The DC-tree: A Fully Dynamic Index Structure for Data Warehouses

Martin Ester, Jörn Kohlhammer, Hans-Peter Kriegel

Institute for Computer Science, University of Munich
Oettingenstr. 67, D-80538 Munich, Germany
{ ester | kohlhamm | kriegel }@dbs.informatik.uni-muenchen.de
phone: +49-89-2178-2195
fax: +49-89-2178-2192

Abstract:
Many companies have recognized the strategic importance of the knowledge hidden in their large databases and have built data warehouses. Typically, updates are collected and applied to the data warehouse periodically in a batch mode, e.g., over night. Then, all derived information such as index structures has to be updated as well. The standard approach of bulk incremental updates to data warehouses has some drawbacks.First, the average runtime for a single update is small but the total runtime for the whole batch of updates may become rather large. Second, the contents of the data warehouse is not always up to date. In this paper, we introduced the DC-tree, a fully dynamic index structure for data warehouses modeled as a data cube. This new index structure is designed for applications where the above drawbacks of the bulk update approach are critical. The DC-tree is a hierarchical index structure - similar to the X-tree - exploiting the concept hierarchies typically defined for the dimensions of a data cube. The DC-tree uses minimum describing sets and the partial ordering of the attribute values induced by the concept hierarchies instead of minimum bounding rectangles and an artificial total ordering. Furthermore, for each minimum describing set in the directory the values of the measure attributes are materialized. We conducted an extensive experimental performance evaluation using the TPC-D benchmark data. Our results demonstrate that the DC-tree yields a significant speed-up compared to the X-tree and the sequential search when processing general range queries on a data cube.

1. Introduction

Many companies have recognized the strategic importance of the knowledge hidden in their large databases and, therefore, have built data warehouses. A *data warehouse* (c.f. [CD 97]) is a collection of data from multiple sources, integrated into a common repository and extended by summary information (such as aggregate views) that is used primarily in organizational decision making. Often, a data cube is used to model a data warehouse and a relational database is used for its implementation. A *data cube* [GBLP 96] consists of several independent attributes, grouped into *dimensions*, and some dependent attributes which are called *measures*. A data cube can be viewed as a *d*-dimensional array whose cells contain the measures for the respective subcube.

Typical queries on a data cube involve a lot of tuples and, consequently, tend to be very expensive. Therefore, it is a common approach ([HRU 96], [Huy 97], [MQM 97]) to materialize the results of many of the relevant queries in order to speed-up query processing. This approach,

however, fails in a very dynamic environment where the queries are not known in advance and where the number of possible queries becomes very large. In such an environment general range queries should be supported. A *range query* [HAMS 97] specifies a contiguous range for each of the dimensions of the data cube and applies a given aggregation operator to the set of selected cells. Several multi-dimensional index structures for data warehouses ([HAMS 97], [RKR 97]) have been proposed which store some derived information to efficiently support general range queries.

Typically, a data warehouse is not updated immediately when insertions and deletions on the operational databases occur. Updates are collected and applied to the data warehouse periodically in a batch mode, e.g., each night [MQM 97]. Then, all derived information has to be updated as well. This update must be efficient enough to be finished when the warehouse has to be available for users again, e.g., the next morning. Due to the very large size of the databases, it is highly desirable to perform these updates incrementally [Huy 97]. The approach of bulk incremental updates, however, has two drawbacks:

(1) While the average runtime for one update is small, the total runtime for the whole batch of updates is rather large. [RKR 97], for example, states: “Updating a large cube is a big I/O problem simply because of its mere size.” Bulk incremental updates require a considerable time window where the data warehouse is not available for OLAP. Global companies with branches all over the world, however, will more and more want to have their data warehouse available 24 hours a day.

(2) The contents of the data warehouse is not always up to date. In many applications, this may not be necessary, but it may become critical in very dynamic applications such as stock markets or the WWW.

In this paper, we introduce the DC-tree, a fully dynamic index structure for data warehouses which avoids these two drawbacks. This new index structure is designed for applications where the above drawbacks of the well-known approaches are critical. The DC-tree is a hierarchical index structure - similar to the X-tree [BKK 96] - exploiting the concept hierarchies typically defined for dimensions such as part, customer or store. This paper is organized as follows. Section 2 discusses related work. Section 3 presents the concepts of the DC-tree and section 4 discusses the major algorithms for constructing and querying DC-trees. We conducted an experimental evaluation of the DC-tree which is reported in section 5. Section 6 summarizes the contributions of this paper and outlines some directions for future research.

2. Related Work

In this section, we briefly review related work from the area of data warehousing as well as from the area of spatial index structures.

The *data cube* [GBLP 96] has been introduced as a multi-dimensional model for data warehouses. It is defined by several independent attributes, the *dimensions*, and some dependent attributes which are called *measures*. Each cell of a data cube contains the measures for the respective subcube. Often, a subset of the data cube is materialized to speed-up query processing. [HRU 96] presents an efficient algorithm which selects a nearly optimal subset of all cells for materialization. The set of all relevant queries is organized in a lattice framework and it is assumed that each query will be answered by using the smallest materialized query from which the given query can be derived. It is not considered to answer a query by combining the materializations of several queries and thus the type of queries supported is somewhat restricted. Furthermore, the proposed approach is static, i.e. it is useful only for the initial load of the cube, but does not support incremental changes on dynamic updates of the data warehouse.

Several one-dimensional index structures have been proposed for efficiently processing queries in a data warehouse and, in particular, bitmap indices have become popular in this context. In a *bitmap index*, leaf pages of an index structure do not contain lists of record ids but bit vectors with one bit for each data record. The bit vector representation very efficiently supports the set operations such as union and intersection. For instance, [OQ 97] discusses several types of bitmap index structures suitable for different query types. [OG 95] introduces bitmap *join indices* which precompute binary joins in a data warehouse. Bitmap indices, however, are static because on the insertion of a data record all index entries have to be updated. Furthermore, one-dimensional index structures build secondary indices which do not impact the clustering of the database. Therefore, they show poor performance for multi-dimensional range queries of the data cube.

Recently, several multi-dimensional index structures for data cubes have been developed. [HAMS 97] introduces a generalized quad-tree where the entries of the nodes consist of the description of a subcube and the materialized maximum measure value for this subcube. This index structure efficiently supports range-max queries (which return the maximum measure contained in a given range of cells) but the branch-and-bound-optimization cannot be applied to other query types such as range-sum queries (which return the sum of the measures contained in a given range of cells). For range-sum queries in dense data cubes, multi-dimensional prefix sums of the data cube are precomputed. Then, any range-sum query can be answered by accessing a small number - which does not depend on the size of the data cube - of appropriate prefix sums. For sparse data cubes, a set of non-intersecting subcubes is found and the prefix sum is only computed for these dense regions. An R*-tree is used to manage the minimum bounding hyper-rectangles and the prefix sums of the dense subcubes. All points not contained in one of the dense regions are also stored in the R*-tree. Unfortunately, no experimental performance results are reported.

