Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces

Stefan Berchtold[1], Christian Böhm[1,2], H. V. Jagadish[3], Hans-Peter Kriegel[2], Jörg Sander[2]

1. *stb software technologie beratung gmbh, Ulrichsplatz 6, 86150 Augsburg, Germany*
Stefan.Berchtold@stb-gmbh.de

2. *Institute for Computer Science, University of Munich, Oettingenstr. 67, 80538 Munich, Germany*
{boehm,kriegel,sander}@dbs.informatik.uni-muenchen.de

3. *University of Michigan, 1301 Beal Ave, Ann Arbor, MI*
jag@eecs.umich.edu

Abstract

Two major approaches have been proposed to efficiently process queries in databases: Speeding up the search by using index structures, and speeding up the search by operating on a compressed database, such as a signature file. Both approaches have their limitations: Indexing techniques are inefficient in extreme configurations, such as high-dimensional spaces, where even a simple scan may be cheaper than an index-based search. Compression techniques are not very efficient in all other situations. We propose to combine both techniques to search for nearest neighbors in a high-dimensional space. For this purpose, we develop a compressed index, called the IQ-tree, with a three-level structure: The first level is a regular (flat) directory consisting of minimum bounding boxes, the second level contains data points in a compressed representation, and the third level contains the actual data. We overcome several engineering challenges in constructing an effective index structure of this type. The most significant of these is to decide how much to compress at the second level. Too much compression will lead to many needless expensive accesses to the third level. Too little compression will increase both the storage and the access cost for the first two levels. We develop a cost model and an optimization algorithm based on this cost model that permits an independent determination of the degree of compression for each second level page to minimize expected query cost. In an experimental evaluation, we demonstrate that the IQ-tree shows a performance that is the "best of both worlds" for a wide range of data distributions and dimensionalities.

1. Motivation

Many different index structures have been proposed for high-dimensional data (cf. [6, 12, 15, 16, 21]). Most multi-dimensional index structures have an exponential dependence (in space or time or both) upon the number of dimensions. In recognition of this fact, an alternative approach is simply to perform a sequential scan over the entire data set, obtaining the benefits at least of sequential rather than random disk I/O[1]. Clever bit encodings can then be devised to make this sequential scan faster (cf. [20]).

The trade-offs between these two classes of techniques (index structures and scan methods) are not simply expressed in terms of the number of dimensions. For instance when data sets are highly skewed, as real data sets often are, multi-dimensional index techniques remain more effective than a scan until a fairly high dimension. Similarly, when there are correlations between dimensions, index techniques tend to benefit.

The question we address in this paper is whether one can come up with an optimal combination of these two classes of techniques. Can one devise a single structure that has the benefits of a traditional multi-dimensional index structure for low dimensional data sets, and the benefits of a bit-mapped scan for high dimensional data sets? Can one, further, trade off dynamically so that the *fraction* of the data set scanned is a function of the portion of data space being considered, and the extent of skew and correlation in this subspace?

Our solution is the IQ-tree. The basic idea is to construct a tree structure top-down, but not necessarily to continue dividing until leaf nodes with data pointers are reached. Instead, at an appropriately chosen stopping point, a bit encoding is used to represent the data in the region covered. The details of this structure are presented in Section 3, along with the layout of the files on disk, and the proposed algorithms for construction, maintenance and searching. A key question to address is the selection of a good stopping point where the trade-off is made between further hierarchical sub-division and bit encoding. Evaluating this trade-off involves the development of a cost model. All of this is the subject of Section 4. In Section 5 we present a careful experimental evaluation of our technique, using both synthetic and real data sets, and compare it against previous techniques. In particular, we demonstrate that it does show a performance that is the “best of both worlds” as we expected. Section 6 has a few extensions and special cases of interest (such as dynamic insertions). Section 7 covers related work. We end with conclusions in Section 8.

For concreteness, we focus on the problem of finding nearest neighbors in a point data set. In Section 2, we discuss the difference between simply counting page accesses

1. For instance [7] suggests that sequential scan is likely to be the most effective search algorithm for the nearest neighbor problem in very high-dimensional spaces.

as a cost metric and differentiate between sequential and random I/O for a more careful cost metric. We show that this difference is likely to be significant in a high-dimensional indexing situation. Based on this understanding, we present a page access strategy suitable for the nearest neighbor problem. This strategy forms the basis for all the subsequent work in this paper.

2. Page access strategy

The actual page access strategy has a significant impact on the cost of an index look-up. In this section we pin down the specific access strategy used. Specifically, for the nearest neighbor problem, we develop a new optimal page access strategy. Consider an indexed access, say in the course of a selection query, which results in the need to fetch five data pages. The simplest implementation is to perform five random accesses, one to each of the specified pages on disk. This is the way most (to our knowledge, all) commercial index structures are written. This is a reasonable choice for most ordinary, one-dimensional indexes, where the selectivity is expected to be small - so that only a very small number of data pages are expected to be selected. However, when posting lists become large, one can order the list of data pages to be accessed according to the layout on disk, and greatly reduce seek times. In fact, modern intelligent disk drives have the built-in capability to do exactly this. In the case of high-dimensional index structures, the need for this sort of batch processing becomes even more acute. Firstly, due to the poor clustering of pages in high-dimensional data spaces, page selectivities tend not to be very good, even if a query selects only a small number of tuples. Secondly, many multi-dimensional index structures require multiple paths to be followed from root to leaf for a single given query region. This redundancy additionally increases the number of pages to be loaded.

There are query types (such as range queries) for which one can easily determine the set of data pages one has to load in order to answer the query. We simply have to process the affected parts of the directory and record the pointers to the data pages. However, for other query types (such as nearest-neighbor queries) this set of data pages is not known in advance. We will first present an optimal algorithm for the first case, where all pages are known in advance, and then extend our algorithm to the second class of queries, in particular nearest neighbor queries.

Let t_{seek} be the time to seek to a specified location on disk. Let t_{xfer} be the time to transfer one block from disk. Suppose an index selects n disk blocks out of a total N in the file and we know the position of each block denoted by p_i, $0 \le i < n$. We assume this list to be sorted according to the position on disk[1]. We can apply the following algorithm: We seek to the position p_0 of the first block on disk. Then, we look at the following block at location p_1. If p_1 is "close enough", it is better to scan through and read all the blocks in positions (p_0+1) to p_1. Else, it is less expensive to seek.

1. Note that although disk surfaces are two-dimensional objects, a one-dimensional view of having a linear file is a very accurate model. Experiments show that how "far" a seek proceeds forward or backward in the file has only negligible influence on the time consumed.

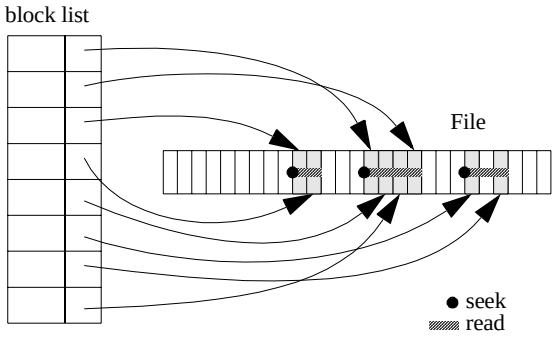

Figure 1: Optimal strategy to fetch blocks from disk (range q.)

After having made this decision, we continue with location p_2 and so on. Whenever seeking to the following block is less time-consuming than over-reading the blocks in between, we seek to this block, else we continue reading. Thus:

If $(p_{i+1} - p_i - 1) \cdot t_{xfer} \le t_{seek}$ then over-read
else seek

Figure 1 depicts an example solution. In the extreme cases, our algorithm chooses either to load the whole database in a single scan (if n is relatively large with respect to N) or to load all pages with random accesses (if n is relatively small with respect to N). However, also intermediate steps of partially scanning the database may occur. With this intermediate solution, we stand to benefit for even quite selective queries as long as the pages accessed have a tendency to cluster.

