Dynamically Optimizing High-Dimensional Index Structures

Christian Böhm and Hans-Peter Kriegel

University of Munich, Oettingenstr. 67, 80538 München, Germany
{boehm,kriegel}@dbs.informatik.uni-muenchen.de

Abstract. In high-dimensional query processing, the optimization of the logical page-size of index structures is an important research issue. Even very simple query processing techniques such as the sequential scan are able to outperform indexes which are not suitably optimized. Page-size optimization based on a cost model faces the problem, that the optimum not only depends on static schema information such as the dimension of the data space but also on dynamically changing parameters such as the number of objects stored in the database and the degree of clustering and correlation in the current data set. Therefore, we propose a method for adapting the page size of an index dynamically during insert processing. Our solution, called DABS-tree, uses a flat directory whose entries consist of an MBR, a pointer to the data page and the size of the data page. Before splitting pages in insert operations, a cost model is consulted to estimate whether the split operation is beneficial. Otherwise, the split is avoided and the logical page-size is adapted instead. A similar rule applies for merging when performing delete operations. We present an algorithm for the management of data pages with varying page-sizes in an index and show that all restructuring operations are locally restricted. We show in our experimental evaluation that the DABS tree outperforms the X-tree by a factor up to 4.6 and the sequential scan by a factor up to 6.6.

1. Motivation

Query processing in high-dimensional data spaces is an emerging research domain which gains increasing importance by the need to support modern applications by powerful search tools. In the so-called non-standard applications of database systems such as multimedia [16, 33, 34], CAD [11, 13, 21, 25], molecular biology [26, 29], medical imaging [27], time series analysis [1, 2, 18], and many others, similarity search in large data sets is required as a basic functionality.

A technique widely applied for similarity search is the so-called feature transformation, where important properties of the objects in the database are mapped into points of a multidimensional vector space, the so-called feature vectors. Thus, similarity queries are naturally translated into neighborhood queries in the feature space.

In order to achieve a high performance in query processing, multidimensional index structures [20] are applied for the management of the feature vectors. Even a number of specialized index structures for high-dimensional data spaces have been proposed [6,

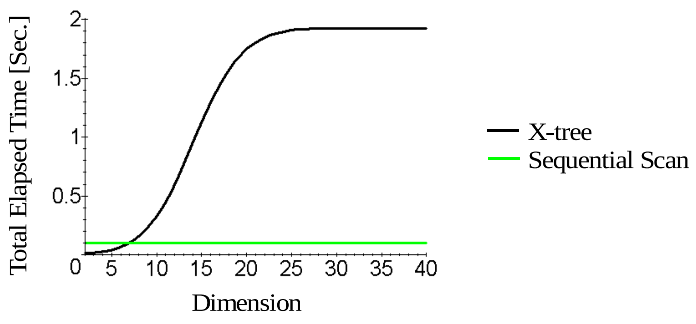

Fig. 1: Performance of query processing with varying dimension.

12, 22, 28, 30, 36]. In spite of these efforts, there are still high-dimensional indexing problems under which even specialized index structures deteriorate in performance. To understand the effects leading to this so-called 'curse of dimensionality', a variety of methods for estimating the performance of query processing has been developed [3, 4, 9, 10, 14, 15, 17, 19, 31, 32, 35]. These cost models can be used for optimization. High-dimensional query processing techniques offer various parameters for optimization such as dimension reduction [8, 16, 30, 34] or the accuracy of the representation of the features [6, 37].

In recent years, a general criticism on high-dimensional indexing has come up. Most multidimensional index structures have an exponential dependency (with respect to the time for processing range queries and nearest neighbor queries) on the number of dimensions. To illustrate this, fig. 1 shows our model prediction of the processing time of the X-tree for a uniform and independent data distribution (constant database size 400 KBytes). With increasing dimension *d*, the processing time grows exponentially until saturation comes into effect, i.e. a substantial ratio of all index pages is accessed. In very high dimensions $d \geq 25$, virtually all pages are accessed, and the processing time approaches thus an upper bound.

In recognition of this fact, an alternative approach is simply to perform a sequential scan over the entire data set. The sequential scan causes substantially fewer effort than processing all pages of an index, because the reading operations in the index cause random seek operations whereas the scan reads sequentially. The sequential scan rarely causes disk arm movements or rotational delays which are negligible under these circumstances. Assuming a logical block size of 4 KBytes, contiguous reading of a large file is by a factor >12 faster than reading the same amount of data from random positions (cf. [14, 37]).