[RKR 97] introduces the *Extended Datacube Model* (EDM) which allows to represent a data cube and its underlying relational data in a uniform way. An EDM is mapped to a cubetree and realized by a collection of packed R-trees. Furthermore, an algorithm is presented to perform bulk incremental updates of the cubetree. The set of incremental updates is sorted and the R-trees corresponding to the cubetree are merge-packed. The cubetree is not really updated, but it is completely rebuilt so that the old cubetree still can be used during maintenance. An experimental evaluation demonstrates that the cubetree supports multi-dimensional range queries very efficiently. The bulk update experiments show that the required I/O time is very high due to the huge size of the data cubes. Further optimization techniques should be explored to reduce this problem.

Many applications require the management of *spatial data* such as points, lines and polygons. Note that the space of interest may either be an abstraction of a real *2D* or *3D* space such as a part of the surface of the earth or some high-dimensional space of feature vectors each describing an object of an application. In order to speed up query processing in spatial databases, many *spatial index structures* have been developed to restrict the search to the relevant part of the space (*cf* [GG 98] for a survey). The *R-tree* [Gut 84] generalizes the 1-dimensional B-tree to *d*-dimensional data spaces, i.e. an R-tree manages *d*-dimensional hyperrectangles instead of 1-dimensional numeric keys. An R-tree may organize extended objects such as polygons using *minimum bounding rectangles* (*MBR*) as approximations as well as point objects as a special case of rectangles. To answer a range query, starting from the root, the set of MBRs intersecting the query range is determined and then their referenced child nodes are searched until the data pages are reached.

Since the overlap of the MBRs in the directory nodes grows with increasing dimension *d*, the R-tree and most other spatial index structures are efficient only for moderate values of *d*. Recently, several index structures such as the X-tree have been designed which are also efficient for large values of *d*. The *X-tree* [BKK 96] includes a new algorithm to find an overlap-free split. Furthermore, the concept of a *supernode*, i.e.a large directory node of variable size (a multiple of the usual block size), is introduced. If the standard topological split (considering properties of the MBRs such as their extension and their partitioning of dead space) results in high overlap, the new split algorithm tries to find an overlap-minimal split which can be determined based on the split history. For point data, there is always an overlap-free split. If the number of MBRs in one of the resulting partitions is below a given threshold the split would be too unbalanced and, therefore, the split algorithm terminates without providing a split. In this case, the current node is extended to become a supernode of twice the standard block size. If the same case occurs for an already existing supernode, the supernode is extended by one additional block.

3. The DC-tree

In this section, we introduce the structure of the DC-tree, which is similar to that of the X-tree. It contains datanodes, normal directory nodes as well as supernodes (although the usage of supernodes in a DC-tree is different). To index a data cube, the MBRs are replaced with MDSs (minimum describing sets). Before the discussion of this topic we explain the integration of concept hierarchies in the DC-tree.

3.1 Concept Hierarchies

A *data cube* [GBLP 96] consists of several functional attributes, grouped into *dimensions*, and some dependent attributes which are called *measures*. A data cube can be viewed as a *d*-dimensional array whose cells contain the measures for the respective subcube. If more than one functional attribute per dimension exists, these multiple attributes are organized by hierarchy schemata. A concept hierachy is an instance of a hierarchy schema. Figure 1 shows an example for a dimension *Customer* and its functional attributes *Region*, *Nation* and *Customer ID*. *ALL* is the root of every concept hierarchy and denotes the union of all values in the concept hierarchy.

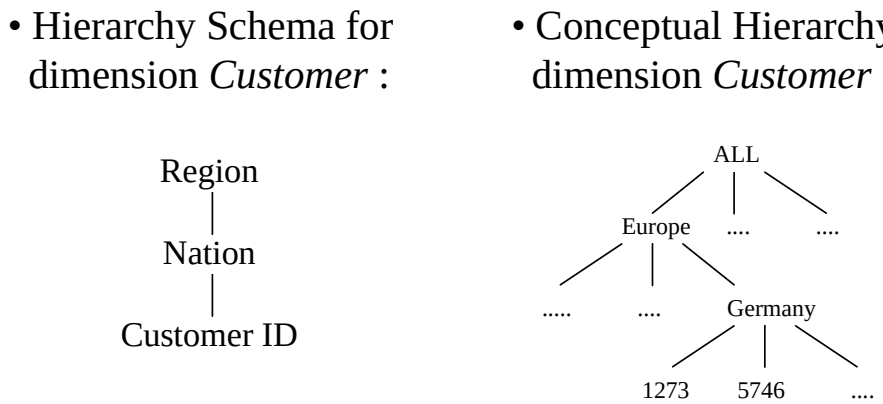


figure 1: Hierarchy Schema and Concept Hierarchy for dimension Customer

We can define a partial ordering on a dimension using its concept hierarchy. This partial ordering is important for the split algorithms of the DC-tree and the definition of MDSs. The following definitions introduce these notions and the concept of a data cube itself more formally.

Definition (Concept Hierarchy, Partial Ordering ≤)

Let D_i, $1 \leq i \leq d$, be sets of attribute values with $ALL \in D_i$. A *concept hierarchy* for D_i is a tree with the following properties:

- the nodes of the tree represent the elements of D_i
- the root represents the special value *ALL*
- the edges of the tree represent the “is-a” relationship between the two connected nodes.

Let $a, b \in D_i$, $1 \le i \le d$. The *partial ordering* $\le$ for D_i is defined as follows: $a \le b$ if and only if a is equal to b or a is a (direct or indirect) son of b in the concept hierarchy of dimension i.

Definition (Data Cube, Data Record)

We define a *datacube D* over the domains D_i ($1 \le i \le d$) with m measures as follows: .

$$D \subseteq D_1 \times \ldots \times D_d \times \Re^m$$

An element of the datacube is called a *data record*. It has the form $(a_1, \ldots, a_d, x_1, \ldots, x_m)$ with $a_i \in D_i, x_j \in \Re$.

For example, in the concept hierarchy of figure 1 *Germany* $\le$ *Europe* and $a \le$ *ALL* holds for each value a.

