Optimal Multidimensional Query Processing Using Tree Striping

Stefan Berchtold[1], Christian Böhm[2], Daniel A. Keim[3], Hans-Peter Kriegel[2] and Xiaowei Xu[4]

[1] stb gmbh, Ulrichsplatz 6, 86150 Augsburg, Germany
{Stefan.Berchtold}@stb-gmbh.de
[2] University of Munich, Oettingenstr. 67, 80538 Munich, Germany
{boehm,kriegel}@informatik.uni-muenchen.de
[3] University of Halle-Wittenberg, Kurt-Mothes Str. 1, 06099 Halle (Saale), Germany
keim@informatik.uni-halle.de
[4] Siemens AG, Information and Communications, 81730 Munich, Germany
Xiaowei.Xu@mchp.siemens.de

Abstract. In this paper, we propose a new technique for multidimensional query processing which can be widely applied in database systems. Our new technique, called tree striping, generalizes the well-known inverted lists and multidimensional indexing approaches. A theoretical analysis of our generalized technique shows that both, inverted lists and multidimensional indexing approaches, are far from being optimal. A consequence of our analysis is that the use of a set of multidimensional indexes provides considerable improvements over one *d*-dimensional index (multidimensional indexing) or *d* one-dimensional indexes (inverted lists). The basic idea of tree striping is to use the optimal number *k* of lower-dimensional indexes determined by our theoretical analysis for efficient query processing. We confirm our theoretical results by an experimental evaluation on large amounts of real and synthetic data. The results show a speed-up of up to 310% over the multidimensional indexing approach and a speed-up factor of up to 123 (12,300%) over the inverted-lists approach.

1. Introduction

The problem of retrieving all objects satisfying a query which involves multiple attributes is a standard query processing problem prevalent in any database system. The problem especially occurs in the context of feature-based retrieval in multimedia databases [3], but also in relational query processing, e.g. in a data warehouse. The most widely used method to support multi-attribute retrieval is based on indexing the data which means organizing the objects of the database into pages on secondary storage. There is a variety of index structures which have been proposed for this purpose. One of the most popular techniques in commercial databases systems is the inverted-lists approach. The basic idea of the *inverted-lists approach* is to use a one-dimensional index such as a B-tree [4] or one of its variants for each attribute. In order to answer a given range query with *s* attributes specified, it is necessary to access *s* one-dimensional indexes and to perform a costly merge of the partial results obtained from the one-dimensional indexes. For queries involving multiple attributes, however, the merging step is prohibitively expensive and is the major drawback of the inverted-lists approach. *Multidimensional index structures* have been developed as an efficient alternative approach for multidimensional query processing. The basic idea of multidimensional index structures such as space-filling curves [14, 8], grid-file based methods [13, 5], and R-tree-based methods [7, 2], is to use one multi-attribute index which provides efficient access to an arbitrary combination of attributes.

Y. Kambayashi, M. Mohania, and A M. Tjoa (Eds.): DaWaK 2000, LNCS 1874, pp. 244–257, 2000.

It is well-known that multidimensional index structures are very efficient for databases with a small number of attributes and outperform inverted lists if the query involves multiple attributes [11]. In many real-life database applications, however, we have to handle databases with a large number of attributes. For databases with a larger number of attributes, the performance of traditional multidimensional index structures rapidly deteriorates. Therefore, specific index structures for high-dimensional data have been proposed. Examples include the TV-tree [12], SS-tree [19], and X-tree [1]. For high dimensions (larger than 12), however, even the performance of specialized high-dimensional index structures decreases.

In this paper, we propose a new approach, called *tree striping*, for an efficient multiattribute retrieval. The basic idea of tree striping is to divide the data space into disjoint subspaces of lower dimensionality such that the cross-product of the subspaces is the original data space. The subspaces are organized using an arbitrary multidimensional index structure. Tree striping is a generalization of the inverted lists and multidimensional indexing approaches, which may both be seen as the extreme cases of tree striping.

The rest of this paper is organized as follows: Section 2 introduces the basic idea of tree striping including the algorithm necessary for processing queries. In Section 3, we then provide a theoretical analysis of our technique and show that optimal query processing is obtained for tree striping. We also show that optimal tree striping outperforms the traditional inverted lists and multidimensional indexing methods. In Section 4, we then discuss the more elaborate query processing algorithms which make use of the specific advantages of "striped" trees and therefore further improve the performance. Section 5 provides the details of our experimental evaluation which includes comparisons of tree striping to inverted lists and two multidimensional index structures, namely the R-tree and the X-tree. The results of our experimental analysis confirm the theoretical results and show substantial speed-ups over the multidimensional indexing and the inverted-lists approaches.

2. Tree Striping

Our new idea presented in this paper is to use the benefits of both the inverted lists and high-dimensional indexing approaches in order to achieve an optimal multidimensional query processing. Our approach, called *tree-striping*, generalizes both previous approaches.

2.1 Basic Idea

The basic idea of tree-striping is to divide the data space into disjoint subspaces of lower dimensionality such that the cross-product of the subspaces is the original data space[1]. This means that each subspace contains a number of attributes (dimensions) and each object of the database occurs in all subspaces. For example, the three-dimensional data space (customer_no, discount, turnover) may be divided into the one-dimensional subspace (customer_no) and the two-dimensional subspace (discount, turnover). Obviously, the dimensionality of the subspaces is smaller than the dimensionality of the data space, and hence, we are able to index the subspaces more efficiently using any multidimensional index structure.