A second advantage of the sequential scan over index-based query processing is its storage utilization of 100%. In contrast, index pages have a storage utilization between 60% and 70% which causes a further performance advantage of about 50% for the sequential scan when reading the same amount of data. The constant cost of the sequential scan is also depicted in fig. 1. The third advantage of the sequential scan is the lacking

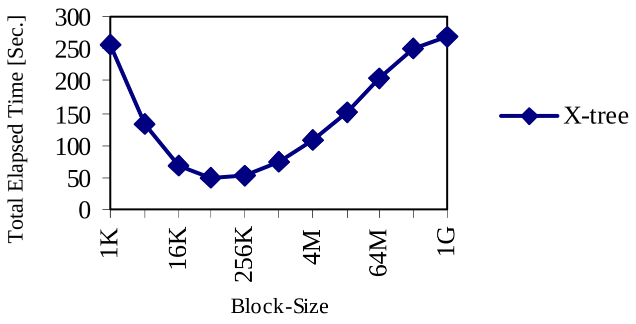

Fig. 2: Block size optimization.

overhead of processing the directory. We can summarize that the index may not access more than 5% of the pages in order to remain competitive with the sequential scan.

In fig. 1, the break-even point of the two techniques is reached at $d = 7$. The trade-off between the two techniques, however, is not simply expressed in terms of the number of dimensions. For instance when data sets are highly skewed (as real data sets often are), index techniques remain more efficient than a scan up to a fairly high dimension. Similarly, when there are correlations between dimensions, index techniques tend to benefit compared to scanning. Obviously, the number of data objects currently stored in the database plays an important role since the sequential scan is linear in the number of objects whereas query processing based on indexes is sub-linear.

Fig. 2 shows the model predictions of the X-tree for 10,000,000 points uniformly and independently chosen from a 20-dimensional data space with varying block size from 1 KByte to 1 GByte. In this setting, the performance is relatively bad for usual block sizes between 1 KBytes and 4 KBytes, quickly improving when increasing the block size. A broad and stable optimum is reached between 64 KBytes and 256 KBytes. Beyond this optimum, the performance deteriorates again. This result shows that block size optimization is the most important advice to improve high-dimensional indexes.

The rest of this paper is organized as follows: Section 2 explains the general idea and an overview of our technique. Section 3 shows the architectural structure of the DABS-tree. The following sections show how operations such as insert, delete and search are handled. In section 6, we show how our model developed in [9, 14] can be applied for a dynamic and independent optimization of the logical block size. Finally, we present an experimental evaluation of our technique.

2. Basic Idea

As we pointed out in section 1, there are three disadvantages for query processing based on index structures compared to the sequential scan:

- data is read in too small portions
- index structures have a substantially lower storage utilization
- processing of the directory causes overhead

In this paper, we will present the DABS-tree (Dynamic Adaptation of the Block Size) which tackles all three problems. We propose a new index structure claiming to outperform the sequential scan in virtually every case. In dimensions where index-based techniques are superior to the sequential scan, the efficiency of these techniques is retained unchanged. In an area of moderate dimensionality, both approaches, conventional indexes as well as the scan, are outperformed.

The first problem is solved by a suitable page-size optimization. As we face the problem that the actual optimum of the logical block size is dependent on the number of objects currently stored in the database and on the data distribution (which may also change over time), the block size has to be adapted dynamically. After a page has been affected by a certain number of inserts or deletions, the page is checked whether the number of points currently stored in the page is close enough to the optimum. Otherwise, the page is split or a suitable partner is sought for balancing or merging.

This means that pages with different logical block size are at the same time stored in the index. Although a constant block size facilitates management, no principal problem arises when sacrificing this facilitation. To solve the second problem, storage utilization, we propose to allow continuously growing block sizes, i.e. we also give up the requirement that the logical block size is a multiple of some physical block size or a power of two or the demand that the block size is only changed by doubling or division by two. Instead, every page has exactly the size which is needed to store its current entries. When an entry is inserted to a page, the block size increases, and the page must usually be stored to a new position in the index file. To avoid fragmentation of the file, we propose garbage collection.

The third problem, directory overhead, cannot be completely avoided by our technique since we do not want to cancel the directory. The directory overhead, however, is weakened, because we simplify the directory. Instead of a hierarchical directory, we only maintain a linear single-level directory which is sequentially scanned. The block size optimization also helps to reduce the directory overhead, because this overhead is taken into account by the optimization.

3. Structure of the DABS-Tree

The structure of the DABS-tree is depicted in fig. 3. Each directory entry contains the following information: The page region in form of a minimum bounding rectangle, the reference (i.e. the background storage address) to the page and additionally the number of entries currently stored in the page. The number of entries is also used to determine the corresponding block size of a data page before loading.