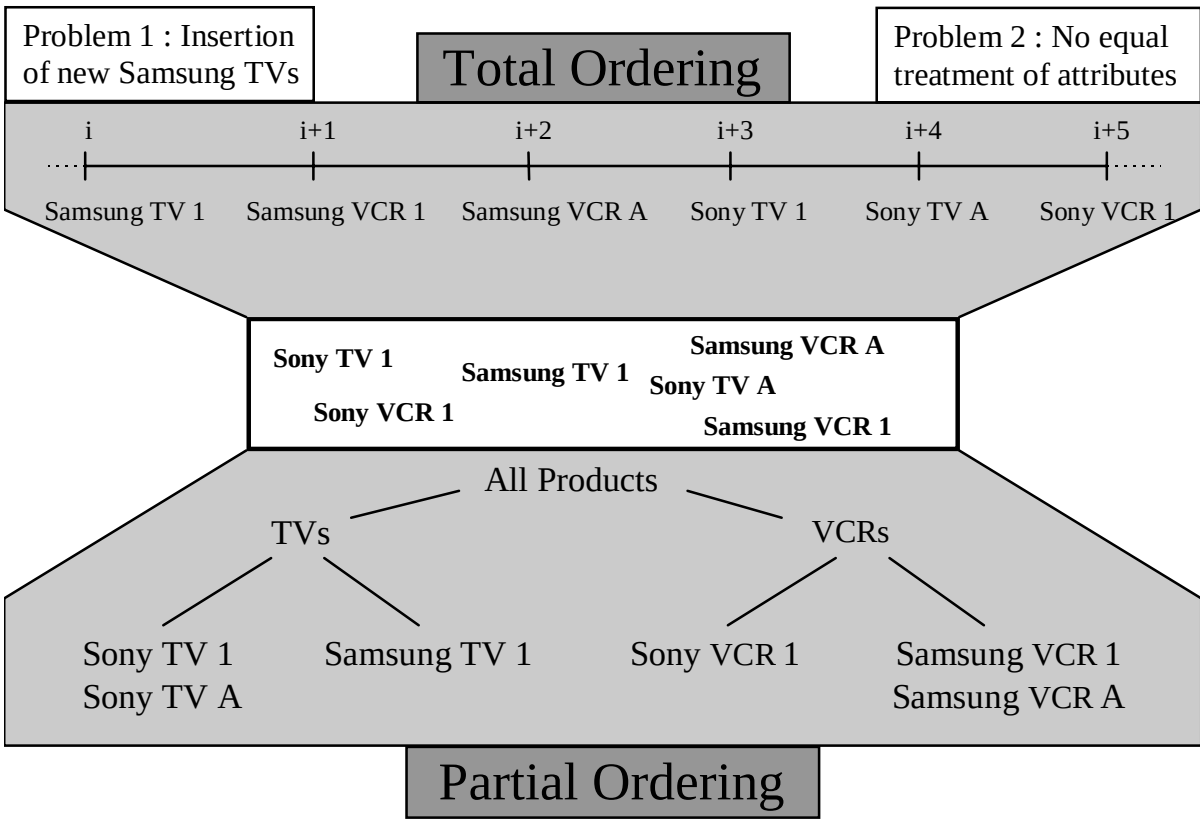


figure 2: Total Ordering and Partial Ordering for Products

In order to apply a spatial access method such as the X-tree for indexing a data cube, one could define an arbitrary total ordering for each dimension. The advantages of a partial ordering compared to a total ordering are illustrated in figure 2 for the dimension *Product*. In a total ordering, an ID has to be assigned to every single product. A problem occurs when new products have to be inserted. For example, a new Samsung TV would receive an unfavourable ID, as it would naturally fit in between i and i+1. Furthermore, the attributes are not treated equally by a

total ordering. The ordering in figure 2, for instance, would rather prefer range queries by makes than by product types.

When using a partial ordering, the insertion of new products is natural, because every leaf of the concept hierarchy is organized as a set. Although this structure prefers range queries by product types, a range query by makes can still be answered efficiently by the leaf sets of the hierarchy.

The DC-tree stores one concept hierarchy per dimension and assigns an ID to every attribute value of a data record that is inserted. This assignment is performed to avoid the storage of long strings, but not to define a total ordering. Note that the DC-tree manages its concept hierarchies dynamically. The concept hierarchies are not fixed, but are updated on every insertion depending on the IDs of the inserted data record, i.e., an empty DC-tree contains empty concept hierarchies for all dimensions.

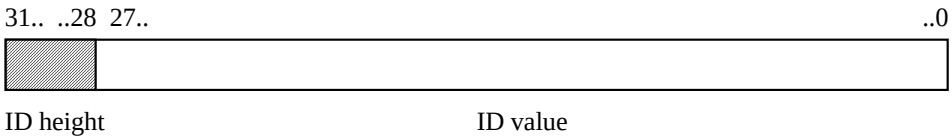


figure 3: Structure of IDs

The definition of an ID is depicted in figure 3. The highest four bits define the height of an ID in the concept hierarchy of its dimension to distinguish IDs from different levels. This representation of an ID has a fixed length and requires only 4 bytes. The DC-tree represents the concept hierarchies by means of dictionaries that store the ID of the father for each ID in one concept hierarchy. For the dimension *Customer* of figure 1, for example, every data record includes one value each for *Region, Nation* and *Customer ID*. The DC-tree assigns an ID to each value and updates its concept hierarchy for the dimension *Customer.*

3.2 Minimum Describing Sets

Minimum bounding rectangles (*MBRs*), used as the approximation method of the X-tree, are not appropriate for the DC-tree. The example in figure 4 demonstrates this fact. For two partially ordered dimensions A and B, let an arbitrary total ordering be imposed on the dimensions given by the indices of the values A_i and B_j. On the lefthand side we have a traditional MBR, which assumes totally ordered dimensions. ([A2, A8], [B2, B5]) is a sufficient definition of this MBR because [A2, A8], e.g., implictly includes each attribute value between A2 and A8. This does not hold if the dimensions are only partially ordered. Then we have to use so-called MDSs (*minimum describing sets*) instead of MBRs.