To insert an object, we divide the object into subobjects according to the division of the data space. Then, we insert the subobjects in the multidimensional index structure managing the corresponding subspace. To process a query, we divide the query accord-

1. Note that a division of the data space into disjoint subspaces is different from a partitioning of the data space where the partitions have the same dimensionality as the original data space whereas subspaces have a lower dimensionality.

ing to the division of the data space and issue the subqueries to the relevant multidimensional indexes. In a second step, we merge the results which have been produced by the indexes using an external sorting algorithm such as merge sort. The general idea and query processing strategy of tree striping is presented in Figure 1.

Note that, in contrast to inverted lists, in general, the selectivity of subspace indexes is relatively high because each index manages information about more than one attribute. Therefore, the amount of partial results produced in the first step is rather small which means that the cost for the merging step are not significant. Our formal model, which will be presented in Section 3, confirms this fact.

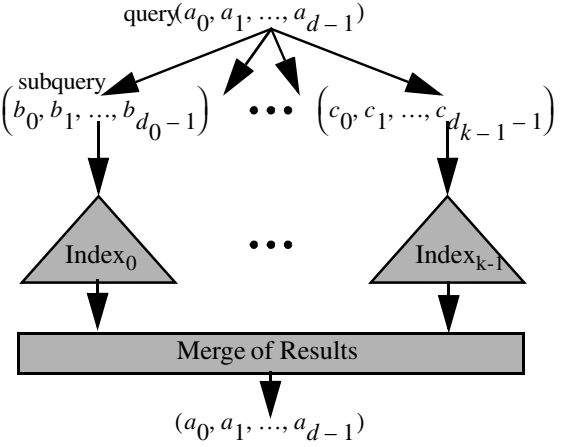


Fig. 1: Tree Striping

It is clear that the number and dimensionality of the data space divisions are important parameters for the performance of our technique. The optimal division mainly depends on the dimension, the number of data items, and the data distribution. The parameters have to be chosen adequately to achieve an optimal performance. For a uniform data distribution, the parameters for an optimal division into subspaces can be obtained easily from the theoretical analysis (cf. Section 3).

2.2 Definition of Tree Striping

In this Section, we formally define the tree striping technique. In the following, we consider objects as vectors in a vector space and attributes as components of the vectors. Given is a data space of dimension d and extension $[0..1]^d$, N vectors v having components $v_0 \ldots v_{d-1}$ and an arbitrary multidimensional index structure *MIS* supporting the relevant query types. First, we need a mapping which assigns the dimensions to the different subtrees.

Definition 1: (Dimension Assignment):

The dimension assignment DA is a mapping $R^d \rightarrow (R^{d_0}, \ldots, R^{d_{k-1}})$ of a d-dimensional vector v to a vector of k d_l-dimensional vectors w^l, $0 \le l < k$, such that the following conditions hold:

1. $\sum_{l=0}^{k-1} d_l = d$ 2. $\forall j \ 0 \le j < d, \ \exists l \ 0 \le l < k, \ \exists i \ 0 \le i < d_l: \ v_j = w_i^l$

3. $\forall l \ 0 \le l < k, \ \forall i \ 0 \le i < d_l, \ \exists j \ 0 \le j < d: \ w_i^l = v_j$

Note that w_i^l denotes the i-th component in the l-th index. To clarify the definition of dimension assignment, we provide a simple example: Given a 5-dimensional data space (d=5). We may define a dimension assignment DA_{odd_even} such that $k = 2$, $d_0 = 3$, and d_1

= 2, i.e. DA_{odd_even} divides the data space into two subspaces of dimensionality 3 and 2. Explicitly, DA_{odd_even} maps even dimensions to the first subspace and odd dimensions to the second subspace, more formally, $DA_{odd\_even}(v) = (w_0, w_1)$, $w_0 = (v_0, v_2, v_4)$, $w_1 = (v_1, v_3)$. Thus a vector v= (0, 4, 6, 5, 1) is mapped to $DA_{odd\_even}((0, 4, 6, 5, 1)) = ((0, 6, 1), (4, 5))$. Obviously, DA_{odd_even} meets the conditions specified in Definition 1 because all dimensions of the data space have been mapped to a subspace and vice versa.

Using the definition of dimension assignment, we are now able to formally define tree-striping:

Definition 2: (Tree Striping):

Given a database DB of N d-dimensional vectors and a dimension assignment DA. Then, a tree-striping TS is defined as a vector of k d_l-dimensional indexes

$$MIS^l = \{w^l\}\ ,\ 0 \le l < k\ , \qquad \text{with } w^l = DA^l(v)\ ,\ v \in DB\ .$$

