1282$, $\sigma_{lower} = 1.0$	-8%
ISOLET617 (SVD-transformed)	$h_{upper} = 3$, $\sigma_{upper} = 0.1282$, $\sigma_{lower} = 1.0$	$+0.7\%$
STOCK360	$h_{upper} = 5$, $\sigma_{upper} = 0.1538$, $\sigma_{lower} = 1.0$	-2%

Table 7: Prediction accuracy for all datasets (resampled index with $M = 1,000$)

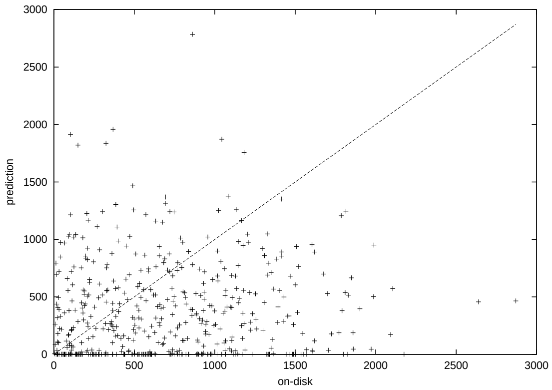


Figure 16: Correlation diagram for cutoff index with $M = 10,000$ and $h_{upper} = 3$ (TEXTURE60 dataset)

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. [WSB98], the second technique is the latest work based on the fractal dimensionality by Korn et al. [KPF00], 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. [WSB98] 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 [KPF00], 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.

Only the resampling technique performed well in this high-dimensional setting, resulting in a prediction of 701 pages. Table 8 summarizes these results. Note that for our 360 and 617-dimensional datasets considered in Table 7, the fractal dimensionality approach is not applicable anymore, since the number of points is too small compared to the number of dimensions.

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

Table 8: Prediction accuracy for different models (TEXTURE60 dataset)

6 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.

Summarizing, this work has shown that good cost prediction requires not only good data models but also knowledge about the specifics of the indexing scheme.

References

[AMN95] Sunil Arya, David M. Mount, and Onuttom Narayan. Accounting for boundary effects in nearest neighbor searching. In *Symposium on Computational Geometry*, pages 336–344, 1995.

[APR99] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 13–24. ACM Press, 1999.

[BBK98a] 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.

[BBK98b] Stefan Berchtold, Christian Böhm, and Hans-Peter 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.

[BBKK97] 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.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[CM99] 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.

[CP98] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *9th Australasian Database Conference (ADC'98)*, pages 15–26, 1998.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FG96] Christos Faloutsos and Volker Gaede. Analysis of n-dimensional quadtrees using the hausdorff fractal dimension. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 40–50. Morgan Kaufmann, 1996.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota*, pages 4–13. ACM Press, 1994.

[Fre87] Michael Freeston. The BANG file: A new kind of grid file. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 260–269. ACM Press, 1987.

[FSR87] Christos Faloutsos, Timos K. Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 426–439. ACM Press, 1987.

[FSTJ00] Christos Faloutsos, Bernhard Seeger, Agma J. M. Traina, and Caetano Traina Jr. Spatial join selectivity using power laws. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2000.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[HNSS93] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. Fixed-precision estimation of join selectivity. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC*, pages 190–201. ACM Press, 1993.

[Hoa61] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7), July 1961.

[JAS00] J. Jin, N. An, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *Proc. Int. Conf. on Data Engineering*, 2000.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *Proc. Conf. on Information and Knowledge Management*, pages 490–499, 1993.

[KJJ99] Flip Korn, Theodore 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.

[KPF00] Flip Korn, Bernd-Uwe Pagel, and Christos Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. Int. Conf. on Data Engineering*, 2000.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 369–380. ACM Press, 1997.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[LNS90] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 1–11. ACM Press, 1990.

[LS90] David B. Lomet and Betty Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *TODS*, 15(4):625–658, 1990.

[PM97] Apostolos Papadopoulos and Yannis Manolopoulos. Performance of nearest neighbor queries in R-trees. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 1997.

[TS96] Yannis Theodoridis and Timos K. Sellis. A model for the prediction of R-tree performance. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada*, pages 161–171. ACM Press, 1996.

[WJ96a] David A. White and Ramesh Jain. Similarity indexing: Algorithms and performance. In *Proc. Storage and Retrieval for Image and Video Databases (SPIE)*, pages 62–73, 1996.

[WJ96b] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 516–523. IEEE Computer Society, 1996.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

A Derivation of the Cost Formulas

A.1 Tree Properties

The following function defines the fanout of a node in level *level* in a tree of height h containing *items* data items:

$$fanout(h, level, items, f) = \begin{cases} f & \text{if } h < level \\ fanout(h-1, level, \frac{items}{f'}, f') & \text{otherwise} \\ \quad \text{where } f' = min(\left\lceil \frac{items}{C_{eff,data} \cdot C_{eff,dir}{}^{level-2}} \right\rceil, C_{max,dir}) & \end{cases} \tag{6}$$

The next function 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:

$$capacity(h, level, items, f) = \begin{cases} items & \text{if } h < level \\ capacity(h-1, level, \frac{items}{f'}, f') & \text{otherwise} \\ \quad \text{where } f' = min(\left\lceil \frac{items}{C_{eff,data} \cdot C_{eff,dir}{}^{level-2}} \right\rceil, C_{max,dir}) & \end{cases} \tag{7}$$

Both functions can be easily obtained from the algorithm presented in [BBK98b].