In [19], Seeger et al. solved a more general problem. In particular, they considered the fact that only a limited buffer in main memory might be available and therefore, not an arbitrarily large chunk of data can be read from disk. Furthermore, they proved that this algorithm is optimal i.e., it minimizes the time needed to load given n blocks.

2.1. Nearest neighbor search

In the case of nearest neighbor search we do not know in advance exactly which blocks to examine. In this case we can adapt the algorithm proposed in [13] by Hjaltason and Samet, which has been proven in [3] to load a minimal number of pages. The algorithm examines blocks in an expanding radius from the given query point until all remaining blocks are farther away (minimum distance) than the nearest data point found. The algorithm therefore keeps a priority list $Q[0..n\text{-}1]$ of pages to be loaded from disk. The list orders the data pages according to increasing distance from the query point and can be generated in a pre-processing step.

```
hjaltason_samet_nearest_neighbor (Index ix, Point q) {
    float nndist = ∞ ;
    pl.init (ix) ;      // initialize pl with all page descriptions
                        // from the directory ordered by MINDIST
    while (nndist ≥ MINDIST (pl.top) ) {
        page = load (pl.top) ;
        foreach Point p ∈ page
            if (distance (p,q) < nndist)
                nndist = distance (p,q) ; nn = p ;
        pl.pop () ;
}   }
```

We know that in any case we have to load the first page on the list $b_{Q[0]}$ in order to guarantee a correct answer to the query. For all other pages, there exists a certain probability that the page has to be loaded. This probability can be estimated using a cost model for nearest-neighbor queries. In Section 2.2, we describe the details of the cost model we are using for this purpose.

Thus, we face the following situation: We are given a list of n blocks; for each block b_i, $0 \le i < n$, we know the position p_i of the block on disk and a probability l_i that the block has to be loaded during query processing. We also know that the probability $l_{Q[0]}$ equals 100%. All pages that have already been processed, have an access probability of 0%, because no page needs to be accessed for a second time.

The basic idea of our nearest neighbor algorithm is to load in each processing step not only page $p_{Q[0]}$ but also other pages with positions near by $p_{Q[0]}$ (such as $p_{Q[0]+1}$, $p_{Q[0]-1}$ etc.), if their access probabilities are high.

In order to determine a good strategy for loading the desired blocks, we have to trade off the corresponding costs. If we decide to load an additional page, we pay a transfer cost. In contrast, if we decide *not* to load the additional page, we run into the risk that we have to access the page later, which causes the corresponding transfer cost *plus* an additional random seek operation. Taking together, we have to balance between direct cost and potential savings when deciding whether to read an additional page or not.

Loading a page with an access probability l_i, causes the following increment to the total estimated cost:

$$c_i = t_{\text{xfer}} - l_i \cdot (t_{\text{seek}} + t_{\text{xfer}}) . \qquad (1)$$

If c_i is negative, then the estimated savings is greater than the additional transfer cost.

Our algorithm behaves as follows: We start from the first block $b_{Q[0]}$ in the list which has a probability of 100% and therefore must be loaded anyway. Then, we consider the next block $b_{Q[0]+1}$ and determine the corresponding access probability $l_{Q[0]+1}$ and cost balance $c_{Q[0]+1}$. If the cost balance is negative, we note to load the sequence $[b_{Q[0]}, b_{Q[0]+1}]$ and continue searching for additional pages to be loaded together with our sequence. Otherwise, we determine the cost balance of the subsequent page $c_{Q[0]+2}$. If the sum of the balances $c_{Q[0]+1} + c_{Q[0]+2}$ is negative, we can note to load the sequence $[b_{Q[0]}, b_{Q[0]+1}, b_{Q[0]+2}]$. Otherwise we determine the cost balance of the next page and try to make the cumulated cost balance negative in one of the following steps. Our algorithm stops when the cumulated cost balance is higher than the seek cost. The same procedure is performed with the pages preceding the pivot page $b_{Q[0]}$.

When the sequence of blocks around $b_{Q[0]}$ with minimum cost balance is determined, we actually load and process the sequence of pages. As the nearest neighbor distance may change, the priority list is pruned i.e., all pages with a distance larger than the nearest neighbor distance are discarded. All pages in the sequence are also deleted from the priority list, and a new pivot page $b_{Q[0]}$ is determined.

Note that we slightly simplified the description of the following algorithm in order to maintain readability:

```
time_optimized_nearest_neighbor (Index ix, Point q) {
    int first, last, i ;
    float a, ccb, nndist = ∞ ;
    pl.init (ix) ;      // initialize pl with all page descriptions
                        // from the directory ordered by MINDIST
    while (nndist ≥ MINDIST (pl.top) ) {
        first = last = page_index (pl.top) ;
        ccb = 0.0 ; //cumulated cost balance
        i = last + 1 ;
        do {
            // forward search for pages to load additionally
            if (is_processed_already (b_i))
                a = 0 ;
            else
                a = determine_probability (b_i) ;
            ccb += t_xfer - a * (t_seek + t_xfer) ;
            if (ccb < 0) {
                last = i ;
                ccb = 0.0 ;
            }
            i ++ ;
        } while (ccb < t_seek) ;
        ccb = 0.0 ;
        i = first - 1 ;
        do {
            // backward search for additional pages
            ...
        } while (ccb < t_seek) ;
        pages = load_page_sequence (b_first, b_last) ;
        foreach Point p ∈ pages
            if (distance (p,q) < nndist)
                nndist = distance (p,q) ; nn = p ;
        for (i = first ; i <= last ; i++)
            pl.delete (b_i) ;
}   }
```

2.2. Access probability of data pages

Given a query point and a data page, we need the access probability of this page for a nearest neighbor query. Note that we know the bounding boxes of the data pages because this information usually is stored in the directory of the index.

The nearest neighbor algorithm accesses a page b_i if and only if the pages processed before b_i contain no data point that is closer to the query point than the minimum bounding rectangle of b_i. The b_i-sphere is defined as the sphere around the query point that touches b_i. Therefore, the access probability of b_i corresponds to the probability that none of the pages in the priority list contains a point in the b_i-sphere. Note that the pages intersecting with the b_i-sphere are exactly all the pages with a higher priority than b_i. This is depicted in Figure 2, where the following situation is visualized: In the first step, our algorithm has already processed the

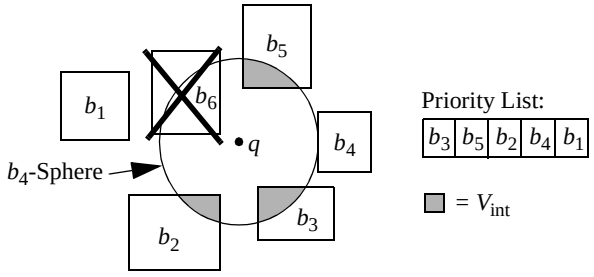

Figure 2: The b_i-sphere and pruning probability of a page

page b_6. Our next pivot page is b_3 (access probability 100%). As b_4 is on the following position on disk, we determine the access probability of b_4. Page b_4 is accessed if the b_4-sphere contains no data point. For this probability we have to determine the volume that is underlaid in grey in the figure. Only the pages b_3, b_5 and b_2, which have higher priority than b_4, intersect with the b_4-sphere. Page b_6 is ignored, because it has been processed before.