Tree striping as defined in Definition 2 is a generalization of the previous approaches. For the special case of $k = d$, tree striping corresponds to inverted lists because the dimension assignment produces d one-dimensional data objects; and for the special case of $k = 1$, tree striping corresponds to the traditional multidimensional indexing approach because we have one d-dimensional index. The most important question is whether there exists a tree striping which provides better results than the extremes (the well-known inverted lists and multidimensional indexing approaches). In particular, we have to determine whether there exists a k ($1 < k < d$) such that tree striping outperforms the other approaches. In the next Section, we introduce a theoretical model showing that an optimal k exists. Our experimental analysis presented in Section 5 confirms the results of our theoretical model and shows performance improvements of up to a factor of 120 times over the inverted lists and up to 280% over the multidimensional indexing approach. A second open question is how the attributes (dimensions) are assigned to the different trees such that the performance improvement is optimal. In Section 4, we discuss the implications of different dimension assignments and also introduce optimized algorithms for query processing using striped trees.

```
SetOfObject query(TreeStrip ts, QuerySpec qs)
{
    int i;
    SetOfSubObject sst[ts.num];
    SubQuerySpec sqs[ts.num];
    SetOfObject st;

    // for all indexes
    for (i = 0; i < ts.num; i++)
    {
        // query i-th index with sub-query
        sqs[i] = ts.opt_dim_assign(i, qs);
        sst[i] = ts.index[i].query(sqs[i]);
        // sort result by primary key
        sst[i].sort();
    }
    // now merge single results
    st = merge(sst, ts.num);
    return st;
}
```

Fig. 2: A First Query Processing Algorithm

Note that tree striping as defined so far is independent of the multidimensional index structure used. Any multidimensional index structure such as the R-tree [7] and its variants (R+-tree [17], R*-tree [2], P-tree [9]), Buddy-tree [16], linear quadtrees [6], z-ordering [14] or other space-filling curves [8], and grid-file based methods [13, 5] may be used for this purpose.

Before we describe our theoretical model, we first provide a simple algorithm for processing queries using striped trees. As the single indexes do not have all information about an object, but only about some attributes of the object, in general, we have to query all indexes in order to process a query. We therefore divide the query specification *qs* into sub-

query specifications *sqs*[*l*] according to the dimension assignment. Then, we query each single index with the sub-query specification *sqs*[*l*] and record the results. In a final step, we have to merge the results by sorting the single results according to the primary key of the objects or any object identificator. Figure 2 shows a first version of a query processing algorithm. An optimized version for querying striped trees is provided in Section 4.

3. Theoretical Model

As already mentioned, most of the multidimensional indexing approaches efficiently solve the multi-attribute retrieval problem on low-dimensional data. From our experience, in real-life database projects, we have learned that even for relational database systems, handling relatively high numbers of attributes (more than 10) occurs, for which the performance of traditional index structures deteriorates. To process arbitrary queries (e.g., point, range, and partial match queries) efficiently on those databases, we have to equally index all the attributes which means that we have to deal with a high-dimensional data space.

Unfortunately, some mathematical problems arise in high-dimensional spaces which are usually summarized by the term 'curse of dimensionality.' A basic effect in high-dimensional space is the exponential growth of the volume: Let us assume a database of 1,000,000 uniformly distributed objects consisting of 20 numerical attributes in the range [0...1]. Let us further assume that we are interested in a query which provides 10 result objects located around the midpoint of the data space (0.5, 0.5, 0.5, ..., 0.5). Which range do we have to query in order to obtain 10 result objects? Obviously, we have to assure that the volume of our query range equals to $10/1{,}000{,}000 = 10^{-5}$, as the volume of the data space equals to 1. This leads to a query range in each attribute of $\sqrt[20]{10^{-5}} \approx 0.56$. So we have to query the range (0.22-0.78, 0.22-0.78, ..., 0.22-0.78). That means a query with a selectivity of 10^{-5} leads to a query range of 0.56 in each attribute in a 20-dimensional data space.

Considering these effects, we are able to provide a concise cost model of processing range queries in a high-dimensional data space using the tree striping technique. For the following, we assume a uniformly distributed set of N vectors in a d-dimensional space of extension $[0..1]^d$. Note that, although we assume a uniform distribution of the data, our model can be applied to real data as well (cf. Section 5). We will use the cost model to determine the optimal number of trees and accordingly the dimensions of the trees for a given data set, i.e. the optimal dimension assignment.

Our cost model is divided into two parts: First, the cost arising from querying the striped trees, and second, the cost for merging the results of the striped trees into one final result. Both cost functions are highly influenced by the dimensions of the striped trees. The index lookup cost is growing super-linearly with growing tree dimension. However, the merging cost is growing super-linearly with the size of the result, which is, in turn, falling with dimension of the trees. This fact implies the assumption that the total cost could form a minimum where both costs are moderate. This minimum should be located anywhere between the d-dimensional index and the inverted-lists approaches.

Several cost models for queries on multidimensional index structures have been published. The most suitable ones for our purposes are the model of Kamel and Faloutsos [10] and the similar model of Pagel, et.al. [15]. We decided to use the model of Kamel and Faloutsos as a basis for our considerations. We therefore assume that the multidimensional index structure aggregates a fixed number of C_{eff} vectors into a data page

such that the bounding box containing the vectors forms a square-shaped hyper-rectangle with the (hyper-) volume

$$V_{BB} = \frac{C_{eff}(d)}{N} .$$

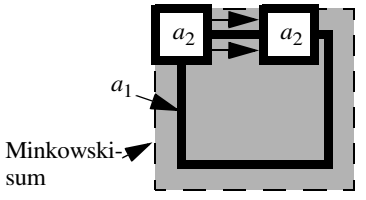


Fig. 3: The Minkowski Sum

Thus, C_{eff} denotes the actual fan-out of the index. From that, the edge length σ of a typical bounding box is

$$\sigma = \sqrt[d]{V_{BB}} = \sqrt[d]{\frac{C_{eff}(d)}{N}} .$$

Next, we determine the probability that such a data page is loaded when processing a square range query with volume V_q. Analogously, we compute the edge length q of V_q as $q = \sqrt[d]{V_q}$. The probability can be determined using the so-called Minkowski sum. Intuitively, the Minkowski sum of two areas a_1 and a_2 can be constructed by painting a_1 using a_2 as a brush (cf. Figure 3). If both areas are multidimensional rectangles, we have to add the side-lengths of a_1 and a_2 accordingly. Thus, the Minkowski sum of query volume and the bounding box is

$$Mink(V_{BB}, V_q) = (\sigma + q)^d = \left(\sqrt[d]{\frac{C_{eff}(d)}{N}} + q\right)^d .$$