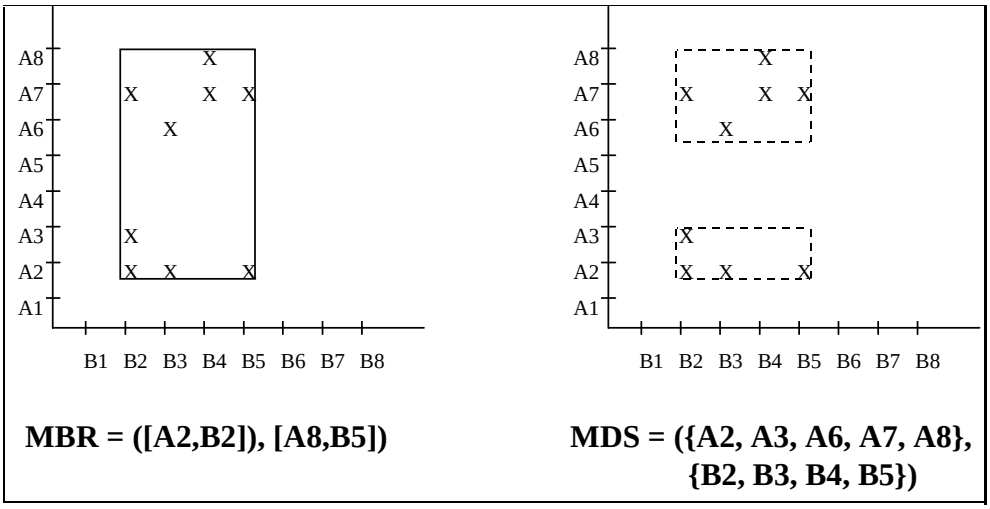


figure 4: Sample MBR and MDS

On the righthand side of figure 4 the same data records are approximated by an MDS. Only attribute values occurring for at least one data record are included in the definition of this MDS. For instance, A1 and A4 are not part of the first set of the MDS. Obviously, the MDS covers less dead space than the MBR. On the other hand, an MDS has to store more information and it has variable size. More formally, an MDS is defined as follows.

Definition (Minimum Describing Set, MDS)

Let D be a data cube with d dimensions D_i. Let $S \subseteq D$ be a subcube of D, i.e. a set of data records $(a_1, \ldots, a_d, x_1, \ldots, x_m)$ with $a_i \in D_i, x_j \in \Re$. A *minimum describing set* (*MDS*) for S is a sequence $(M_1, \ldots, M_d)$ with $M_i \subseteq D_i$ satisfying the following two properties:

1. (*coverage*) For all $(a_1, \ldots, a_d, x_1, \ldots, x_m) \in S$ and for all i, $1 \leq i \leq d$, there is some $m_i \in M_i$ with $a_i \leq m_i$.

2. (*minimality*) If $(N_1, \ldots, N_d)$, $N_i \subseteq D_i$, satisfies the above property of coverage, then for all i, $1 \leq i \leq d$, and for all $m_i \in M_i$, there is some $n_i \in N_i$ with $m_i \leq n_i$.

Each set contains IDs of a specific functional attribute of the dimension it is assigned to. As an example we examine the following data record (prior to an assignment of IDs according to chapter 3.1) for a data cube with dimensions *Customer*, *Supplier* and *Time* and one measure:

([Europe, Germany, 3405], [North America, USA, 229], [1996, Nov, 27], 310.27 $)

To distinguish data records from each other, an MDS of a single data record has to use the attribute values of the lowest level in the concept hierarchy of each dimension. For the sample data record, the MDS is ({3405},{229},{27}). An MDS that approximates a whole datanode or a directory node may use values of higher levels in the concept hierarchies, e.g., ({Europe, Asia}, {USA}, {1995, 1996}).

The last element of the above sample data record is the measure value according to the measure attribute of the data cube. The measure value is not part of the MDS, but is related to it and will be stored together with the MDS in each node of the DC-tree. The measure value for an MDS of a datanode or a directory node is the aggregation (e.g. the sum or the average) of the measure values of all data records covered by this MDS.

Finally, the following definition introduces several notions for MDSs which are essential for the algorithms of the DC-tree.

Definition (Contains, Volume, Overlap, Extension)

Let D be a data cube with d dimensions D_i. Let $M = (M_1, \ldots, M_d)$, $M_i \subseteq D_i$, and $N = (N_1, \ldots, N_d)$, $N_i \subseteq D_i$, be MDSs. Let $|S|$ denote the cardinality of a set S.

N contains M if for each dimension i, $1 \leq i \leq d$, and for all $m_i \in M_i$, there is some $n_i \in N_i$ with $m_i \leq n_i$.

The *volume* of M, denoted as *volume*(M), is defined as $volume(M) = \prod_{i=1}^{d} |M_i|$.

The *overlap* of M and N, denoted as *overlap*(M,N), is defined as

$$overlap(M, N) = \prod_{i=1}^{d} |M_i \cap N_i|.$$

The *extension* of M and N, denoted as *extension*(M,N), is defined as

$$extension(M, N) = \prod_{i=1}^{d} |M_i \cup N_i|.$$

Note that for the operations *overlap* and *extension* in each dimension i the elements of both M_i and N_i have to belong to the same level of the concept hierarchy. This is necessary because, e.g., the union of *American customers* and *North America* makes no sense.

As the MDSs have no fixed size, the storage of the entries of a DC-tree is more complicated than it was for the X-tree using MBRs. The structure of a directory node of the DC-tree is illustrated in figure 5.

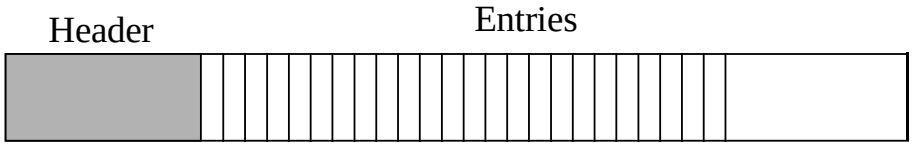


figure 5: Structure of a Directory Node

Each directory entry consists of a pointer to the corresponding son node and a pointer to its MDS. The header contains the MDS and the measure values of the node. Thus, the entries of a

directory node have fixed size, but the header of the directory node has variable size. Thus, a node could become overfilled because of an extension of its MDS without increasing the number of entries of the node. On the other hand, to assure the balance of the DC-tree, only those directory splits are allowed that are initiated by a split of a son node. Therefore, a directory node may not be split because of the extension of its MDS. Instead, we have to increase its capacity by making the directory node a supernode or by extending an already existing supernode.

4. Algorithms for the DC-tree

In this section, we present the major algorithms for constructing and querying DC-trees. While the insert algorithm is quite similar to that in the X-tree, we developed a completely new split algorithm. It uses the MDSs and the measure values to build up the DC-tree in a way that accelerates queries on the tree.