The probability that neither of the intersection volumes contains a data point corresponds to the product of the single probabilities not to contain a point in the intersection. We have to consider all pages which are located in the priority list before b_i. Assume j such that $i = Q[j]$:

$$P_{\text{access}}(b_i) = \prod_{0 \le k < j} P_{\text{no point in int}}(b_{Q[k]}) . \quad (2)$$

The probability that no point is in the intersection volume $V_{\text{int}}(b_k)$ is determined by the intersection volume, the volume of the *MBR* of the corresponding page region $V_{MBR}(b_k)$ and the number of points stored in the page $M(b_k)$:

$$P_{\text{no point in int}}(b_k) = \left(1 - \frac{V_{\text{int}}(b_k)}{V_{MBR}(b_k)}\right)^{M(b_k)} \quad (3)$$

The intersection volume of a page region given by its lower and upper bounds $((lb_0, ..., lb_{d-1}), (ub_0, ..., ub_{d-1}))$ and the b_i-sphere given by the center point q and the radius r of the b_i-sphere which corresponds to the distance between q and b_i can be determined by the following volume integration:

$$V_{\text{int}}(b_k) = \int_{lb_0}^{ub_0} \cdots \int_{lb_{d-1}}^{ub_{d-1}} \left(\begin{cases} 1 \text{ if } |p-q| < r \\ 0 \text{ otherwise} \end{cases} \right) dp_{d-1} \ldots dp_0 . \quad (4)$$

In the case of the maximum metric, the volume integral can be transformed into the following formula applying the maximum function to the lower bounds and the minimum function to the upper bounds in each dimension:

$$V_{\text{int,max}}(b_k) = \prod_{0 \le i < d} (\min\{ub_i, q_i + r\} - \max\{lb_i, q_i - r\}) \quad (5)$$

For Euclidean and other metrics, the volume can be estimated using approximations. Note that the model can also be extended to *k*-nearest neighbor queries: Instead of determining the probability that a point is located in a single volume V_{int}, we have to determine the probability density function of the number of points in V_{int}. Combining these probability density functions, we can determine the probability that b_k can be pruned in case of a *k*-nearest neighbor query. The details of the extended model are left out due to space restrictions.

3. The IQ-tree

The data space is divided into a set of partitions represented by the minimum bounding rectangles (MBRs) of the points located in the corresponding region of the data space. The MBRs are utilized for query processing, as usual, to restrict the search to relevant parts of the data space.

In high-dimensional query processing, however, we face the problem that the number of relevant parts thus selected may be rather large, resulting in a large number of data pages to be read from disk. Therefore, at the "data level" of query processing, the IQ-tree employs the strategies developed for efficient file scanning and for optimizing page accesses.

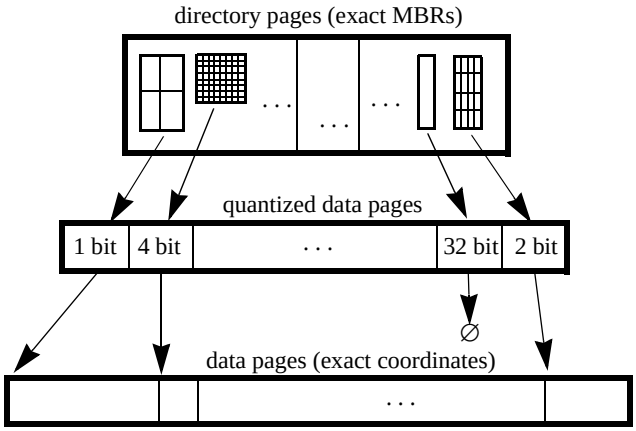

Figure 3: Structure of the IQ-tree

Additionally, compressed representations of the data points, using appropriate bit encodings, are processed instead of the exact geometry of the points. Only if a query condition cannot be evaluated based on the compressed representation of a point, is the exact geometry of that point consulted.

To minimize additional accesses to the exact geometry of points, the bit encodings of data points must be chosen very carefully. For skewed distributions the point density in some parts of the data space is higher than in other parts. Choosing just enough bits to separate the points in areas of low density would result in a poor encoding of points located in areas of high density, since many of them would then share the same compressed representation. On the other hand, choosing a more fine grained encoding, i.e. using more bits, to separate also the points in areas of high density, would unnecessarily increase the size of *all* approximations, thereby decreasing the efficiency of the technique.

Therefore, to get an optimal quantization with respect to accuracy *and* compression rate of the representations, the IQ-tree determines a separate encoding scheme for each partition, depending on the actual point density in the partition. Consequently, in some parts of the data space points may be represented more accurately than points located in other parts of the data space, using however as few bits as possible in each partition. Note that, in contrast to the VA-file (cf. section 5 or [20]), our quantizations are relative to the geometry of the MBR of the corresponding data page. Thus, we need even less bits to encode with the same accuracy. This strategy is similar to the concept of "actual coded data regions" proposed in [12].

3.1. Structure of the IQ-tree

The structure of an IQ-tree is illustrated in Figure 3. The tree consists of three levels organized in three distinct files. The first level, called the *directory*, stores directory pages, which contain exact (floating point) representations of MBRs. The second level consists of so called *quantized data pages*, i.e. data pages containing the data points in compressed form. The third level file consists of data pages containing the exact data.

Each MBR stored in the directory refers to a quantized data page which in turn refers to a data page containing exact coordinates of points. These MBRs describe the minimum rectangular region containing the points which are stored in the corresponding data pages.

Quantized data pages have a fixed block size which means that, depending on the length of the bit encoding, different numbers of points can be stored and represented in a single quantized data page. Consequently, the corresponding data pages on the third level have a variable size. Recall that the exact geometry of a point will be consulted only occasionally, if a query condition cannot be evaluated on the approximation of the point.

A number of g bits per dimension is used to approximate the location of points in a data page by virtually dividing the MBR along each dimension into 2^g partitions of equal size. The points contained in the corresponding region of the space are then represented by the bit encodings of the virtual grid cells in which they are located. The quantization factor g will, in general, vary from page to page, depending on the point density in MBR. Note, however, that if a page should be quantized using 32 bit, then we actually do not compute the grid cell approximations of the points. Instead, we use the 32 bits in the quantized data page to store the exact (floating point) representation. In this case, an explicit (and redundant) exact representation on the third level is omitted.

3.2. Searching

An IQ-tree could be seen as having a 2-tier directory consisting of a big linear root node and a second level containing boxes around actual data points. Thus, the search algorithms designed for a standard multi-dimensional search tree can be applied without significant changes.

The general condition for guiding a search is: If a specific area on a certain level of the tree cannot be excluded from the search then we have to follow the pointer associated with the corresponding MBR and recursively go down a level in the tree. This condition can be applied to MBRs for sets of points as well as to box approximations of single points. The condition also guarantees a minimum number of accesses to the exact representations of points, i.e. exact coordinates are loaded only if a point is a true answer or if the query condition cannot be evaluated using its compressed representation.

For instance, when evaluating nearest neighbor queries, the box approximations of points are inserted into the priority list just like regular MBRs of data pages. The exact representation of a point p must be consulted only if the approximation b_p of p becomes the pivot "page" of the priority list pl. This means that b_p has currently the minimal distance d_{min} to a query point q, among all other boxes and MBRs in pl. Then, b_p can never be pruned from pl because no distance between q and other points can be smaller than d_{min}. From this it follows that there is no strategy to avoid loading the exact coordinates of p. Even if the exact distances between q and all other points were now known, we only know that the distance between p and q is in a range d_{min} to d_{max} depending on the actual position of p within b_p. If there exists a point which is closer to q than d_{max} we must load the coordinates of p to decide which one is closer. In the other case, if there is no such point, we must load p because p then is the nearest neighbor of q. There is also no reasonable strategy to delay the lookup of p if b_p is the pivot page because the true distance between p and q is the smallest possible (and therefore is the most efficient) pruning distance available in the current state of the algorithm.

3.3. Construction and maintenance

To build an IQ-tree, we first have to perform a top-down partitioning of the data space until the points in each partition will fit into a quantized data page using a 1 bit representation. For that purpose, the partitioning scheme proposed in [4] is used. This strategy is effective for fast bulkloading of high-dimensional index structures such as the X-tree, and can be applied easily using a block size corresponding to a 1 bit quantization factor for each page. This yields an initial IQ-tree which is optimal with respect to compression rate, but which may perform poorly with respect to accuracy of representation since too many points on each data page may share the same bit encoding.