The Minkowski sum $Mink(V_{BB}, V_q)$ equals to the probability that a randomly located bounding box and a randomly located query intersect. Thus, the expected number of data pages intersecting V_q is $Mink(V_{BB}, V_q)$ multiplied by the number of pages:

$$P_{\text{index}}(d, N, q) = \frac{C_{eff}(d)}{N} \cdot \left(\sqrt[d]{\frac{C_{eff}(d)}{N}} + q\right)^d = \left(1 + q \cdot \sqrt[d]{\frac{N}{C_{eff}(d)}}\right)^d$$

The number C_{eff} of data vectors in a data page also depends on the dimension d of the vectors. Assuming that each coordinate value is stored as a 32-bit floating point value and that there is an additional unique object identifier which also requires 32 bit, we determine C_{eff} as:

$$C_{eff} = \frac{\text{page-size} \cdot \text{storage-utilization}}{4 \cdot (d + 1)}$$

The cost for combining the results of the multidimensional index accesses mostly depend on the selectivities of the indexes. If |FRS| is the size of the final result set of query Q then $|IRS_i|$ is the intermediate result set produced by the i-th index having dimension d_i. Thus:

$$\frac{|IRS_i|}{N} = \left(\frac{|FRS|}{N}\right)^{\frac{d_i}{d}} = q^{d_i}$$

Note that we have to sort each intermediate result set according to the object identifiers in order to be able to merge them into the final result set. We have to apply an external sorting algorithm since, for larger q or minor d_i, the result set will exceed the available main memory. According to Ullman [18], the cost for performing multiway merge-sort on a relation of B blocks is $2B \cdot \log_M(B)$ where M is the number of cache pages available to the sorting process. We can store the object identifiers in a densely packed fashion such that $|IRS_i|$ object identifiers require $4 \cdot |IRS_i| /$ page-size pages. From that, the cost for sorting the result set of a single index are:

$$P_{\text{sort}}(d_i, N, q) = \frac{8 \cdot N \cdot q^{d_i}}{\text{page-size}} \cdot \log_M\left(\frac{(4 \cdot N \cdot q^{d_i})}{\text{page-size}}\right)$$

To determine the total cost, P_{sort} and P_{index} have to be summed up for all striped trees. For merging the result sets, each of them has to be scanned once more. Total cost is:

$$P(d_i, N, q) = \sum_{i=1}^{k} \left[\left(1 + q \cdot \sqrt[d_i]{\frac{N}{C_{eff}(d)}} \right)^{d_i} + \frac{4 \cdot N \cdot q^{d_i}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M \left(\frac{(4 \cdot N \cdot q^{d_i})}{\text{page-size}} \right) \right) \right]$$

In the following, we assume, that the d dimensions of our data space are striped into k divisions[1]

$$d_0 = d_1 = \ldots = d_{k-1} = \frac{d}{k}$$

In this case, our cost function can be simplified to:

$$P(d, k, N, q) = k \cdot \left[\left(1 + q \cdot \left(\frac{N}{C_{eff}(d/k)} \right)^{k/d} \right)^{d/k} + \right.$$

$$\left. + \frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M \left(\frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}} \right) \right) \right]$$

Figure 4 shows the total cost over k in a typical setting with a database of 1,000,000 uniformly distributed objects in a 15-dimensional data space. The selectivity of the query is 0.01%. There is a clear minimum between k=2 and k=3.

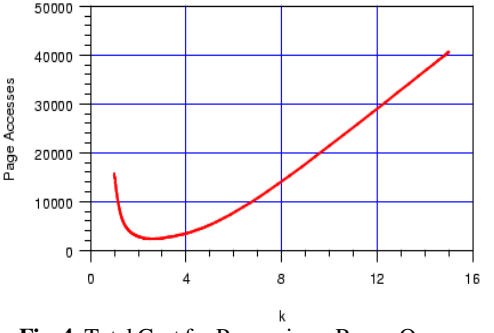


Fig. 4: Total Cost for Processing a Range Query

Thus, we are able to determine an optimal k by solving the following equation:

$$\frac{\partial}{\partial k} P(d, k, N, q) = 0 \quad \text{(eq. 1)}.$$

The analytic evaluation of this equation yields a rather large formula which is omitted due to space limitations. A MAPLE-generated C function determining the derivative can be used to calculate the optimum.