The directory consists simply of a linear array of directory entries. We intentionally cancel the hierarchically organized directory, because the efficiency of query processing is not increased by hierarchies but rather decreased. We confirm this effect by the following consideration:

In our experiment presented in section 1 (cf. fig. 2), we determined an optimum block size of 64 KBytes. For 10,000,000 data points in a 20-dimensional space, we need 20,000 data pages to store the points. Using a hierarchical directory, we need 78 index

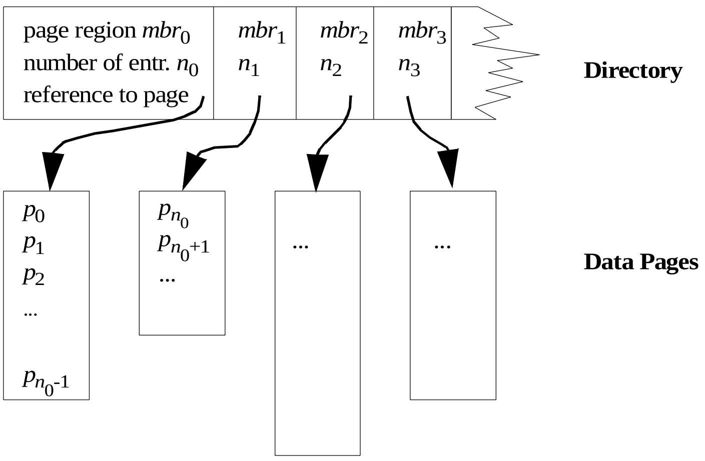

Fig. 3: Structure of the DABS-tree.

pages at the first directory level and the root-page. Even if we assume no overlap among the directory pages, query processing requires an average of 44 directory page accesses. The cost for these accesses are 1.14 seconds of I/O time. A sequential scan of a linear directory, however, requires 0.71 seconds. Both kinds of directory cost are negligible compared to 49 seconds of cost for accessing the data pages. Even though, the sequential scan of a linear directory causes fewer effort than a hierarchical directory. This observation even holds for fairly low dimensions.

The data pages contain only data-specific information. Besides the point data and eventually some additional application-specific information, no management information is required. The data pages are stored in random order in the index file. Conventional index structures usually do not utilize the space in the data pages to 100% in order to leave empty space for future insert operations. In contrast, the DABS-tree stores the data pages generally without any empty position inside a data page and without any gap between different pages. Whenever a new entry is inserted to a data page, the page is stored at a new position. The empty space in the file where the data page formerly used to be is passed to a free memory management. A garbage collection strategy is applied to build larger blocks of free memory, and, thus to avoid fragmentation (cf. section 5). Temporarily, the free blocks decrease the storage utilization of the index structure below 100%. The free blocks, however, are never subject to a reading operation during insert processing. Therefore, the performance of query processing cannot be negatively affected.

In order to guarantee overlap-free page regions, we hold additionally to the linear directory a kd-tree [5]. A kd-tree partitions the data space in a disjoint and overlap-free way. The page regions of the DABS-tree are always located inside a single kd-tree region. The kd-tree facilitates insert processing, because it offers unambiguously a data page for the insert operation. In contrast, the heuristics for choosing a suitable page in the X-tree cannot guarantee that no overlap occurs. The kd-tree is also used for the

```
Point DABS_nearest_neighbor_query (Point q) {
    typedef struct {float distance, int pageno, int num_objects} AplEntry ;
    AplEntry apl [number_of_pages]
    int i, j;
    Point cpc ;
    float pruning_dist = +infinity ;

    // First Phase
    DIRECTORY dir = read_directory () ;
    for (i = 0 ; i < number_of_pages ; i ++) {
        apl [i] . distance = mindist (q, dir [i] . mbr) ;
        apl [i] . pageno = dir [i] . pageno ;
        apl [i] . num_objects = dir [i] . num_objects ;
    }
    qsort (apl, number_of_pages, sizeof (AplEntry), cmp_float) ;

    // Second Phase
    for (i = 0 ; i < number_of_pages && apl [i] . distance < pruning_dist ; i ++) {
        Page p = LoadData (apl [i] . pageno, apl [i] .num_objects) ;
        for (j = 0 ; j < apl [i] . num_objects ; j ++)
            if (dist (q, p . object [j] . point) < pruning_dist) {
                cpc = p . object [j] . point ;
                pruning_dist = dist (q, p . object [j] . point) ;
            }
    }
    return cpc ;
}
```

Fig. 4: Algorithm for nearest neighbor queries.

merging operation which may be necessary due to delete operations, or because the optimal page size has increased on the basis of a changed data distribution. The kd-tree is not used for search.

4. Search in the DABS-Tree

Point queries and *range queries* are handled in a straightforward way. First, the directory is sequentially scanned. All data pages qualifying for the query (i.e. containing the query point or intersecting with the query range, respectively) are determined, loaded and processed.

Nearest neighbor queries and *k-nearest neighbor queries* are processed by a variant of the HS algorithm [24]. As the directory is flat, the algorithm can even be simplified, because the *active page list* (*APL*) is static in absence of a hierarchy. For hierarchically organized directories, query processing requires permanent insert operations to the APL, because in each processing step the pivot page is replaced by its child pages. Therefore, the APL must be re-sorted after processing a page.