For the best query performance based on the IQ-tree architecture, obviously, the compression rate and the accuracy of the representations have to be optimized simultaneously. Therefore, we must now search for a refinement of the initial partitions using, where necessary, more bits for the quantization of data pages. Note that increasing the number of bits to represent the points on a data page will require more space, and therefore, less of data can be stored in a quantized data page with constant block size. Consequently, if the number of bits for the encoding of the points is increased, the partition must also be split. In these cases, we apply the same split heuristic as for the construction of the initial IQ-tree, i.e. we split the page along the dimension where the MBR has its largest extension.

An X-tree splits until all points fit into pages using a 32-bit (exact) representation, whereas a VA-file uses a constant number of bits to represent the data. In contrast, the IQ-tree is designed to adjust the splits as well as the degree of quantization to the actual distribution of the data, and thereby to optimize query performance. To be able to optimize, we need a cost estimate and we obtain this through the cost model given in section 3.4. The reader not interested in the details of the cost model may continue reading in section 3.6.

3.4. Cost model

The cost of query processing T_{QP} on an IQ-tree consists of the following components:

(1.) cost for loading the first level directory T_{1st}
(2.) cost for optimized loading and processing parts of the second-level directory T_{2nd}
(3.) cost for accessing the exact geometry T_{3rd}

All three cost components depend on our decision whether to use a quantized second-level for a subset of our data or not, and to quantize at what resolution.

Of all components of our cost model, the cost for the refinement step are the most sensitive to the data distribution and are most significant to the total query cost. Therefore, we spend the highest effort in the estimation of the refinement cost adapting the cost estimation as good as possible to the specific situation of the actual data distribution. In our

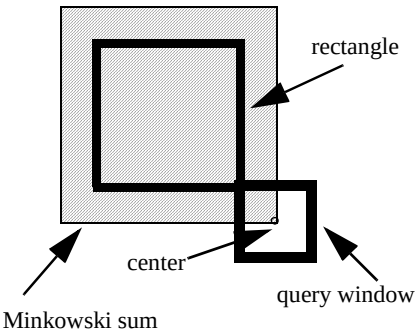

Figure 4: Estimating access probabilities by using the concept of the Minkowski sum

description, we start from a uniform, independent distribution. We extend our model then to allow for correlations by the concept of the fractal dimension.

We assume in our model that the distribution of the query points follows the distribution of the data points i.e., every data point is a potential query point.

Estimating cost for the refinement step

We are given a data page with a minimum bounding rectangle $MBR = (lb_0, ..., lb_{d-1}, ub_0, ..., ub_{d-1})$ and a number m of data points $P_0, ... P_{m-1}$. Let g be the number of bits used to quantize the data vectors. We want to determine the number of look-ups to the exact geometry.

Under uniformity/independence assumption, we can define the local point density ρ by using the volume of the page region:

$$\rho = \frac{m}{\prod_{0 \le i < d} (ub_i - lb_i)}. \quad (6)$$

From the point density, we can estimate the nearest neighbor distance, which is valid in the corresponding region of the data space. The radius is chosen such that the corresponding volume of the hypersphere contains an expectation of one data point[1]:

$$\frac{1}{\rho} = V_{query}(r) \qquad r = V^{-1}_{query}(\frac{1}{\rho}) \quad (7)$$

Depending on the metric we apply to the data, we have to use different formulas for V_{sphere}. In case of the Euclidean metric, it corresponds to the well-known formula for the volume of a d-dimensional hyper-sphere:

$$V_{query}(r) = V_{sphere}(r) = \frac{\sqrt{\pi}^d}{\Gamma(d/2+1)} \cdot r^d \quad (8)$$

with the gamma-function $\Gamma(x)$, which is the extension of the faculty operation into the domain of real numbers: $\Gamma(x+1) = x \cdot \Gamma(x)$, $\Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$. In case of the maximum metric, it is simply the d-dimensional hypercube:

$$V_{query}(r) = V_{cube}(r) = (2r)^d. \quad (9)$$

1. In order to adapt the cost model to k-nearest neighbor queries, one simply has to determine the volume in which an expected number of k points is located. To keep the formulas simple, we omitted this step in the presentation of the cost model.

The probability with which a point stored in the data page must be refined corresponds to the Minkowski sum of the query volume and the volume of the cell approximating the data point. The Minkowski sum is the enlargement of the cell by the query volume, as depicted in Figure 4. The volume of the quantization cell is given by:

$$V_{cell} = \prod_{0 \le i < d} \frac{(ub_i - lb_i)}{2^g} = \frac{V_{mbr}}{2^{d \cdot g}}. \quad (10)$$

The Minkowski sum can be exactly determined for the maximum metric:

$$V_{mink(cell,NN\text{-}sphere),max} = \prod_{0 \le i < d} \left(\frac{ub_i - lb_i}{2^g} + 2r \right) \quad (11)$$

For the Euclidean metric, it can be approximated by the following formula, where a is the geometrical mean of the side lengths of the *mbr* of the data page[2]:

$$V_{mink(..),eucl} = \sum_{0 \le k \le d} \binom{d}{k} \cdot a^k \cdot \frac{\sqrt{\pi}^{d-k}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot r^{d-k} \quad (12)$$

with $a = \frac{1}{2^g} \cdot \sqrt[d]{\prod_{0 \le i < d} (ub_i - lb_i)}$.

Under the uniformity and independence assumption, the Minkowski sum divided by the volume of the data space corresponds to the refinement probability of a data point. The expectation of the number of refinements per query is this probability multiplied with the number of data points P.

We can drop the assumptions of uniformity and independence by the concept of the fractal dimension [2, 3, 8, 9]. The fractal dimension takes correlation effects into account. In essence, correlated data points are not spread over the complete d-dimensional data space but concentrate on a lower-dimensional part of the data space (which is not an actual sub*space* of the data space, unless the correlation is linear). The fractal dimension D_F is the actual dimension of the subpart where the data is located.

If a volume containing correlated points becomes enlarged, the most important observation is that the number of enclosed points does not grow linearly with the volume enlargement. Rather, the number of enclosed points grows, as if the volume object had the dimension D_F rather than d. This can be exploited in our formulas in the following way: The point density ρ is replaced by a fractal point density ρ_F, which is determined according to the adapted volume:

$$\rho_F = \frac{M}{\left(\prod_{0 \le i < d} (ub_i - lb_i) \right)^{D_F/d}}. \quad (13)$$

The same adaptation is required for the estimation of the nearest neighbor distance:

$$r = V^{-1}_{query}(1/\rho^{d/D_F}). \quad (14)$$

The formulas for the cell size and the Minkowski sum are basically unaffected by correlation effects. However, as

2. A more detailed explanation of the computation of Minkowski sums can be found in [3].

we assume the query distribution to follow the distribution of the data points, the Minkowski sum doesn't any longer directly correspond to the access probability of the exact point data. Rather, the access probability is determined by the fraction of all query points located in the Minkowski enlargement. It is determined by the following formula:

$$P_{\text{refinement}} = \frac{\rho_F}{N} \cdot V_{\text{mink(cell,NN-sphere)}}^{D_F/d} \quad (15)$$

The cost for loading the second level directory comprises the probability with which a whole data page is accessed, together with the estimate of the reading cost incurred by our optimized nearest neighbor algorithm.

Estimating cost for the second-level directory

There are numerous cost models published for determining an expectation of the number of page accesses [1, 3, 9, 10, 17]. We only present a short description of the particularities in our situation. Further details of the applied cost model are presented in [8]. Our cost estimate for the second level directory T_{2nd} takes also correlation effects into account. We are given a number n of data pages and a number N of points and we have to determine the number k of pages to be read at least. The cost incurred by optimized reading is determined later. The volume of a page region is estimated such that it contains an expectation of N/n points:

$$V_{MBR} = \left(\frac{1}{n}\right)^{d/D_F} \cdot V_{DS}. \quad (16)$$

Likewise, the volume of the nearest neighbor sphere is to contain an expectation of one point:

$$V_{\text{NN-Sphere}} = \left(\frac{1}{N}\right)^{d/D_F} \cdot V_{DS}. \quad (17)$$

The Minkowski sum of the nearest neighbor sphere and the *MBR* is multiplied with n to estimate k:

$$k = n \cdot \frac{\rho_F}{N} \cdot V_{\text{mink(MBR, NN-sphere)}}^{D_F/d} \quad (18)$$

The Minkowski sum is computed as described above. If $D_F \geq \log_2(n)$, the Minkowski sum must be adapted to take boundary effects into account. The details are provided in [8].