Unfortunately, the cost model presented so far is accurate only in the low-dimensional case. This is caused by the fact that in high-dimensional data spaces the data pages cannot be split in each dimension. If we split a 20-dimensional data space once per dimension, we obtain 2^{20}=1,000,000 data pages. Obviously, the number of data objects would have to grow exponentially with the dimension in order to allow one split per dimension. Therefore, we provide a special high-dimensional adaptation of our cost model. Our extension assumes that data pages are split only in the first d' dimensions where d' is the logarithm of the number of data pages to the basis of two:

$$d' = \log_2 \left(\frac{N}{C_{eff}(d_i)} \right) .$$

The data pages have the average extension 1/2 in d' dimensions and extension 1 in all remaining dimensions (d-d'). When determining the Minkowski sum, we additionally

1. For d_0, ..., d_{k-1}, only whole numbers are meaningful. This effect is handled later, but is of minor importance for our cost model.

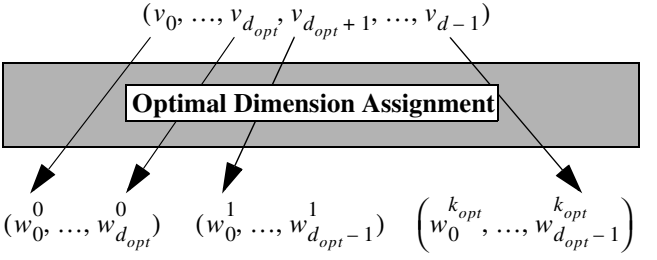


Fig. 5: Optimal Dimension Assignment

have to consider that only a part of the volume is located inside the data space because in the dimensions which have not been split, the extension of the Minkowski-sum is still 1, rather than (1+q):

$$HiDiMink(V_{BB}, V_q) = \left(\frac{1}{2} + \frac{q}{2}\right)^{d'} \cdot 1^{d-d'} \quad .$$

Thus, the expected number of data pages accessed in the high-dimensional case is:

$$P_{\text{index, HiDi}}(d, N, q) = \frac{N}{C_{eff}(d_i)} \cdot \left(\frac{1}{2} + \frac{q}{2}\right)^{\log_2(\frac{N}{C_{eff}(d)})}$$

Adding the sort cost we obtain the following total cost for high-dimensional data spaces:

$$P_{\text{HiDi}}(d, k, N, q) = k \cdot \left[\frac{N}{C_{eff}(d_i)} \cdot \left(\frac{1}{2} + \frac{q}{2}\right)^{\log_2(\frac{N}{C_{eff}(d_i)})} + \right.$$

$$\left. + \frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}} \cdot \left(1 + 2 \cdot \log_M(\frac{4 \cdot N \cdot q^{d/k}}{\text{page-size}})\right) \right]$$

4. Query processing

For optimal response times, we have to make two decision: We first have to choose an adequate dimension assignment and second, we have to choose the right strategy for processing queries.

As a result of the theoretical analysis presented in Section 3 there exists an optimal number k of striped trees, which can be determined according to our cost model (cf. equation 1). Since k is a real number, however, we cannot directly use k as a parameter for our query processor. Instead, we use the floor of k

$$k_{opt} = \lfloor k \rfloor$$

and then determine the optimal dimensionality of our trees given by

$$d_{opt} = \lfloor d/k \rfloor \, .$$

Since in general, $(k_{opt} \cdot d_{opt})$ is smaller than d we have to distribute the remaining

$$d_{rem} = d - (k_{opt} \cdot d_{opt})$$

```
void insert(TreeStrip ts, object t)
{   int l;
    SubObject st[ts.num];
    // for all indexes
    for (l = 0; l < ts.num; l++)
    {   // determine sub-objects
        st[l] = ts.opt_dim_assign(l, t);
        // insert sub-objects into l-th index
        ts.index[l].insert(st[l]);
    }
}
```

Fig. 6: Insertion Algorithm

attributes to our trees. Thus, we obtain d_{rem} trees with dimensionality $(d_{opt}+1)$ and $(k_{opt}-d_{rem})$ trees with dimensionality d_{opt}. In the following, we have to distinguish between two cases: The first case is that we have additional information about the selectivity of the attributes, which usually occurs for relational databases. The second case is that we have no additional information which usually occurs in indexing multimedia data using feature vectors. Let us first consider the more general case that we do not have any additional information and therefore assume that all attributes have the same selectivity. In this case, the optimal dimensionality d_{opt} of our trees may be used to define the following Optimal Dimension Assignment.

Definition 3: (Optimal Dimension Assignment):
The dimension assignment DA_{opt} is a dimension assignment according to Definition 1 such that:

$$DA^l(v)_i = w_i^l = \begin{cases} v_{l(d_{opt}+1)+i} & \text{if } l < d_{rem} \\ v_{d_{rem}(d_{opt}+1)+(l-d_{rem})d_{opt}+i} & \text{otherwise} \end{cases},$$

where $0 \le l < k_{opt}$, $d_l = \begin{cases} d_{opt}+1 & \text{if } l < d_{rem} \\ d_{opt} & \text{otherwise} \end{cases}$, and $0 \le i < d_l$.

Intuitively, the optimal dimension assignment assigns the i-th component of the original vector v to a component of one of the vectors w^l such that the first vector w^0 receives the first d_0 components $(v_0 \ldots v_{d_0-1})$, the second vector w^1 accommodates the components $(v_{d_0} \ldots v_{d_0+d_1-1})$ and so on.

Using the optimal dimension assignment according to Definition 3, we now are able to present the insert algorithm of our tree striping technique, as depicted in Figure 6. In order to insert an object t, we simply divide t into a set of k_{opt} sub-objects $st[l]$ (using the optimal dimension assignment) and insert them into the according striped tree $ts.index[l]$ $(0 \le l < k_{opt})$.