A.2 Cost Formula for the On-Disk Index

In this section, we derive the formula for the I/O cost of the on-disk index. In order to do so, we will give cost formulas for each of the functions in the algorithm in Figure 3. During the following discussion, we will refer to lines from this algorithm.

The function `BuildTreeLevel` starting at line 4 generates the nodes for one level of the tree. If we are at the leaf level, only a new leaf page is created on disk (line 5). Otherwise, the data is partitioned into $fanout$ parts where

$$fanout = fanout(height, level, end - start, 0)$$

$start$ and $end - 1$ denote the index of the first and last data item (in sequential order) of the chunk of data currently regarded.

For each part, `BuildTreeLevel` is called recursively in order to build the pages of the next lower level (line 8+9). Once these are built, a new inner node is created with pointers to these pages. The cost of `BuildTreeLevel` is therefore

$$cost_{BuildTreeLevel}(level, start, end) = \begin{cases} t_{seek} + t_{xfer} & \text{if } level = 1 \\ \\ cost_{PartitionData}(start, end, fanout) + & \\ \sum_{i=0}^{fanout-1} cost_{BuildTreeLevel}(level - 1, & \text{otherwise} \\ start + \frac{end-start}{fanout} \cdot i, start + \frac{end-start}{fanout} \cdot (i+1)) & \end{cases} \quad (8)$$

Note that we do not include the cost for creating the inner node in line 10 since we assume that all inner nodes are kept in memory.

The partitioning of the data is performed by the remaining portion of the code. `PartitionData` bipartitions the data recursively until the desired number of partitions is reached. To detect the dimension with the highest variance in line 13, the portion of the data starting at $start$ and ending at end has to be scanned. This induces a cost of

$$t_{seek} + \left\lceil \frac{end - start}{B} \right\rceil \cdot t_{xfer}$$

Therefore, the overall cost of this function can be calculated as

$$cost_{PartitionData}(start, end, parts) = \begin{cases} 0 & \text{if } parts = 1 \\ \\ t_{seek} + \left\lceil \frac{end-start}{B} \right\rceil \cdot t_{xfer} + & \\ cost_{SelectOnDimension}(start, end) + & \text{otherwise} \\ cost_{PartitionData}(start, \frac{start+end}{2}, \frac{parts}{2}) + & \\ cost_{PartitionData}(\frac{start+end}{2}, end, \frac{parts}{2}) & \end{cases} \quad (9)$$

Note that this formula is slightly simplified since in case $parts$ is odd, the bipartitioning needs to result in an uneven split with $\lceil parts/2 \rceil$ partitions in the first half and $\lfloor parts/2 \rfloor$ in the second half.

`SelectOnDimension` implements a slightly modified *find* algorithm [Hoa61] to select the kth smallest element of an array (k is in our case $(start + end)/2$) and at the same time causing a bisection of this array. It is modified because we are dealing with restricted memory. If the chunk of data from *start* to *end* fits entirely into memory, it is read, bisected in memory, and written back, introducing a cost of

$$2 \cdot (t_{seek} + \left\lceil \frac{end - start}{B} \right\rceil \cdot t_{xfer})$$

Otherwise, three samples of size $M/3$ each are read from disk (line 23) to determine the median of the data causing a cost of

$$3 \cdot (t_{seek} + \left\lceil \frac{M}{3 \cdot B} \right\rceil \cdot t_{xfer})$$

This median is then used as a pivot during the bipartitioning in line 25. We implemented this bipartitioning using two buffers of size $\frac{M}{2}$ as described by the authors [BBK98b]. Since we move two windows of this size over the chunk of data from *start* to *end* and read and write each window once, this step introduces a cost of

$$2 \cdot (\left\lceil \frac{end - start}{M/2} \right\rceil \cdot t_{seek} + \left\lceil \frac{end - start}{B} \right\rceil \cdot t_{xfer})$$

since there are $(end - start)/(M/2)$ windows and each window causes a random disk access.

Combining the above results, we have a cost of

$$cost_{SelectOnDimension}(start, end) = \begin{cases} 2 \cdot (t_{seek} + \left\lceil \frac{end-start}{B} \right\rceil \cdot t_{xfer}) & \text{if } end - start \leq M \\ \\ 3 \cdot (t_{seek} + \left\lceil \frac{M}{3 \cdot B} \right\rceil \cdot t_{xfer}) + & \\ 2 \cdot (\left\lceil \frac{end-start}{M/2} \right\rceil \cdot t_{seek} + \left\lceil \frac{end-start}{B} \right\rceil \cdot t_{xfer}) + & \text{otherwise} \\ cost_{SelectOnDimension}(start, \frac{start+end}{2}) & \end{cases} \tag{10}$$