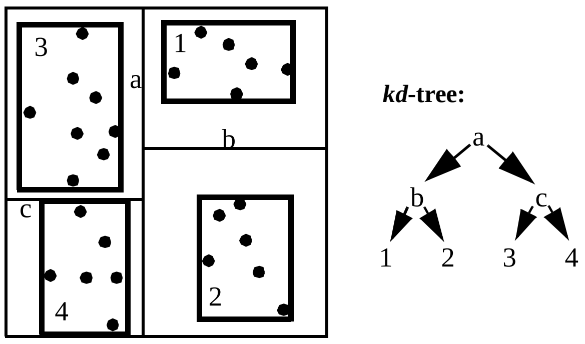

Fig. 5: The additional *kd*-tree.

In our case, the nearest neighbor algorithm works in two phases: The first phase scans the directory sequentially. During the scan, the distance between the query point and each page region is determined and stored in an array. Finally the distances in the array are sorted by the Quicksort algorithm, for instance [23]. In the second phase, the data pages are loaded and processed in the order of increasing distances. The closest point candidate determines the pruning distance. Query processing stops when the current page region is farther away from the query point than the closest point candidate. Fig. 4 depicts the algorithm for nearest neighbor queries. The *k-nearest neighbor* algorithm works analogously with the only difference that a *closest point candidate list* consisting of *k* entries is maintained and that the last entry in this list determines the pruning distance.

5. Handling Insert Operations

5.1 Searching the Data Page

To handle an insert operation, we search in the kd-tree, which is held in addition to the linear directory, for a suitable data page. The kd-tree has the advantage to partition the data space in a complete and disjoint fashion which makes the choice of the corresponding page unambiguous. Eventually, the MBR in the linear directory which is always located inside the corresponding kd-tree region (cf. fig. 5) must be slightly enlarged.

The page is loaded to the main memory, and the point is inserted. Usually, the page cannot be stored at its old position since we enforce a 100% storage utilization of pages. Therefore, it is appended to the end of the index file. The empty block at the former position of the page is passed to a free storage manager which performs garbage collections if the overall storage utilization of the index file decreases below a certain threshold value (e.g. 90%).

Note that in contrast to conventional index structures, the overall storage utilization can never decrease the efficiency of query processing, because empty parts of the index file are not subject to reading operations. By a low storage utilization, we only waste storage memory, but not processing time.

5.2 Free Storage Management

The free storage manager currently observes the storage utilization of the index file. When the storage utilization reaches some threshold value $su_{\min}$, the next new page is not appended to the end of the index file. Instead, a local garbage collection is raised which performs local restructuring of the file to collect empty pages as follows:

Let the size of the next page to be stored be *s*. The storage manager searches for the shortest interval of subsequent pages in the index file covering *s* Bytes of empty space. With a suitable data structure to organize the empty space, this search can be performed efficiently. Once the shortest interval with *s* Bytes of empty space is found, we load all pages in this interval to the main memory and restore them densely, thus creating a contiguous empty space of at least *s* Bytes. We store the new page to this space.

Now we will claim an important property of the restructuring action: Locality. We show that the size of the interval in the file which is to be restructured is bounded by the size *s* of the new page multiplied with some factor depending on the storage utilization $su_{\min}$.

Lemma 1. Locality of Restructuring

In an index file with a storage utilization $su \le su_{\min}$, there exists an interval with the length

$$l = \frac{s}{1 - su_{\min}}$$

containing at least *s* Bytes of free storage.

Proof (Lemma 1)

Assume that all intervals of the length *l* have less than *s* Bytes of free storage. Then, the number *e* of free Bytes in the file with length *f* is bounded by:

$$e < \frac{f}{l} \cdot s$$

By the definition of the storage utilization, we get the following inequation

$$su = 1 - \frac{e}{f} > 1 - \frac{s}{l} = su_{\min}$$

which contradicts the initial condition $su \le su_{\min}$.

❑

If we choose, for instance, a storage utilization of $su_{\min} = 50\%$, Lemma 1 tells us that restructuring is bounded to an interval twice as large as the size *s* of the page we want to store. For a storage utilization of $su_{\min} = 90\%$, the interval is at most ten times as large as the new page. As there are no specific overflow conditions in our index structure, the pages are periodically checked by using a cost estimation whether they must be split. For the details, cf. section 6.

Deleting in the DABS-tree is straightforward. The point is deleted from the corresponding page and a small block is passed to the free storage manager. If the storage utilization falls below the threshold $su_{\min}$, a local restructuring action is raised for the last data page in the file. Since there is no clear underflow condition in the DABS-tree, the pages are periodically tested by using a cost model whether they are to merge.

6. Dynamic Adaptation of the Block Size

In this section, we will first show the dynamic adaptation from an algorithmic point of view. Then, we will show how the cost model developed in [9, 14] is modified and applied to take split and merging decisions, respectively.