A more complex algorithm is required for processing queries on striped trees. A rather simple query processing algorithm has already been presented in Section 2. The algorithm depicted in Figure 2, however, has a major drawback: Let us assume that we have to process a partial range query *PRQ* which specifies attributes a, b and c:

$$PRQ = \{*,*,[a_l, a_u], *,*,[b_l, b_u], *,*,[c_l, c_u], *,*\} .$$

Let us further assume that all these three attributes are located in the first of the striped trees. Obviously, it does not make sense to query any tree other than the first tree because all other trees do not have any selectivity. The algorithm presented in Figure 2, however, executes queries on all trees ignoring the expected selectivity of the trees. In order to process queries efficiently we have to take the selectivity of a tree into account and query a tree only if the expected gain in selectivity is worth the cost of querying the tree.

Another potential improvement of the query processing algorithm can be exemplified by the following situation: Assume that the three specified attributes a, b and c in the

above example are spread over two striped trees T_0 managing attributes a and b, and T_1 managing attribute c. After querying tree T_0 we will typically receive a set of answers (candidates) which may contain some false hits. This assumption holds because the selectivity of T_0 is much higher than the selectivity of T_1. If we furthermore assume to have meaningful queries, i.e. queries having a good selectivity on all attributes, in general the set of candidates will be small. In this case, the cost for loading the candidate objects from secondary storage and checking if the objects fulfill the query specification may be lower than the cost of querying additional trees.

```
SetOfObject query(TreeStrip ts, QuerySpec qs)
{
    int i, cost_index, cost_linear;
    SetOfSubObject sst[ts.num];
    SubQuerySpec sqs[ts.num];
    SetOfObject st; // set of candidates
    // sort indexes according to selectivity
    ts.sort_index(qs);
    // determine sub-queries
    for (i = 0; i < ts.num; i++)
        sqs[i] = ts.opt_dim_assign(i, qs);
    i = 0;
    // estimate cost
    cost_index = cost_modell(sqs[0]);
    cost_linear = cost_linear_scan(sqs[0]);
    while (i < ts.num &&
            cost_index < cost_linear)
    {   // query index
        sst[i] = ts.index[i].query(sqs[i]);
        // sorted merge of result
        sst[i].sort();
        merge(st, sst, ts.num);
        // estimate cost
        if (i < ts.num)
        {   cost_index = cost_modell(st, sqs[i+1]);
            cost_linear = cost_linear(st);
        }
    }
    if (i < ts.num)
    {   // load attributes
        database.load(st);
        remove_false_hits(st, qs);
    }
    return st;
}
```

Fig. 7: Query Processing using Tree Striping

Let us now consider the second case where we do have some additional information about the selectivity of the attributes. A different selectivity of the attributes may be induced by the attributes of different data types (e.g., a boolean attribute usually has a selectivity of 50%) and by different data distributions. We can use this information to adapt the optimal dimension assignment. If we are able to query the tree containing the attributes with the highest selectivity first, the resulting set of candidates will be rather small and will contain only a few false hits. Therefore, query processing can be finished without querying the other trees. This means that, if we have information about the selectivity of attributes, we should sort the attributes according to their selectivity before applying the dimension assignment[1]. Note that in some cases, a non-uniform division may lead to better results. For example, let us assume that we have objects with 9 attributes (a, b, ... i), that k_{opt} equals to 3, and that the attributes a to d have a high selectivity whereas the selectivity of attributes e to i is rather low. Then, it is beneficial to divide the objects into sub-objects (a, b, c, d), (e, f), and (g, h, i) which would be a sub-optimal division assuming no a-priori knowledge about the selectivity of attributes.

Considering all these effects, we are now able to provide a more sophisticated algorithm for the processing of queries on striped trees. The algorithm (cf. Figure 7) first determines whether a linear search of the database is expected to be cheaper than a search using trees which may be the case for very large queries. The algorithm then sorts the striped trees according to their selectivity, i.e. the tree which probably provides the

1. Note that this operation involves not only the query-processing but also the dimension assignment, since we have to ensure that the attributes with the best selectivity are assigned to the first trees.

smallest set of candidates is queried first. If the querying of the first tree leads to a small set of candidates, we determine whether loading these candidates from secondary storage is cheaper than querying the second tree. If this is the case, we load the attributes and output all candidates fulfilling the query specification. Otherwise, we query the second tree. This process iterates until all trees have been queried or the candidates are loaded and processed.

As the implementation of multidimensional index structures is complex, the assignment of different data types such as strings and floating numbers into one tree is not practicable. The division of a object may therefore be induced not only by the expected performance improvement but also by other considerations. Obviously, this can lead to sub-optimal dimension assignments. Our practical experience, however, shows that a slightly sub-optimal dimension assignment performs nearly as well as the optimal dimension assignment.

5. Experimental Analysis

To show the practical relevance of our method, we performed an extensive experimental evaluation of tree striping and compared it to the inverted lists and the multidimensional indexing approach. All experimental results have been computed on an HP9000/780 workstation with several GBytes of secondary storage. For the experiments, we used an object-oriented implementation (C++) of the R*-tree [2] and the X-tree [1].