4.1 Insert

As mentioned above, the insert algorithm of the DC-tree only slighty differs from the corresponding algorithm in the X-tree. Figure 6 shows the insert procedure for directory nodes. The data record to be inserted already contains the assigned IDs for its attribute values and its measure value. After updating the measure value of the directory node, the choose-subtree algorithm selects a son *follow*, in which the data record will be further inserted. If *follow* had to be splitted as a result of this insertion, the directory node contains a new son and can now be overfilled itself. In this case, the split algorithm for directory nodes will be called. A successful split will

```
int DCTreeDirNode :: Insert (data *d)
{
        Update the measure value of the directory node;
        follow = choose_subtree (d);
        follow -> insert (d);
        If follow was splitted :
        {
                Insert new son;
                If the capacity is reached :
                {
                        Split (DCTreeDirNode *NewBrother);
                        If split was successful:
                                return SPLIT;
                        Else :
                                return SUPERNODE;
        }
        return NONE;
}
```

figure 6: Insert Algorithm for Directory Nodes

then create a new brother of this directory node. If the split was not successful, a supernode will be created or, if the directory node has already been a supernode, the supernode will be enlarged.

4.2 Split

Unlike the insert procedure the split algorithm of the DC-tree has not much in common with the corresponding algorithm of the X-tree. In particular, it exploits the partial ordering induced by the concept hierarchies.

Again, the algorithm for directory nodes will illustrate the split procedure of the DC-tree (see figure 7). The directory split runs through all dimensions, until it finds an appropriate split or until it has examined every dimension.

```
DCTreeDirNode :: Split (DCTreeDirNode *NewBrother)
{
        While found = FALSE and at least one dimension is left :
        {
                Choose dimension by level of the concept hierarchy;
                Adapt MDSs of entries to MDS of directory node;
                Hierarchy_split (Modified_MBLs[], Split_Dimension);
                In case of balanced nodes and not too much overlap :
                        found = TRUE;
        }
        If no appropriate split was found :
        {
                Create supernode;
        }
}
```

figure 7: Split Algorithm for Directory Nodes

The algorithm starts with selecting a split dimension by considering the hierarchy level of the elements of the MDS in the different dimensions. To induce a balanced structure of the DC-tree, the algorithm always selects the dimension with the highest hierarchy level of the elements of the MDS. For instance, if an MDS contains only the ALL value in one dimension and in all other dimensions attribute values from lower levels of their concept hierarchies, then the dimension with the ALL value will be selected as split dimension.

Now the MDSs of the directory entries will be adapted to the MDS of the directory node. The reason for this procedure lies in the constraint for the operations with MDSs as described in chapter 3.2. All MDSs corresponding to the entries of a node have to be comparable to each other, i.e. in every dimension they have to contain elements of the same level in the corresponding concept hierarchy. Because the MDS of a directory node itself contains all MDSs of the

entries of this directory node (see section 3.2), this directory MDS is the best choice for the adaption of the MDSs.

The modified MDSs and the chosen split dimension are the parameters of the hierarchy split, an algorithm that is described in the next section. It results in two groups of MDSs. If these two groups are too unbalanced or their overlap is too high, another dimension will be selected. If no appropriate split was found for any dimension, a supernode is created.

4.3 Hierarchy Split

The hierarchy split is based on the quadratic split of [Gut 84], that was initially developed for the R-tree and its variants. The idea of this algorithm is to split a group of MBRs into two subgroups by first choosing two MBRs, so-called seeds, that should not be contained in the same subgroup. Then each remaining MBR is inserted into one of the subgroups, according to certain criteria such as the volume of the resulting group. Because each run of the while loop considers each remaining MBR for the next insertion, the runtime of the algorithm is quadratic.

```
Hierachy_Split (MBAs[], split dimension)
{
        For each pair of MDSs compute the covering MDS;
        Choose the pair with the largest MDS as the two seeds;
        While remaining MDSs exist :
        {
                Compute the extension of the two groups in the split dimension
                for each remaining MDS;
                Choose an MDS by the largest difference between the
                extensions of the two groups in the split dimension;

                Insert this MDS into the group with the least resulting overlap of
                the groups;
                Resolve ties by choosing the least extension of the groups;
                Resolve further ties by choosing the least volume of both groups;
        }
}
```

figure 8: Hierarchy Split

The hierarchy split shown in figure 8 has the same basic structure as the original quadratic split in the R-tree. First, two seeds are being chosen from all MDSs. These two MDSs form the initial two groups of the algorithm. In every run of the while loop, the algorithm has to make two decisions: Which MDS will be inserted next and into which group will this MDS be inserted?

The first decision is made by using the split dimension. The purpose of splitting along a split dimension is to obtain two groups with disjunct attribute values in the split dimension. The

algorithm of [Gut 84] had no parameter for a split dimension and the next MBR to be inserted was chosen by the largest difference between the extensions of the two groups as a whole. This criterion is not applicable for the hierarchy split. The algorithm of figure 8 considers the extension of the groups just in the split dimension. It selects a group such that the new MDS and the MDS of the group share as many attribute values as possible in the split dimension. In the best case, the two resulting groups will contain disjoint attribute values in the split dimension.

The second decision is based on three criteria: overlap, extension and volume (in the order of their importance). Thus, the chosen MDS will be inserted into the group with the least resulting overlap of the two groups. Should the overlap be equal, the extension will be considered. If even the volume increase or the volume do not allow a decision, any of the groups will be chosen.

4.4 Range Query

To demonstrate how the DC-tree makes use of the measure values stored in the entries of the tree, we will discuss the range query algorithm for directory nodes (see figure 9). The query range for this algorithm is an MDS, the range_MDS.

```
double DCTreeDirNode :: Range_Query (range_MDS)
{
        result = 0.0;
        For each directory entry :
        {
                For each dimension :
                {
                        If the range_MDS and the MDS of the entry are not on the
                        same level in the current dimension, adapt the MDS with
                        the lower level to the one with the higher level;
                }
                If the overlap between range_MDS and the MDS of the entry is not
                empty :
                {
                        If the MDS of the entry is contained in the range_MDS :
                                Result += measure value of entry;
                        Else :
                                Result += entry -> Range_Query (range_MDS);
                }
        }
        return result;
}
```

figure 9: Range Query Algorithm

The range query algorithm runs through every entry of the directory node. The for-loop makes the two MDSs comparable to each other. It is similar to the one in the split algorithm (see figure 7) but in this case here we do not know which of the two MDSs contains the higher level attribute values.

If the overlap between the range_MDS and the MDS of the entry is empty, the entry is not relevant for the query and the result remains as it is. Otherwise, we have to further analyze the overlap. If the MDS of the entry is fully contained in the range, then the measure value stored in the son node referenced by the current directory entry, is added to the result. This is the advantage of the algorithm, because the entry and all nodes below this entry do not have to be considered and the algorithm can simply use the measure value computed during the insertions. Obviously, if the MDS of the entry and the range overlap each other, we cannot use the measure value and have to recursivly call the range query for the son node. Note, that in this case the aggregation *SUM* is being used within the algorithm. Other aggregations like *AVERAGE* or *COUNT* would have to be treated accordingly.