6.1 Split and Merge Management

Basically, it is possible to evaluate the cost model after every insert or delete operation and to determine whether a page must be split or merged with some neighbor. This is, however, not very economic, because the optimum is generally broad. Therefore, we have to check rather seldom if the current page size still is close to the optimum.

We choose the following strategy: For each page, we have an update counter variable which is increased in each insert or delete operation the page is subject to. We perform our model evaluations when the value of the update counter reaches some user defined threshold which may be defined as a fixed number (e.g. 20 operations) or as a ratio of the current page capacity (e.g. 25% of the points in the page).

Note that it is theoretically possible (although not very likely) that pages must be merged after performing insert operations or that pages must be split after performing delete operations. This is not intuitive, as conventional index structures with a fixed block size know to split only after inserts and to merge after deletions. In our dynamic optimization, however, any of these operations can change the distribution of the data points and thus change the page size optimum into each direction.

Whenever the threshold of update operations is reached, a cost estimate for the current page with respect to query processing is determined. Then, some split algorithm is run tentatively. The page regions of the created pages are determined, and the query processing cost for the new pages is estimated. If the performance has decreased, the split is undone, and merging is tested in the same way.

A merging operation can only be performed if a suitable partner is available. In order to maintain overlap-free page regions, only two leaf pages with a common parent node in the kd-tree are eligible for merging. If the current page does not have such a counterpart, merging is not considered. Otherwise, the cost estimates for the two single pages and for the resulting page are determined and compared. If the performance estimate improves, the merge is performed. Finally, the relevant update counters are reset to 0.

6.2 Model Based Local Cost Estimation

For our local cost optimization, we must estimate how cost of query processing changes when performing some split or merge operation. Generally, we assume as reference query the nearest neighbor query with the maximum metric, because this assumption causes the lowest effort in the model computation. Practically, the difference in the page size optimum is low when changing the reference query to the Euclidean metric or to some *k*-nearest neighbor query.

In both cases, when taking a split or a merge decision, we compare the cost caused by one page with the cost caused by two pages with the half capacity. At the one hand, this action changes the accessing cost, because the transfer cost decreases with decreasing capacity. The access probability is also decreased by splitting. At the other hand, it is unpredictable whether the sum of the costs caused by the two smaller pages is really lower than the cost of the larger page.
Therefore, it is reasonable, to draw the following balance for the split decision:

$$\Delta_T = (t_{\text{Seek}} + C_1 \cdot t_{\text{Point}}) \cdot X_1 + (t_{\text{Seek}} + C_2 \cdot t_{\text{Point}}) \cdot X_2 - (t_{\text{Seek}} + C_0 \cdot t_{\text{Point}}) \cdot X_0 ,$$

where C_0 and X_0 are the capacity and the access probability of the larger page, and C_1 and C_2 (X_1 and X_2) the capacities (access probabilities) of the two smaller pages. The time t_{Point} is the transfer time for a point, i.e. $t_{\text{Point}} = t_{\text{transfer}} \cdot \text{sizeof (Point)}$. The time t_{Seek} denotes the delay time for a random access operation, subsuming the times for disk head movement and rotational delay. These times are hardware dependent. If the cost balance Δ_T is positive, the larger page causes fewer cost than the two smaller pages. In this case, a split should be avoided and a merge should be performed.

It is possible to estimate the access probability according to the cost estimates provided in [9, 14]. This approach, however, assumes no knowledge about the regions of the pages currently stored in the index. In our local cost optimization, the exact coordinates of the relevant page regions are known. Therefore, we can achieve higher accuracy if this information is considered. Additionally, it is possible to take into account the local exceptions in the data distribution.

First, we determine the fractal local point density according to the volume and the capacity of the larger page:

$$\rho_F = \frac{C_0}{V(\text{MBR}_0)^{D_F/d}}$$

Hereby, D_F denotes the fractal dimension of the data set [10]. From the local point density, we can derive an estimation of the nearest neighbor distance:

$$r = \frac{1}{2} \cdot \sqrt[D_F]{\frac{1}{\rho_F}}$$

Now, we are able to determine the Minkowski sum of the nearest neighbor query and the page region. If MBR_0 is given by a vector of lower bounds (lb_0, ... $lb_{d\text{-}1}$) and upper bounds (ub_0, ... $ub_{d\text{-}1}$), the Minkowski sum is determined by:

$$V_{R \oplus C}(\text{MBR}_0, 2r) = \prod_{0 \le i < d} (ub_i - lb_i + 2r)$$

This Minkowski sum can be explicitly clipped at the data space boundary (here for simplicity assumed to be the unit hypercube):

$$V_{(R \oplus C) \cap DS}(\text{MBR}_0, 2r) = \prod_{0 \le i < d} (\min\{ub_i + r, 1\} - \max\{lb_i - r, 0\})$$

We assume that the query distribution follows the data distribution. Therefore, the access probability X_0 corresponds to the ratio of points in the Minkowski sum with respect to all points in the database:

$$X_0 = \frac{\rho_F}{N} \cdot V_{(R \oplus C) \cap DS}(\mathrm{MBR}_0, 2r)^{D_F/d}$$

Analogously, the access probabilities for the smaller pages X_1 and X_2 are determined by their page regions MBR_1 and MBR_2. The access probabilities are used in the cost balance for taking split or merge decisions.