Now we know that we have to read k out of n pages in the approximation level and thus, we have to develop a cost estimate for optimized reading of these pages. We know the cost for a random seek operation t_{Seek} and for the transfer of pages $t_{\text{X-fer}}$ and determine probabilities for the distances between two pages to be loaded depending on n and k. For this purpose, assume that the k pages are uniformly distributed over the file.

The probability that the distance between two pages to load is greater or equal to some parameter a is given by

$$P_{\text{dist} \geq a} = \left(1 - \frac{k}{n}\right)^a. \quad (19)$$

From this, the probability with which the distance is equal to a, can be derived:

$$P_{\text{dist} = a} = \left(1 - \frac{k}{n}\right)^a - \left(1 - \frac{k}{n}\right)^{a+1} \quad (20)$$

As the maximum number of pages which can be over-read corresponds to the quotient $v = t_{\text{seek}}/t_{\text{xfer}}$, we assume that for every distance greater than the quotient a seek is performed (causing the cost $t_{\text{seek}} + t_{\text{xfer}}$), whereas for every smaller distance a continuous read is performed causing the cost $a \cdot t_{\text{xfer}}$. Taken together, the cost formula is:

$$\begin{aligned} T_{2nd} &= \left(\sum_{1 \leq i < v} P_{\text{dist}=i} \cdot i \cdot t_{\text{xfer}}\right) + P_{\text{dist} \geq v} \cdot (t_{\text{seek}} + t_{xfer}) \\ &= \left(1 - \frac{k}{n}\right)^{\frac{t_{\text{seek}}}{t_{\text{xfer}}}} \cdot \left(\frac{k^2}{n} \cdot t_{\text{seek}} + \left(\frac{k^2}{n} - n + k\right) \cdot t_{\text{xfer}}\right) + n \cdot t_{\text{xfer}} \end{aligned} \quad (21)$$

Estimating cost for the first-level directory

The cost for loading the first level directory is linear in the number of data pages, as every data page has exactly one entry in the first level directory comprising *mbr*, and the address of the second level page.

$$T_{1st} = t_{\text{seek}} + t_{xfer} \cdot n \cdot \frac{|\text{Dir\_Entry}|}{\text{blocksize}} \quad (22)$$

Properties of the cost functions

The estimated cost for query processing is

$$T = T_{1st} + T_{2nd} + T_{3rd} \quad (23)$$

We are interested in the monotonicity of the cost components when changing the resolution g. We will show analytically that the refinement cost T_{3rd} is monotonically decreasing and that the derivative of T_{3rd} is monotonically increasing. For example, proceeding from 1 bit to 2 bits always improves the refinement cost, and the improvement is stronger than the improvement when going from 2 bits to 4 bits.

We present here the analysis under the simplifying assumption that splits decrease all side lengths of the *MBR* by the same factor. (A similar analysis is also possible for the case that some side lengths are affected more than others, but is not presented here due to space limitations). Under this assumption, equation (15) can be rewritten to

$$P_{\text{refinement}} = \left(\prod_{0 \leq i < d} \left(c_i \cdot g^{-\frac{1}{d}} \cdot 2^{-g} + 1\right)\right)^{\frac{D_F}{d}} \quad (24)$$

with suitable parameters c_i which are all positive. This can be rewritten into a sum with suitable parameters c_i' which are also positive:

$$P_{\text{refinement}} = \left(\sum_{0 \leq i \leq d} c_i' \cdot g^{-\frac{i}{d}} \cdot 2^{-gi}\right)^{\frac{D_F}{d}} \quad (25)$$

If we differentiate the refinement probability according to the chain rule, and the product rule, we obtain the following result:

$$\frac{\partial}{\partial g} P_{\text{refinement}} = \frac{D_F}{d} (\textstyle\sum \dots)^{\frac{D_F}{d} - 1} \cdot \left(\sum_{0 \leq i < d} s_i\right) \quad (26)$$

with $s_i = -\frac{c_i' \cdot i}{d} \cdot g^{-\frac{i}{d} - 1} \cdot 2^{-gi} - c_i' \cdot gi \cdot g^{-\frac{i}{d}} \cdot 2^{-gi-1}$.

As all s_i are obviously negative, the derivative as such is negative, and the refinement probability is monotonically decreasing. If we apply again chain rule and product rule to build a second derivative $(\partial^2/(\partial^2 g))P_{\text{refinement}}$, all summands are again turned into positives. Therefore, the second derivative of $P_{\text{refinement}}$ is positive and its first derivative is monotonically increasing.

The cost for the scan of the first-level directory is obviously linear in the number of data pages. For the cost of optimized processing of the second-level directory, we cannot state anything about the behavior because sometimes splitting is beneficial to query processing, sometimes not.

3.5. Optimal quantization

In the previous section, we developed the cost formulas for access cost on the three directory levels which are now used for optimization. We showed that these cost functions fulfil monotonicity properties facilitating a local optimization.

We are given an initial set of P partitions p_i, $0 \le i < P$, such that each partition could be stored using a 1-bit representation. Now, each partition could either be split into two sub-partitions or could be stored using the 1-bit representation. If we decide to split, we can store each sub-partition with a 2-bit representation or we can choose to split either (or both) sub-partitions further. This process recursively continues until we reach the 32-bit representation i.e., the exact representation. If one records the whole process, one gets a binary tree, the split-tree, rooted at the initial partitions. Figure 5 depicts this scenario. A solution to the partitioning problem is defined as follows:

Definition 1. (Solution)

A solution to the partitioning problem is a set S of partitions such that for all leaves x in all split trees there exists exactly one node $s \in S$ such that the path from x to the root node of x contains s.

In other words, if we want to have a valid solution and we decide to split a partition, then both of the sub-partitions must be quantized and stored on disk, or further split. The optimal solution is the solution that picks nodes such that estimated query cost is minimized. More formally:

Definition 2. (Optimal Solution)

A solution O is defined to be optimal, if the total query cost as predicted by our cost model are minimal among all solutions:

$\forall S$: estimated_cost(O) ≤ estimated_cost(S) .

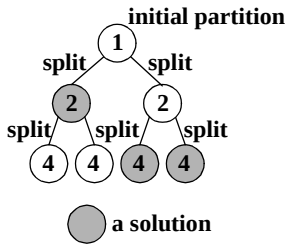

Figure 5: Split or quantize?

The choice of stopping point along one path from the root is not independent of choices made along other paths. Therefore it appears at first as if we have no choice but to test all the potential solutions. As there are 458,330 potential solutions how to quantize a single initial partition, we get the very large number of $458{,}330^P$ potential solutions to test. Obviously, this naive algorithm is not feasible.