5. Performance Evaluation

In this section, we demonstrate the performance of the DC-tree indexing a realistic data cube. To outline the advantages of the DC-tree, we compare the DC-tree with the X-tree and the sequential search. Before analyzing the results, the test environment and the generation of the queries will be illustrated.

5.1 Test Environment

All performance tests use the database of the TPC Benchmark D [TPC 98]. The intended data cube is created by SQL select operations on the TPC-D database. The output of these operations

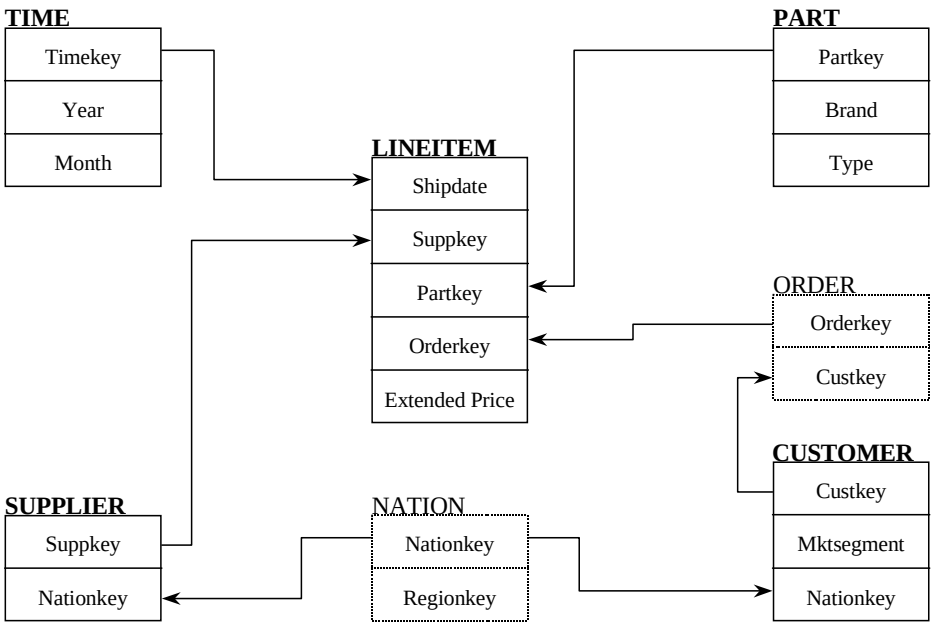


figure 10: TPC-D Database Scheme

is stored in a flatfile which functions as the insert file for the DC-tree and for the two other index structures being compared with the DC-tree.

As not all attributes in the TPC-D database were important for this performance evaluation, the corresponding database schema was simplified (see figure 10). Thus, the resulting data cube consists of four dimensions: *Supplier, Customer, Part* and *Time*. Figure 10 involves, that each data record contains 14 attributes organized in hierarchy schemata as illustrated in figure 11. The 14th attribute is the measure attribute *Extended Price*.

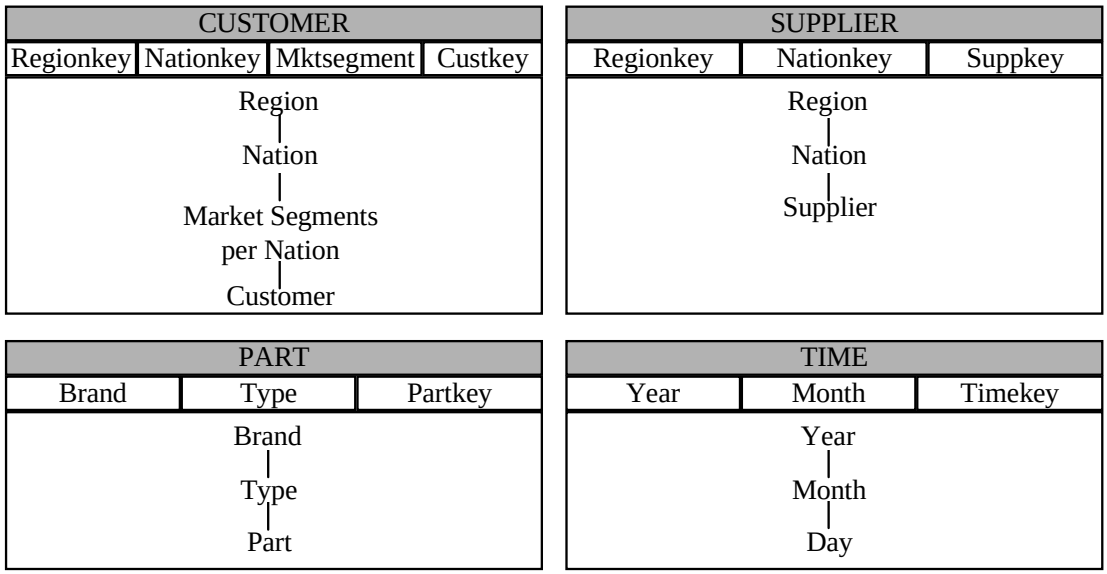


figure 11: Dimensions of the Test Datacube

5.2 Generation of the range queries

To evaluate the results, the generation of the range queries and the processing of these queries in the X-tree and in the sequential search are important. Figure 12 shows the algorithm for the generation of the range queries.

As the range is an MDS itself, the algorithm chooses a level in the concept hierarchy of each dimension that the set of the MDS in this dimension will be assigned to. The size of the range (i.e. the size of the MDS) is limited by the selectivity. For instance, a selectivity of 25 % involves a range, that contains up to 25 % of all attribute values of the chosen level in each dimension. These attributes values will be randomly chosen.

The generation of a range query for a DC-tree is different from that for an X-tree, because the X-tree was developed for using MBRs and therefore cannot use MDSs and concept hierarchies. To use the existing range query algorithms of the X-tree for our test environment, we assigned a dimension to each level of the concept hierarchies. Figure 13 shows the fourteen dimensions of the X-tree and the corresponding hierarchy levels of the DC-tree.

```
Gen_range (number, sel)
{
        While number queries have not yet been executed :
        {
                Randomly choose a level in the concept hierarchy of each dimension;

                According to the selectivity sel, compute the percentage of values on
                the chosen level, that will be contained in the range;

                Randomly select the chosen number of hierarchy values in each
                dimension on the chosen level and store them in range_MDS;

                result = dwt -> range_query (range_MDS);
        }
}
```

figure 12: Generation of Range Queries

By using the total ordering of the IDs assigned to the attribute values by the insert procedure (see section 3.1), the range_MDS can be converted to a range_MBR for the X-tree. Thus, we can compare an index structure using totally ordered dimensions (the X-tree) with an index structure using partially ordered dimensions (the DC-tree).