6.3 Monotonicity Properties of Splitting and Merging

The most important precondition for the correctness of a local optimization is the monotonicity of the first derivative of the cost function with respect to the page capacity. If the first derivative is not monotonically increasing, the cost function may have various local optima where the optimization easily could get caught in.

As depicted in fig. 2, the cost function indeed forms a single local optimum which is also the global optimum. Cost are very high for block sizes which are either too small or too large. Minimum cost arise in a relatively broad area between these extremes.

Under several simplifying assumptions, it is also possible to prove that the derivative of the cost function is monotonically increasing. From this monotonicity, we can conclude that there is at most one local minimum. The assumptions required for this proof are uniformity and independence as well as neglecting boundary effects. For this simplified model

$$T(C) = \left(\sqrt[d]{\frac{1}{C}} + 1\right)^d \cdot \left(t_{\mathrm{Seek}} + \frac{C}{\mathrm{sizeof(point)}} \cdot t_{\mathrm{transfer}}\right),$$

it is possible to show that the second derivative of the cost function is positive:

$$\frac{\partial^2}{\partial C^2} T(C) \geq 0 .$$

The intermediate results in this proof, however, are very complex and thus not presented here.

7. Experimental Evaluation

To demonstrate the applicability and the practical relevance of our technique, we performed an experimental evaluation on both, synthetic and real data. The improvement potential was already shown in fig. 2 where a clear optimum for page sizes was found at 64 KBytes outperforming the X-tree with a standard page of 4K by a factor of 2.7 and the sequential scan by a factor of 3.6.

The intention of our next experiment is to show that the optimum is not merely a hardware constant but to a large extent dependent on the data to be indexed. For this purpose, we constructed a DABS-tree on several data files containing uniformly and independently distributed points of varying dimension. The number of objects was fixed in this experiment to 12,000. We observed the block size which was generated by the

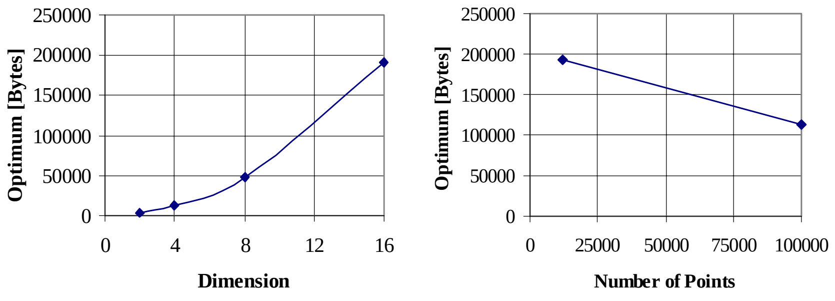

Fig. 6: Optimal block size for uniform data.

local optimization. The results are depicted on the left side of fig. 6. In the two-dimensional case, quite a usual block size of 3,000 Bytes was found to be optimal. In the high-dimensional case, however, the optimum block size reaches values up to 192 KBytes with even increasing tendency.

In our next experiment, depicted on the right side of fig. 6, we show the usefulness of dynamic optimization. We used the 16-dimensional index of the preceding experiments and increased the number of objects to 100,000. Hereby, the optimum page size decreased from 192 KBytes to 112 KBytes.

In our next experiment, depicted in fig. 7, we compared the DABS-tree with the X-tree and the sequential scan. As expected, the performance in low-dimensional cases is similar to the X-tree; in high-dimensional cases it is similar to the sequential scan. In any case, both approaches are clearly outperformed. In the 4-dimensional example, the DABS-tree is 43% faster than the X-tree and 157% faster than the sequential scan. In the 16-dimensional example, the DABS-tree outperforms the sequential scan by 17% and the X-tree by 462%.