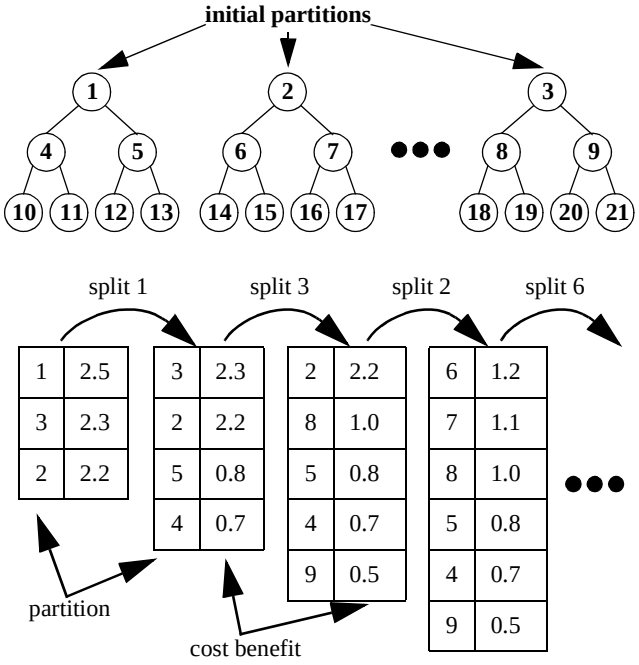

Figure 6: Ordering of partitions

However, taking more knowledge about the specific cost model into account, we can do better: Some part of the cost (or cost savings) implied by a split is independent from which particular partition is split. Separating this out, the total cost/benefit of a split can be considered the sum of

- constant cost (equal for each partition)
- variable cost (varying for each partition)

Which properties of these cost benefits/disadvantages do we know? As in the constant part of the cost, a complex cost model is involved, we cannot assume any properties. However, we know that the variable cost for the refinement step decreases monotonically if we split, and that this decrease itself decreases. In other words, the first split of a partition bears more cost savings than the following split which in turn has a higher cost savings than its succeeding split. This allows us to construct an algorithm that in any step simply splits the one partition that has the largest cost benefit with respect to variable cost. As all partitions share the same constant cost disadvantage, it is guaranteed that no other partition will lead to the global optimum. In the following, we will present the details of our algorithm and prove that it is optimal.

Our optimization algorithm starts from a set of initial partitions provided by the construction algorithm. Then, using the cost model, we order these partitions according to their cost benefit (variable cost) and store them in a sorted list. We can do so because all partitions share the constant cost and vary only in the variable cost. We split the partition and get two sub-partitions that we insert at the appropriate place in the list. We pick again the top element of the list and split it, and so on. Unfortunately, we are not allowed to stop the process if the variable cost benefit induced by a split operation is less than the constant cost disadvantage. This is due to the fact that we do not know the behavior of the con-

stant cost. It actually might be the case that for a split there is less variable benefit than constant gain, however, if we further split, this ratio flips. In other words, there may exist several local optima that differ from the global optimum. Therefore, we have to continue splitting until we reach the 32-bit representation. In each step of the algorithm we record the expected total query cost according to our cost model. Finally, we simply pick the optimum solution and undo all useless split operations. This algorithm reduces the computational cost from $458{,}330^P$ test-and-partition operations to $32 \cdot P$ operations, which is exactly the cost to build a regular hierarchical index. More formally, we get the following algorithm:

```
optimal_partitioning(InitialSetOfPartitions SP) {
    ListOfPartitions LP, opt_LP;
    Partition p, p1, p2;
    float total_cost, minimal_cost;
    LP := SP.sort_according_to_variable_cost_benefit();
    minimal_cost := ∞;
    while (!LP.empty()) {
        // select the partition p having maximal cost benefit
        p = LP.get_top_element();
        // check if p fits into a page using 32-bit representat.
        if (p.fits(32))
            p = p.set_cost_benefit(-∞);
        else
            LP = LP.remove(p);
        p.split(&p1, &p2);
        p1.determine_benefits();
        LP.insert_and_resort(p1);
        p2.determine_benefits();
        LP.insert_and_resort(p2);
        total_cost = LP.total_cost();
        if (total_cost < minimal_cost) {
            opt_LP := LP
            minimal_cost := total_cost
        }
    }
    // undo all splits until the state recorded in opt_LP
    ...
}
```

3.6. Proof of optimality

As we already mentioned, our algorithm leads to an optimal solution of the partitioning problem according to Definition 1 and 2[1]. We will prove this fact in the rest of the section. This will be done in two steps: We first show that among all solutions to the problem resulting in l partitions, our solution bears minimal cost. Then, we show that our algorithm is optimal.

Lemma 1. In order to determine which of two solutions s_1 and s_2 both comprising l $(P \le l \le (32 \cdot P))$ partitions bears less cost it is sufficient to compare the variable cost of s_1 and s_2.

Proof. The constant cost only depend on the number of partitions of a solution. As s_1 and s_2 share the same constant cost, we only have to consider the variable cost to determine which of the solutions bears less cost. ***q.e.d.***

1. We only claim that our algorithm finds an optimal solution with respect to a given cost model and a given split algorithm.

Lemma 2. Among all solutions resulting in exactly l $(P \le l \le (32 \cdot P))$ partitions, our proposed solution leads to minimal cost.

Proof. Let O be the solution computed by our algorithm. Assume there exists a solution S which is different of O and leads to less cost. According to Lemma 1 this means that S has a higher variable cost benefit than O. According to the properties of the variable cost, nodes that are higher in the split tree have a higher cost benefit than their subnodes. In other words, the first split of a partition bears more cost savings than the next which in turn bears more cost savings than its successor. Therefore, if the split of a partition bears a cost benefit of b, no subpartition can have a benefit greater than b. As in any step of our algorithm, we split the partition that has maximal cost benefit, and as no sub-partition can have a higher cost benefit than its parent, we are guaranteed to pick in each step the set of nodes that has a maximal total cost benefit. Thus, there cannot exist a partition that is not in our solution but has a higher cost benefit than any partition included in our solution. Therefore, a different solution S can only have less cost benefit. Thus, our solution is optimal. ***q.e.d.***

Theorem 1. The solution of our algorithm is optimal.

Proof. As we test all values of l, $(P \le l \le (32 \cdot P))$, and for each value, we generate the optimal solution having l partitions, we are guaranteed to pick the optimal solution. ***q.e.d.***

An index structure must allow dynamic updates, and the IQ-tree is no exception. The actual insert and delete algorithms are standard, and not described here. The one new twist is that we may need to do some additional work to continue to maintain optimality of our representation. For instance, when an update modifies the variable cost for a page, it may turn out to be preferable to undo the split for this page, and to split a different page instead. Similarly, when an insert causes a data page overflow, we have to decide whether to split the page or to quantize it at coarser granularity. The technique is conceptually simple, though some careful book-keeping is required. Details are omitted here for lack of space.

4. Experimental evaluation

In this section, we present an extensive experimental evaluation of the IQ-tree using data sets which are typical for a broad range of applications:

- uniform data of varying dimensionality (UNIFORM)
- 16-dimensional data from a CAD application. Each point represents the Fourier coefficients of the curvature of a CAD object (CAD)
- 16-dimensional data containing color histograms of a set of pixel images (COLOR)
- 9-dimensional data describing the weather on a number of stations distributed around the world (WEATHER)

For each experiment we separated from database a set of query points, thus not contained in the database, but following the distribution of the respective data set. Then, the performance of each technique was measured by the average total time (in seconds) over all these query points for a nearest neighbor search. The experiments were performed on HP 9000/780 workstations running HP-UX 10.20.

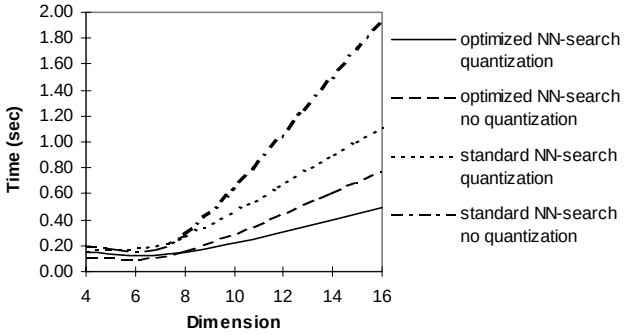

Figure 7: Experiment on UNIFORM (500.000 data points, varying the dimension)

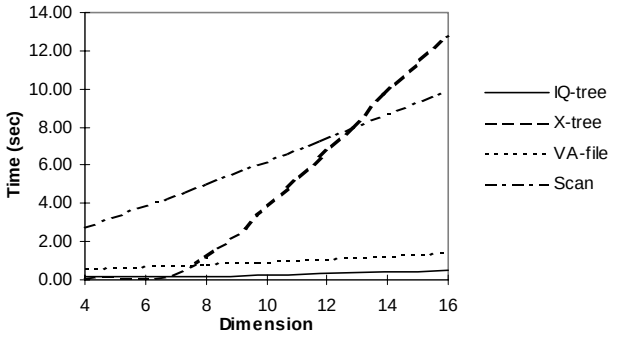

Figure 8: Experiment on UNIFORM (500.000 data points, varying the dimension)

4.1. Impact of the particular concepts

In the first set of experiments, we used uniform data of varying dimensionality to demonstrate the impact of the different concepts introduced for the IQ-tree. For that purpose, we also implemented reduced versions of the IQ-tree: one without quantization, one without optimized page access strategy, and a version implementing none of these concepts.