For the sequential search we can use the range_MDS generated by the algorithm in figure 12. The range query algorithm simply runs through every existing data record and determines whether this data record is contained in the range_MDS or not. In the positive case, the measure value of the data record is added to the result.

5.3 Results

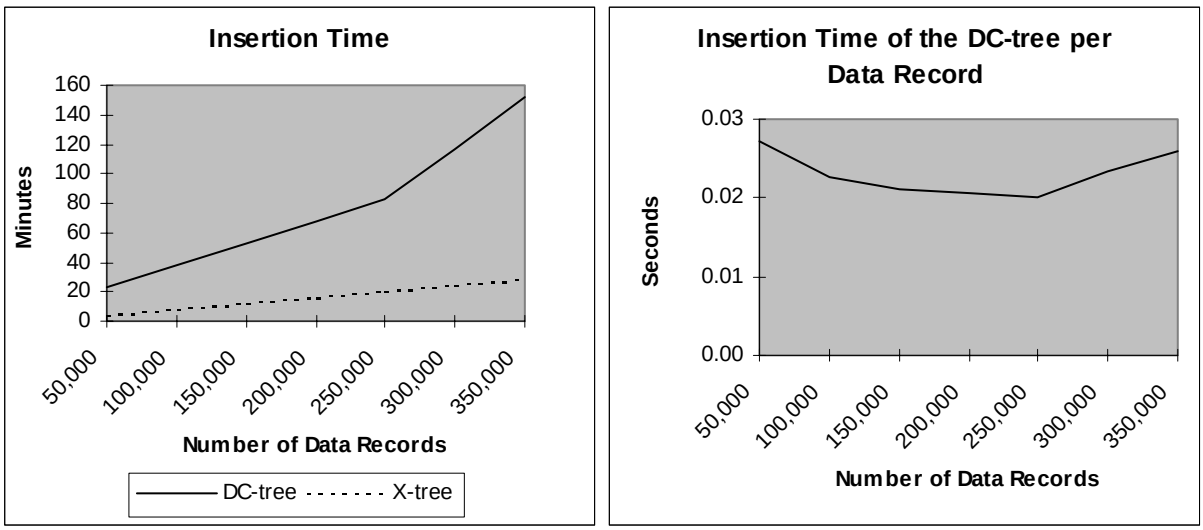
In this section, we present the results of the comparison between the DC-tree, the X-tree and the sequential search. The size of the underlying test data cubes ranges from 50’000 data records to 350’000 data records. The insertion time and the time per range query will be analyzed for different selectivities.

	CUSTOMER				SUPPLIER		
	Regionkey	Nationkey	Mktsegment	Custkey	Regionkey	Nationkey	Suppkey
X-Tree	0	1	2	3	4	5	6

	PART			TIME			
	Brand	Type	Partkey	Year	Month	Timekey	Measure
X-Tree	7	8	9	10	11	12	13

figure 13: Dimensions of the X-tree for the Test Datacube

Figure 14 (a) shows the insertion time for the DC-tree and the X-tree for up to 350'000 data records. As the X-tree stores no concept hierarchies and, therefore, many computations the DC-tree has to do are obsolete, the insertion time is significantly lower for the X-tree. However, figure 14 (b) depicts that the insertion of one single data record into the DC-tree takes only about 0.025 seconds on a HP C160 workstation (64 PA-RISC) with 768 Megabyte RAM and HP UX 10.20. Thus, the dynamic insertion of data records has no significant impact on the runtime of a data warehouse and it is reasonable to keep the DC-tree up-to-date at all times.



(a) **(b)**

figure 14: Insertion Time

Several tests with range queries of different selectivities were performed to compare the efficiency of the DC-tree to the X-tree and the sequential search. Figure 15 shows three comparative tests between the DC-tree and the X-tree for the selectivities 1 %, 5 % and 25 % as well as a comparison between the DC-tree and the sequential search. For all selectivities the time per query is determined as the average of 100 random queries.

To assure a fair comparison, the main memory available for the X-tree was restricted to the memory size that the DC-tree uses. In all performed tests the range queries are executed much faster on the DC-tree than on the comparative index structure. In fact, we obtain a speed-up of about 4.5 for range queries on the DC-tree compared with those on the X-tree.

When looking at the absolute numbers in figure 15 (a) - (c), range queries with selectivity 5 % are answered faster than the others. The reason for this fact lies in a trade-off between the level on which the DC-tree can completely answer a range query and the performance costs when executing the range queries with larger range-MDSs. The larger the query MDS, the higher is the probability that the MDS of an entry is fully contained in the query MDS. Thus, the larger the