In case of a moderate dimensionality, and provided that the number of points stored in the database is high, both techniques, the X-tree as well as the sequential scan, are

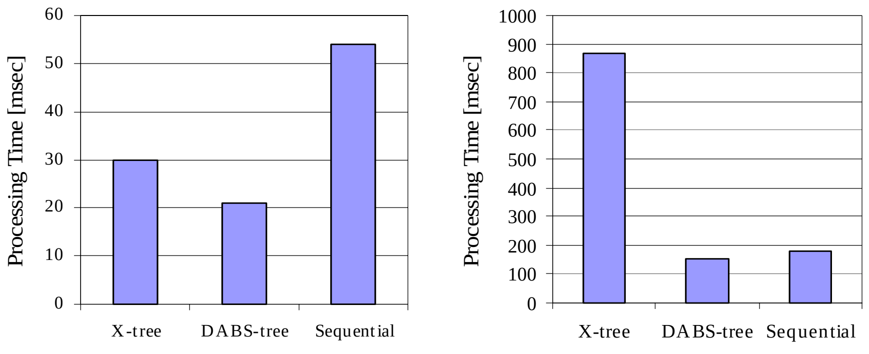

Fig. 7: Performance for 4-dimensional (left) and 16-dimensional (right) data.

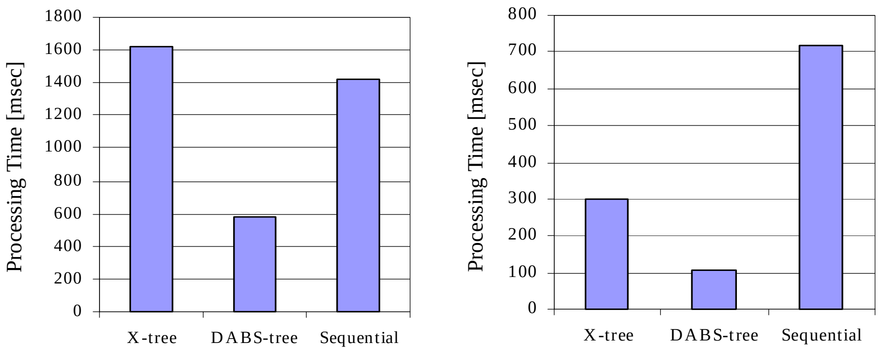

Fig. 8: Query processing on uniform (l.) and real data (r.).

clearly outperformed. This is demonstrated in the example of our 16-dimensional data-base with 100,000 points (fig. 8, left side). Here, the improvement factor over the X-tree is 2.78. The improvement over the sequential scan is with 2.44 in the same order of magnitude.

The intention of our last experiment (fig. 8, right side) is to confirm that our optimization technique is also applicable to real data and that high performance gains are reachable. For this purpose, we constructed a DABS-tree with 50,000 points from our CAD application. We measured again the performance of nearest neighbor queries. As query points, we also used points from the same application which were not stored in the database. The data space dimension was 16 in this example. We outperformed the X-tree by a factor of 2.8 and the sequential scan by 6.6.

8. Conclusion

In this paper, we have proposed a dynamic optimization technique for multidimensional index structures. In contrast to conventional page-size optimization where the administrator determines the optimal page-size parameter before installing the database, our index is automatically adapted according to the data distribution of the objects currently stored in the database. Our technique uses a flat directory whose entries consist of an MBR, a pointer to the data page and the size of the data page. Before splitting pages in insert operations, a cost model is consulted to estimate whether the split operation is beneficial. Otherwise, the split is avoided and the logical page-size is adapted instead. A similar rule applies for merging when performing delete operations. We present an algorithm for the management of data pages with varying page-sizes in an index and show that all restructuring operations are locally restricted. We show in our experimental evaluation that the DABS tree outperforms the X-tree by a factor up to 4.6 and the sequential scan by a factor up to 6.6.