The results are depicted in figure 7. As expected, quantization of the data points pays off only for dimensions larger than 8 whereas our optimized page access strategy for a nearest neighbor search improves the performance for all dimensions, with the performance gain increasing with growing dimension. For high-dimensional data however, we can roughly say that each of the concepts, i.e. quantization and optimized nearest neighbor search, contribute equally to the performance of the IQ-tree.

4.2. Performance comparison

For comparison we use two well known techniques for nearest neighbor search in high-dimensional spaces: the VA-file [20] as the best known compression technique and the X-tree [6] as the best known index structure for that purpose. For the VA-file a number of bits per dimension must be specified to determine the compression rate. Therefore, we first tested the VA-file with different numbers of bits per dimension (between 2 and 8) and then selected the compression rate for which the VA-file performed best. We also implemented the sequential scan as a reference technique.

First, we applied the VA-file, the X-tree and the sequential scan to the UNIFORM data set used in section 4.1 to see how they perform with increasing dimension of the data space, compared to the IQ-tree. Figure 8 depicts the results.

For dimensions smaller than 8 the performance of the X-tree and the IQ-tree is nearly indistinguishable. In such data sets both trees outperform the VA-file and the sequential scan. With increasing dimension, however, the performance of the X-tree degenerates and becomes even worse than the performance of the sequential scan for dimensions larger than 12. This effect does not occur for the IQ-tree and the VA-file. They both perform well for high-dimensional data due to their data compression technique. In this setting, the IQ-tree performs up to 26 times faster than the X-tree and up to 3 times faster than the VA-file.

As expected, the performance of the IQ-tree is close to the performance of the X-tree for lower dimensions and shows a rather constant improvement of the performance of the VA-file when increasing the dimension of the data space. Note that the runtime for the IQ-tree is about 3 times faster than the runtime of the VA-file even for 16-dimensional uniformly distributed data points. This is due to the fact that the IQ-tree does not only utilize quantization, but additionally has a tree structure from which it can benefit even if the selectivity of the tree decreases.

To verify this fact we conducted another set of experiments using our 16-dimensional UNIFORM data set, now varying the number of objects in the database. Figure 9 shows the results.

As before, the IQ-tree and the VA-file beat the X-tree and the sequential scan by at least an order of magnitude. The IQ-tree runs faster than the VA-file for this data set by a factor of 1.6 up to 3, and this factor increases when increasing the number of objects in the database. This can be explained by the fact that even in this setting of 16-dimensional uniformly distributed data points, the IQ-tree has at least *some* selectivity with respect to the queries. Therefore the cost for a nearest neighbor search increases slower for the IQ-tree than for the VA-file.

The above experiments on uniformly distributed data sets confirm the claimed properties of the IQ-tree in a setting which is well understood but which may be of limited practical impact. Therefore, we performed similar experiments using our real world data sets, CAD, COLOR and

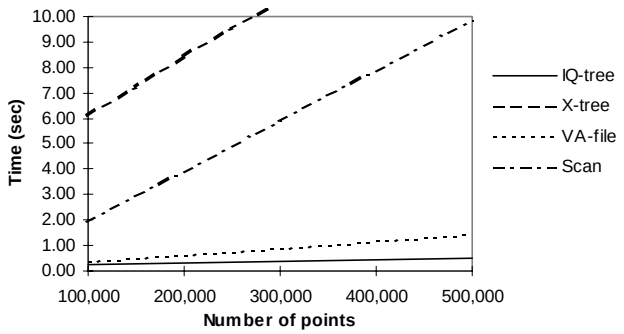

Figure 9: Experiment on UNIFORM (16 dimensions, varying the number of points)

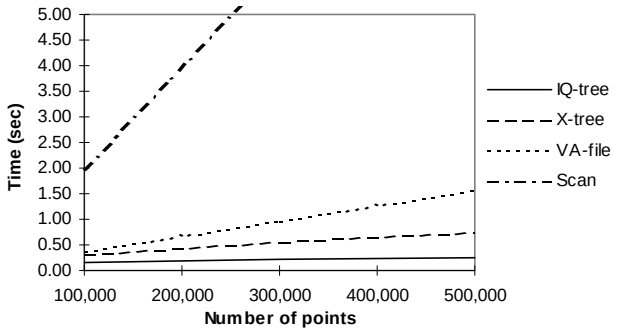

Figure 10: Experiment on CAD
(16 dimensions, varying the number of points)

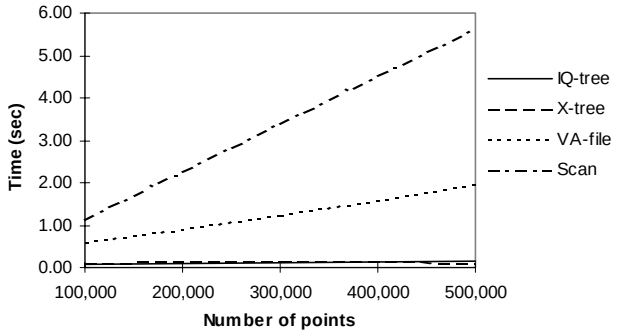

Figure 12: Experiment on WEATHER
(9 dimensions, varying the number of points)

WEATHER described above. In all experiments we varied the number of objects and not the dimension, since the dimension of a real world data set is fixed.

The results for the CAD data set are depicted in figure 10. In this data set the points are moderately clustered, and as a consequence, the X-tree performs well on this data set. It runs up to 2 times faster than the VA-file although the dimension of the data set is high. The IQ-tree, on the other hand, even beats the X-tree. The IQ-tree runs up to 3 times faster than the X-tree and up to 5 times faster than the VA-file. The runtime of the sequential scan is out of question.

Figure 11 shows the results for the COLOR data set which is only very slightly clustered. Again, the IQ-tree performs best compared to the other techniques: up to 2.6 times faster than the VA-file and up to 6.6 times faster than the X-tree. It is also worth noting that, although the dimension of the data is high and the data set is only slightly clustered, the X-tree performs better than the sequential scan.

In the last set of experiments, the WEATHER data set was used. This data set is highly clustered and also has a rather low fractal dimension. As we can see in figure 12 the results for this data set clearly show the benefits of a hierarchical indexing scheme. The performance of the X-tree and the IQ-tree are nearly the same for this data set. Both indexing techniques outperform the VA-file up to a factor of 11.5 which again increases when increasing the number of points in the database.

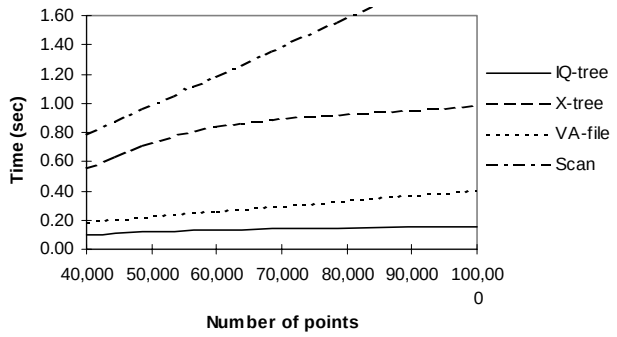

Figure 11: Experiment on COLOR
(16 dimensions, varying the number of points)

To summarize, the IQ-tree outperforms all competing techniques because it combines the “best of both worlds (indexes and compression techniques)”, i.e. it benefits from a tree structure and also implements a local quantization technique which is applied to the data pages of the tree.