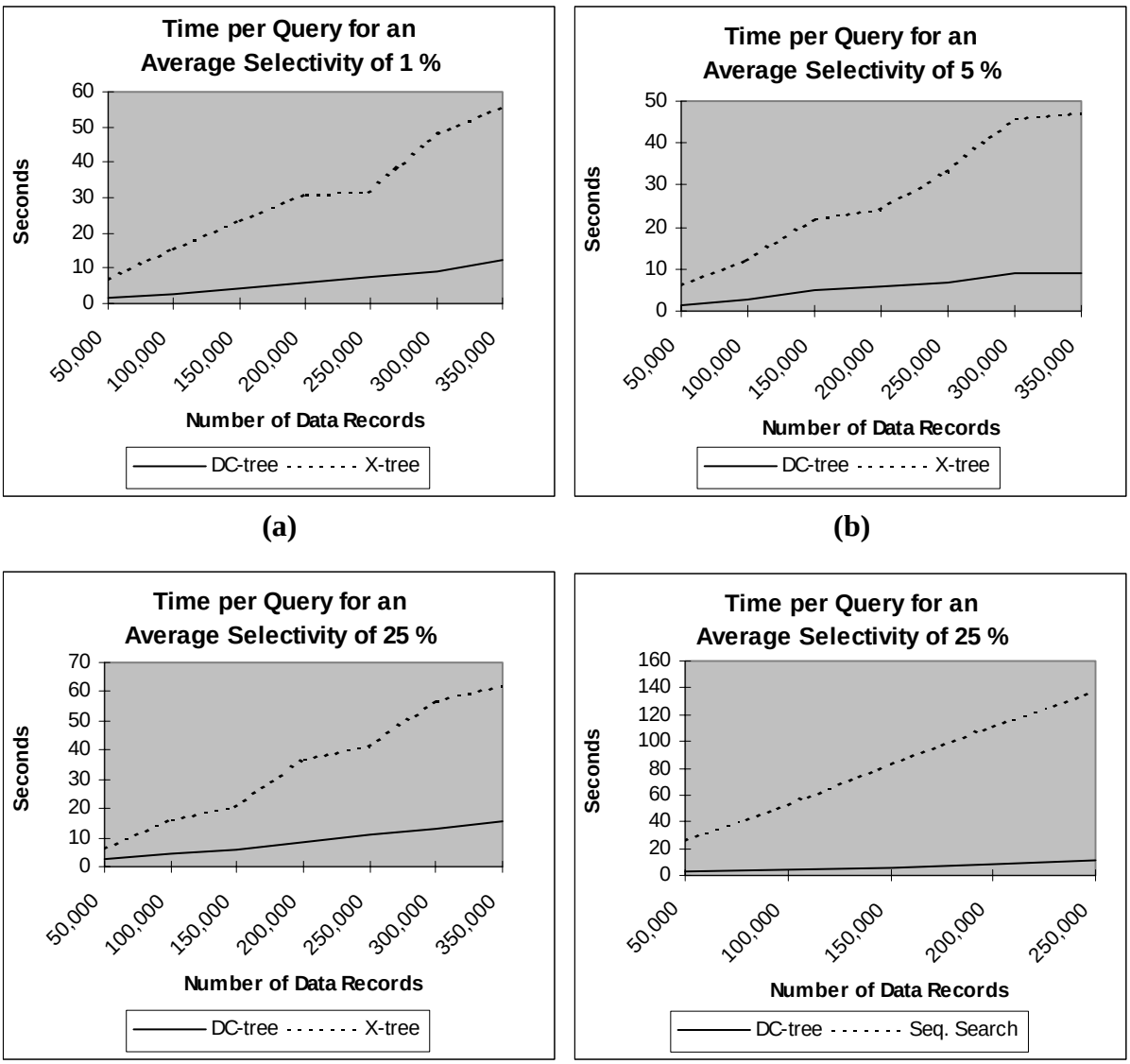


figure 15: Query Time

query MDS, the higher is the level of the DC-tree on which the range query can be answered by using a materialized measure value. Thus, a larger query MDS yields a better performance. On the other hand, a larger query MDS involves more expensive computations of the overlap, because a large MDS consists of large sets for the single dimensions. While the query MDSs of range queries with selectivity 1 % are in general too small to answer the query on a high level of the DC-tree and the query MDSs of range queries with a selectivity of 25 % involve very expensive computations, the range queries with selectivity 5 % seem to be the best compromise among the performed tests.

The comparison between the DC-tree and the sequential search, depicted in figure 15 (d), shows that the sequential search is no reasonable alternative. Even when using a selectivity of 25 % (the worst case for the DC-tree), we obtain a speed-up of 12.5 for range queries on the DC-tree compared to the sequential search.

6. Conclusions

The data cube has quickly become a popular model for data warehouses. Recently, several multi-dimensional index structures have been proposed which store some derived information to efficiently support general range queries in data cubes. Typically, updates are collected and applied to the data warehouse periodically in a batch mode, e.g., over night. Then, all derived information has to be updated as well. The standard approach of bulk incremental updates to data warehouses has the following drawbacks. While the average runtime for one update is small, the total runtime for the whole batch of updates is rather large. Second, the contents of the data warehouse is not always up to date.

In this paper, we introduced the DC-tree, a fully dynamic index structure for data warehouses. This new index structure is designed for applications where the above drawbacks of the well-known approaches are critical. The DC-tree is a hierarchical index structure - similar to the X-tree - exploiting the concept hierarchies typically defined for the dimensions of a data cube. The DC-tree uses minimum describing sets and the partial ordering of the attribute values induced by the concept hierarchies instead of minimum bounding rectangles and an artificial total ordering. Furthermore, for each minimum describing set in the directory the values of the measure attributes are materialized. Our experimental performance evaluation on the TPC-D benchmark data demonstrated a significant speed-up compared to the X-tree and the sequential search when processing general range queries on a data cube.

Future work includes the following issues. The split algorithm of the DC-tree is rather expensive and, in particular, more expensive than the split algorithm of the X-tree. Therefore, alternative split algorithms should be investigated which have less than quadratic cost but nevertheless yield reasonably good splits. A data cube is typically implemented by using a relational DBMS. Thus, the DC-tree should be integrated into a commercial DBMS to evaluate its performance in this context.

Acknowledgements

We thank Jörg Sander for fruitful discussions in early phases of this project.

References

[BKK 96] Berchtold S., Keim D. A., Kriegel H.-P.: "*The X-Tree: An Index Structure for High-Dimensional Data*", Proc. 22th Int. Conf. on Very Large Data Bases, Bombay, India, 1996, pp. 28-39.

[GBLP 96] Gray J., Bosworth A., Layman A., Pirahesh H.: "*Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tabs and Subtotals*", Proc. 12th Int. Conf. on Data Engineering, 1996, pp. 152-159.

[CD 97] Chaudhuri, S., Dayal, U.: "*An Overview of Data Warehousing and OLAP Technology*", ACM SIGMOD Record 26(1), March 1997.

[GG 98] Gaede V., Günther O.: "*Multidimensional Access Methods*", ACM Computing Surveys, Vol. 30, No. 2, 1998, pp. 170-231.

[Gut 84] 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.

[HAMS 97] Ho Ch., Agrawal R., Megiddo N., Srikant R.: "*Range Queries in OLAP Data Cubes*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 73-88.

[HRU 96] Harinarayan V., Rajaraman A., Ullman J.D.: "*Implementing Data Cubes Efficiently*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1996, pp. 205-216.

[Huy 97] Huyn N.: "*Multiple-View Self-Maintenance in Data Warehousing Environments"*, Proc. 23rd Int. Conf. on Very Large Data Bases, Athens, Greece, 1997, pp. 26-35.

[MQM 97] Mumick I. S., Quass D., Mumick B. S.: "*Maintenance of Data Cubes and Summary Tables in a Warehouse"*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 100-111.

[OG 95] O'Neil P., Graefe G.: "*Multi-Table Joins through Bitmapped Join Indices*", SIGMOD Record 24(3), 1995, pp. 8-11.

[OQ 97] O'Neil P., Quass D.: "*Improved Query Performance with Variant Indices*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 38-49.

[RKR 97] Roussopoulos N., Kotidis Y., Roussopoulos M.: "*Cubetree: Organization of and Bulk Incremental Updates on the Data Cube*", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 89-99.

[TPC 98] Transaction Processing Council (TPC): "*TPC Benchmark D*", http://www.tpc.org, 1998.