References

1. Agrawal R., Faloutsos C., Swami A.: *'Efficient similarity search in sequence databases'*, Proc. 4th Int. Conf. on Foundations of Data Organization and Algorithms, 1993, LNCS 730, pp. 69-84
2. Agrawal R., Lin K., Shawney H., Shim K.: *'Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases'*, Proc. of the 21st Conf. on Very Large Databases, 1995, pp. 490-501.
3. Arya S., Mount D.M., Narayan O.: '*Accounting for Boundary Effects in Nearest Neighbor Searching*', Proc. 11th Symp. on Computational Geometry, Vancouver, Canada, pp. 336-344, 1995.
4. Aref W. G., Samet H.: *'Optimization Strategies for Spatial Query Processing'*, Proc. 17th Int. Conf. on Very Large Databases (VLDB'91), Barcelona, Catalonia, 1991, pp. 81-90.
5. Bentley J.L.: '*Multidimensional Search Trees Used for Associative Searching*', Communications of the ACM, Vol. 18, No. 9, pp. 509-517, 1975.
6. Berchtold S., Böhm C., Jagadish H. V., Kriegel H.-P., Sander J.: *'Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces'*, Proc. Int. Conf. on Data Engineering, Konstanz, Germany, 2000.
7. Berchtold S., Böhm C., Kriegel H.-P.: *'The Pyramid-Technique: Towards indexing beyond the Curse of Dimensionality'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, pp. 142-153,1998.
8. Berchtold S., Böhm C., Keim D., Kriegel H.-P., Xu X.:*'Optimal Multidimensional Query Processing Using Tree Striping'*, submitted.
9. 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, 1997, Tucson, Arizona.
10. Belussi A., Faloutsos C.: *'Estimating the Selectivity of Spatial Queries Using the `Correlation' Fractal Dimension'*. Proceedings of 21th International Conference on Very Large Data Bases, VLDB'95, Zurich, Switzerland, 1995, pp. 299-310.
11. Berchtold S., Kriegel H.-P.: '*S3: Similarity Search in CAD Database Systems*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona, pp. 564-567.
12. Berchtold S., Keim D., Kriegel H.-P.: '*The X-Tree: An Index Structure for High-Dimensional Data*', 22nd Conf. on Very Large Databases, 1996, Bombay, India, pp. 28-39.
13. Berchtold S., Keim D., Kriegel H.-P.: '*Using Extended Feature Objects for Partial Similarity Retrieval*', VLDB Journal Vol. 6, No. 4, pp. 333-348, 1997.
14. Böhm C.: *'Efficiently Indexing High-Dimensional Data Spaces'*, Ph.D. Thesis, Faculty for Mathematics and Computer Science, University of Munich, Utz-Verlag München, 1998.
15. 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, September 1977, pp. 209-226.
16. Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *'Efficient and Effective Querying by Image Content'*, Journal of Intelligent Information Systems, 1994, Vol. 3, pp. 231-262.
17. Faloutsos C., Kamel I.: '*Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension*', Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Minneapolis, Minnesota, 1994, pp. 4-13.
18. Faloutsos C., Ranganathan M., Manolopoulos Y.: *'Fast Subsequence Matching in Time-Series Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1994, pp. 419-429.

19. Faloutsos C., Sellis T., Roussopoulos N.: '*Analysis of Object-Oriented Spatial Access Methods*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1987.

20. Gaede V., Günther O.: '*Survey on Multidimensional Access Methods*', Technical Report ISS-16, Humbold-Universität Berlin, 1995.

21. Gary J. E., Mehrotra R.: '*Similar Shape Retrieval using a Structural Feature Index*', Information Systems, Vol. 18, No. 7, 1993, pp. 525-537.

22. Henrich, A.: '*The LSD^h-tree: An Access Structure for Feature Vectors*', Proc. 14th Int. Conf. on Data Engineering, Orlando, 1998.

23. C.A.R. Hoare, '*Quicksort*', Computer Journal, Vol. 5, No. 1, 1962.

24. Hjaltason G. R., Samet H.: '*Ranking in Spatial Databases*', Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 83-95.

25. Jagadish H. V.: '*A Retrieval Technique for Similar Shapes*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 208-217.

26. Kastenmüller G., Kriegel H.-P., Seidl T.: '*Similarity Search in 3D Protein Databases*', Proc. German Conference on Bioinformatics (GCB`98), Köln (Cologne), 1998.

27. Korn F., Sidiropoulos N., Faloutsos C., Siegel E., Protopapas Z.: '*Fast Nearest Neighbor Search in Medical Image Databases*', Proc. 22nd VLDB Conference, Mumbai (Bombay), India, 1996, pp. 215-226.

28. 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, pp. 369-380.

29. Kriegel H.-P., Seidl T.: '*Approximation-Based Similarity Search for 3-D Surface Segments*', GeoInformatica Journal, Kluwer Academic Publishers, 1998, to appear.

30. 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.

31. Papadopoulos A., Manolopoulos Y.: '*Performance of Nearest Neighbor Queries in R-Trees*', Proc. 6th Int. Conf. on Database Theory, Delphi, Greece, in: Lecture Notes in Computer Science, Vol. 1186, Springer, pp. 394-408, 1997.

32. Pagel B.-U., Six H.-W., Toben H., Widmayer P.: '*Towards an Analysis of Range Query Performance in Spatial Data Structures*', Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'93, Washington, D.C., 1993, pp.214-221.

33. Shawney H., Hafner J.: '*Efficient Color Histogram Indexing*', Proc. Int. Conf. on Image Processing, 1994, pp. 66-70.

34. Seidl T., Kriegel H.-P.: '*Efficient User-Adaptable Similarity Search in Large Multimedia Databases*', Proc. 23rd Int. Conf. on Very Large Databases (VLDB'97), Athens, Greece, 1997, pp. 506-515.

35. Yannis Theodoridis, Timos K. Sellis: '*A Model for the Prediction of R-tree Performance*'. Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, 1996, Montreal, Canada. ACM Press, 1996, ISBN 0-89791-781-2 pp. 161-171.

36. White D.A., Jain R.: '*Similarity indexing with the SS-tree*', Proc. 12th Int. Conf on Data Engineering, New Orleans, LA, 1996.

37. 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, New York, 1998.