The test data used for the experiments are real data consisting of text data describing substrings of a large database of texts, and synthetic data consisting of uniformly distributed points in high-dimensional space. The block size used for our experiments is 4 KByte, and all query processing techniques were allowed to use the same amount of cache. For a realistic evaluation, we used very large amounts of data (up to 80 MBytes) in our experiments. The total amount of disk space occupied by the created indexes (inverted lists, multidimensional indexes and tree-striped indexes) is about 2 GBytes and the CPU time for inserting the data adds up to about one week.

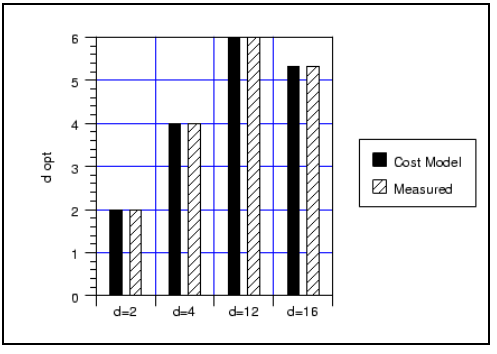
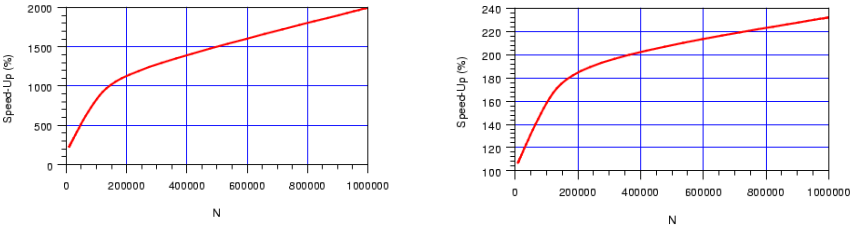


Fig. 8: Comparison of Measured Optimal Dimension Assignment and Model Estimation

In a first experiment, we confirmed our theoretical result (cf. Section 3) that the tree striping technique as a generalization of the lists and multidimensional indexing approaches outperforms both other techniques. For the experiment, we used 1,000,000 uniformly distributed data objects of varying dimensionality ($d = 2..16$). We built the according indexes (R*-tree) and queried the indexes with a selectivity of 10^{-5} which corresponds to an expected result of about 10 hits. In order to avoid statistical effects, we used the average cost of 100 uniformly distributed query windows. The observed variance was rather small. We compared different tree stripings (varying the value of k) and determined the optimal dimension assignment (optimal value of k). The tested dimension assignments for the 16-dimensional data set are (16), (8, 8), (6, 5, 5), (4, 4, 4, 4), (2, 2, 2, 2, 2, 2, 2, 2), and (1, 1, ..., 1, 1). The data sets of other dimensionality have been tested analogously. In



a. Speed-Up over Inverted Lists **b. Speed-Up over Multidimensional Indexing**

Fig. 9: Speed-Up of Tree Striping for an Increasing Number of Data Items

Figure 8, we show the optimal dimensionality (d_{opt}) of striped trees depending on dimensionality of the data. For d=2 and d=4, the optimal dimension assignment of tree striping provides one d-dimensional index, i.e. it is identical to multidimensional indexing. As expected according to our theoretical analysis, for higher dimensions the optimal dimension assignment of tree striping is between the extreme cases: For d=12, we obtain two 6-dimensional indexes and for d=16, we obtain a division into 3 indexes with dimensionality (6, 5, 5). Note that in all experiments, the optimal dimension assignment estimated by our cost model exactly matches the measured optimum. For our experiments, we therefore use the optimal dimension assignment as determined by our cost model.

Another important criterion for the evaluation of indexing techniques is their scalability, that is the behavior of the technique for an increasing size of the database. We therefore performed an experiment using a fixed dimensionality (d=16) and a fixed query selectivity of 10^{-5} and varied the number of data items from 10,000 to 1,000,000. Again, we used our cost model to determine the optimal dimension assignment. The speed-up over multidimensional indexing starts with a moderate value of 107% for a small database but, as the size of the database increases, the speed-up also increases to up-to 230% over multidimensional indexing for the largest database of 1,000,000 objects (cf. Figure 9). The speed-up over the inverted-lists approach starts with 228% and reaches its maximum of 2,000% (20 times faster) for the largest database of 1,000,000 objects (cf. Figure 9).

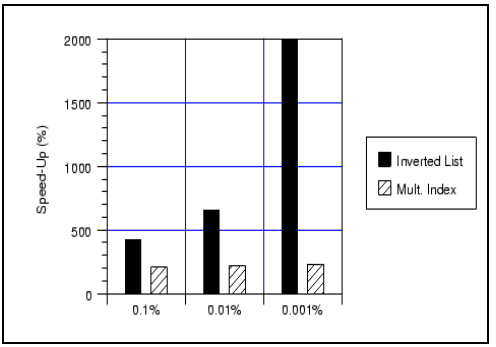


Fig. 10: Performance for Varying Selectivities

The intention of the experiment depicted in Figure 10 is to show that the high speed-ups are independent from the selectivity of the queries. We repeated the previous experiments for different dimensionality (shown are the experiment for d=12 and d=16) using selectivities between 10^{-3} and 10^{-5}. Again, we obtained a speed-up of 210% to 220% over the multidimensional index and a speed-up factor of 4 to 20 over the inverted lists.

To show the practical relevance of our technique, in a last series of experiments, we evaluated the tree striping technique using real data which consists of text data describing substrings of a large database of texts.