Note again, that for each data set we *manually* determined the optimal number of bits per dimension for the VA-file. That means the performance of the VA-file could be even worse than depicted if a non-optimal number of bits is selected. A main advantage of our technique, in contrast, is that it *automatically* adapts the compression rate to the characteristics of the respective data set.

5. Related work

In the recent literature, a variety of index structures suitable for high-dimensional data spaces have been proposed. Most of the work extended existing index structures that have been proposed for geographic applications such as the R-tree [11] or the K-D-B-tree [18].

Lin, Jagadish and Faloutsos presented the TV-tree [16] which is an R-tree-like index structure. The central concept of the TV-tree is the telescope vector (TV). Telescope vectors divide attributes into three classes: attributes which are common to all data items in a subtree, attributes which are ignored and attributes which are used for branching in the directory. The major drawback of the TV tree is that information about the behavior of single attributes, e.g. their selectivity, is required.

Another R-tree-like high-dimensional index structure is the SS-tree [21] which uses spheres instead of bounding boxes in the directory. Although the SS-tree clearly outperforms the R*-tree, spheres tend to overlap in high-dimensional spaces.

The SR-tree, a variant of the SS-tree has been proposed by Katayama and Satoh in [15]. The basic idea of the SR-tree is to use both hyperrectangles and hyperspheres as an approximation in the directory. Thus, a page region is described by the intersection of both objects.

In [14], Jain and White introduced the VAM-Split R-tree and the VAM-Split KD-tree. VAM-Split trees are rather similar to KD-trees, however in contrast to KD-trees, split dimensions are not chosen in a round robin fashion but depending on the maximum variance. VAM Split trees are built in main memory and then stored to secondary storage.

Therefore, the size of a VAM-Split tree is limited by the available main memory.

In [6], the X-tree has been proposed which is an index structure adapting the algorithms of R*-trees to high-dimensional data using two techniques: First, the X-tree introduces an overlap-free split algorithm which is based on the split history of the tree. Second, if the overlap-free split algorithm would lead to an unbalanced directory, the X-tree omits the split and the according directory node becomes a so-called supernode. Supernodes are directory nodes which are enlarged by a multiple of the block size.

A further approach, the pyramid technique [5], is based on a one-dimensional transformation by partitioning the data space into pyramids, meeting at the center of the data space. This method accelerates hypercube range queries, and is, under some conditions, not subject to the '*dimensionality curse*'.

Most recently, the VA-file [20] was developed, an index structure that actually is not an index structure. The authors prove in the paper that under certain assumptions, above a certain dimensionality no index structure can process a nearest neighbor query efficiently. Thus, they suggest to speed-up the sequential scan rather than trying to fight a war that they say is already lost. The basic idea of the VA-file is to keep two files: a bit-compressed (quantized) version of the points and the exact representation of the points. Both files are unsorted, however, the ordering of the points in the two files is identical. Query processing is equivalent to a sequential scan of the compressed file with some look-ups to the second file whenever this is necessary. In particular a look-up occurs, if a point cannot be pruned from the nearest neighbor search only based on the compressed representation.

6. Conclusions

In this paper, we presented the IQ-tree, a new index compression technique for high-dimensional data spaces. The IQ-tree makes use of data compression techniques and involves these techniques in a regular multi-dimensional index structure. The technical challenge is to determine an optimal compression rate for each page. We proposed an algorithm that computes such an optimal quantization and we proved the optimality of our algorithm. Furthermore, we presented a new page scheduling strategy for nearest neighbor algorithms that, based in a new cost model, can avoid many random seeks.

To demonstrate the practical impact of our technique and also to empirically show the superiority of the IQ-tree, we ran a variety of experiments on various synthetic and real data sets. Our experiments show a large performance gain compared to the X-tree and the VA-file as representatives of both approaches we unified in the IQ-tree. This holds for any configuration we investigated. The maximum speed-up observed in our experiments was 26 compared to the X-tree and 11.5 compared to the VA-file. Thus, the IQ-tree combines the advantages of hierarchical search and a fast linear search. It automatically adapts to both situations, showing a better overall behavior than competing structures for low-, medium, and high-dimensional data spaces.

Acknowledgments

We would like to thank Florian Krebs for implementing our ideas very efficiently.

References

1. Arya S.: '*Nearest Neighbor Searching and Applications*', Ph.D. thesis, University of Maryland, 1995.
2. Belussi, A., Faloutsos, C.: '*Estimating the Selectivity of Spatial Queries Using the "Correlation Fractal Dimension"*', Proc. of 21st. Int. Conf. on Very Large Databases (VLDB), 1995.
3. Berchtold S., Böhm C., Keim D., Kriegel H.-P.: '*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*', ACM PODS Symposium on Principles of Database Systems (PODS), 1997.
4. Berchtold S., Böhm C., Kriegel H.-P.: *'Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations'*, 6th. Int. Conf. on Extending Database Technology (EDBT), 1998.
5. Berchtold S., Böhm C., Kriegel H.-P.:*'The Pyramid Technique: Towards Breaking the Curse of Dimensionality'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1998
6. Berchtold S., Keim D., Kriegel H.-P.: '*The X-Tree: An Index Structure for High-Dimensional Data*', Proc. 22nd Int. Conf. on Very Large Databases (VLDB), 1996, pp. 28-39.
7. Beyer K., Goldstein J., Ramakrishnan R., Shaft U.: '*When is "Nearest Neighbor" meaningful?*', Proc. Int. Conf. on Database Theory (ICDT), 1999.
8. Böhm C.: *'Efficiently Indexing High-Dimensional Data Spaces'*, Ph.D. thesis, University of Munich, 1998.
9. Faloutsos C., Kamel I.: '*Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension'*, Proc. Symp. on Principles of Database Systems (PODS), 1994, pp. 4-13.
10. Friedman J. H., Bentley J. L., Finkel R. A.: '*An Algorithm for Finding Best Matches in Logarithmic Expected Time*', ACM Transactions on Mathematical Software, Vol. 3, No. 3, 1977.
11. Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984, pp. 47-57.
12. Henrich, A.: '*The LSD^h-tree: An Access Structure for Feature Vectors*', Proc. Int. Conf. on Data Engineering (ICDE), 1998.
13. Hjaltason G. R., Samet H.: '*Ranking in Spatial Databases*', Proc. 4th Int. Symp. on Large Spatial Databases (SSD), 1995.
14. Jain R, White D.A.: *'Similarity Indexing: Algorithms and Performance'*, Proc. SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670, 1996, pp. 62-75.
15. Katayama N., Satoh S.: *'The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997.
16. Lin K., Jagadish H. V., Faloutsos C.: *'The TV-Tree: An Index Structure for High-Dimensional Data'*, VLDB Journal, Vol. 3, pp. 517-542, 1995.
17. Pagel B.-U., Six H.-W., Toben H., Widmayer P.: *'Towards an Analysis of Range Query Performance in Spatial Data Structures'*, Proc. Symp. on Principles of Database Systems (PODS), 1993, pp.214-221.
18. Robinson, J.T.: '*The K-D-B-Tree: A Search Structure for Large Multi-Dimensional Dynamic Indexes*', Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 10-18, 1981.
19. Seeger B., Larson P.A., McFayden R.: *'Reading a Set of Disk Pages'*, Proc. 19th Int. Conf. on Very Large Data Bases (VLDB), 592-603, 1993.
20. Weber R., Schek H.-J., Blott S.: *'A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*', Proc. Int. Conf. on Very Large Databases (VLDB), 1998.
21. White D.A., Jain R.: *'Similarity indexing with the SS-tree'*, Proc. 12th Int. Conf on Data Engineering (ICDE), 1996.