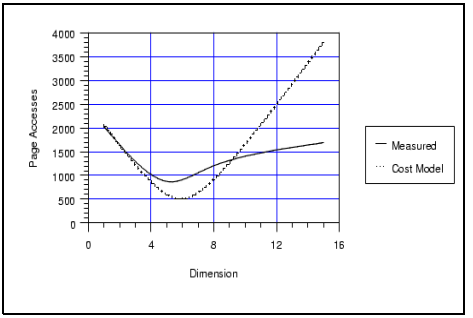


Fig. 11: Optimal Dimension Assignment for Real Data (Text Data)

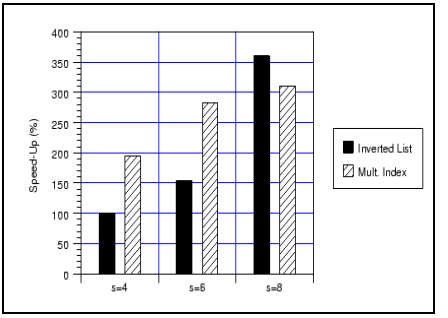


Fig. 12: Performance of Partial Range Queries (Text Data)

In Figure 11, we compare the measured performance for range queries with a selectivity of 0.2% to the performance determined by our model (cf. Section 3). The minima of the two curves correspond to the optimal dimension assignment (d_{opt}). Note that the model estimates the optimal dimension assignment correctly ($d_{opt} \approx 5$) although it assumes a uniform distribution of the data. The difference between model and measurements for large dimensions (i.e. small k), however, may be explained by the non-uniform distribution of the real data. In Figure 12, we present the speed-up of tree striping over inverted lists and multidimensional indexing for partial range queries with a varying number of attributes specified (s=4..8). It is interesting that for a partial range query with 4 attributes specified, tree striping degenerates to inverted lists. If more than 4 attributes are specified, tree striping becomes better than both, inverted lists and multidimensional indexing. Note that for s=6, inverted lists are better than multidimensional indexing, whereas for s=8, multidimensional indexing is better than inverted lists.

6. Conclusions

In this paper, we propose a new technique for multidimensional query processing, called tree striping. Tree striping is a generalization of the inverted-lists technique and the multidimensional indexing approach. A theoretical analysis of our technique shows that tree striping clearly outperforms both - the inverted lists and multidimensional indexing approaches. An experimental evaluation of our technique confirms the results of our theoretical analysis, unveiling significant speed-up factors of tree striping over inverted lists and multidimensional indexing for different databases of varying size and dimensionality, as well as for different query types.

Our future work will include an application of tree striping to other multidimensional index structures not handled in this paper and we expect a substantial performance improvement over the non-striped version. We further plan to develop a parallel version of the tree striping technique. We expect a nearly linear speed-up for a parallel version since the separate indexes can be queried independently, which should provide a linear speed-up of the query processing time.

References

1. 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.
2. Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.
3. 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.
4. Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indices',* Acta informatica, Vol. 1, No. 3, 1972, pp. 173-189.
5. Freeston M.: *'The BANG file: A new kind of grid file',* Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, CA, 1987, pp. 260-269.
6. Gargantini I.: *'An Effective Way to Represent Quadtrees',* Comm. of the ACM, Vol. 25, No. 12, 1982, pp. 905-910.
7. Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 47-57.
8. Jagadish H. V.: *'Linear Clustering of Objects with Multiple Attributes',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 332-342.
9. Jagadish H. V.: *'Spatial Search with Polyhedra',* Proc. 6th Int. Conf. on Data Engineering, Los Angeles, CA, 1990, pp. 311-319.
10. Kamel I., Faloutsos C.: '*On Packing R-trees*', CIKM, 1993, pp. 490-499.
11. Kriegel H.-P.: *'Performance Comparison of Index Structures for Multi-Key Retrieval',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA, 1984, pp. 186-196.
12. Lin K., Jagadish H. V., Faloutsos C.: *'The TV-tree: An Index Structure for High-Dimensional Data',* VLDB Journal, Vol. 3, 1995, pp. 517-542.
13. Nievergelt J., Hinterberger H., Sevcik K. C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure',* ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.
14. Orenstein J., : *'A comparison of spatial query processing techniques for native and parameter spaces',* Proc. ACM SIGMOD Int. Conf. on Management of Data, 1990, pp. 326-336.
15. Pagel B.U., Six H.-W., Toben H., Widmayer P.: '*Toward an Analysis of Range Query Performance in Spatial Data Structures*', Proc. ACM PODS, Symposium on Principa1s of Database Systems, 1993, pp. 214-221.
16. Seeger B., Kriegel H.-P.: *'The Buddy Tree: An Efficient and Robust Access Method for Spatial Data Base Systems',* Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia, 1990, pp. 590-601.
17. Sellis T., Roussopoulos N., Faloutsos C.: 'The R+-Tree: A Dynamic Index for Multi-Dimensional Objects', Proc. 13th Int. Conf. on Very Large Databases, Brighton, England, 1987, pp. 507-518.
18. Ullman J.D.: *'Database and Knowledge-Base System', Vol. II*,Compute Science Press, Rockville, MD, 1989.
19. White, D., Jain R.: *'Similarity Indexing with the SS-tree',* Proc. 12th Int. Conf. on Data Engineering, New Orleans, LA, 1996, pp. 516-523.