Multidimensional Index Structures in Relational Databases

Article *in* Journal of Intelligent Information Systems · July 2000

DOI: 10.1023/A:1008729828172 · Source: DBLP

4 authors, including:

Christian Böhm
Ludwig-Maximilians-University of Munich
158 PUBLICATIONS **5,318** CITATIONS

SEE PROFILE

Stefan Berchtold
audimex gmbh
58 PUBLICATIONS **5,075** CITATIONS

SEE PROFILE

Peer Kröger
Christian-Albrechts-Universität zu Kiel
662 PUBLICATIONS **77,920** CITATIONS

SEE PROFILE

Multidimensional Index Structures in Relational Databases

CHRISTIAN BÖHM boehm@informatik.uni-muenchen.de
University of Munich, Munich, Germany

STEFAN BERCHTOLD Stefan.Berchtold@stb-gmbh.de
stb gmbh software technologie beratung, Augsburg, Germany

HANS-PETER KRIEGEL kriegel@informatik.uni-muenchen.de
University of Munich, Munich, Germany

URS MICHEL michel@informatik.uni-muenchen.de
University of Munich, Munich, Germany

Abstract. Efficient query processing is one of the basic needs for data mining algorithms. Clustering algorithms, association rule mining algorithms and OLAP tools all rely on efficient query processors being able to deal with high-dimensional data. Inside such a query processor, multidimensional index structures are used as a basic technique. As the implementation of such an index structure is a difficult and time-consuming task, we propose a new approach to implement an index structure on top of a commercial relational database system. In particular, we map the index structure to a relational database design and simulate the behavior of the index structure using triggers and stored procedures. This can be easily done for a very large class of multidimensional index structures. To demonstrate the feasibility and efficiency, we implemented an X-tree on top of Oracle 8. We ran several experiments on large databases and recorded a performance improvement up to a factor of 11.5 compared to a sequential scan of the database.

Keywords: multidimensional index, relational database, similarity search, range query.

1. Introduction

Efficient query processing in high-dimensional data spaces is an important requirement for many data analysis tools. Algorithms for knowledge discovery tasks such as clustering (Ester, *et al.* 1996; Sander, *et al.* 1998), association rule mining (Agrawal & Srikant, 1994) or OLAP (Ho, *et al.* 1997) are often based on range search or nearest neighbor search in multidimensional feature spaces. For instance, the well-known DBSCAN algorithm for cluster analysis performs a multidimensional range search for each point of the data set. Similarly, the algorithm for outlier detection by Knorr and Ng (1998) is based on repeated range queries. Since these applications deal with large amounts of usually high-dimensional point data, multidimensional index structures must be applied for the data management in order to achieve a satisfying performance.

Multidimensional index structures have been intensively investigated during the last decade. Most of the approaches (Guttman, 1984; Lomet & Salzberg, 1989) were designed in the context of geographical information systems where two-dimensional data spaces are prevalent. The performance of query processing often deteriorates when the dimensionality increases. To over-

come this problem, several specialized index structures for high-dimensional query processing have been proposed (Berchtold, *et al.* 1996; Berchtold, *et al.* 1998a; Berchtold, *et al.* 1998b). See (Böhm, 1998) for a comprehensive survey on the relevant techniques.

Recently, there is an increasing interest in integrating high-dimensional point data into commercial database management systems. Data to be analyzed often stem from productive environments which are already based on relational database management systems. These systems provide efficient data management for standard transactions such as billing and accounting as well as powerful and adequate tools for reports, spreadsheets, charts and other simple visualization and presentation tools. Relational databases, however, fail to manage high-dimensional point data efficiently for advanced data mining algorithms. Therefore, it is common to store productive data in a relational database system and to replicate the data for analysis purposes outside the database in file-based multidimensional index structures. We call this approach the *hybrid solution*.

The hybrid solution bears various disadvantages. Especially the integrity of data stored in two ways, inside and outside the database system, is difficult to maintain. If an update operation involving both, multidimensional and productive data fails in the relational database (e.g. due to concurrency conflicts), the corresponding update in the multidimensional index must be undone to guarantee consistency. Vice versa, if the multidimensional update fails, the corresponding update to the relational database must be aborted. For this purpose, a two-phase commit protocol for heterogeneous database systems must be implemented, a time-consuming task which requires a deep knowledge of the participating systems. The hybrid solution involves further problems. File systems and database systems usually have different concepts for data security, backup and concurrent access. File-based storage does not guarantee physical and logical data independence. Thus, schema evolution in "running" applications is difficult.

A promising approach to overcome these disadvantages is based on object-relational database systems. Object-relational database systems are relational database systems which can be extended by application-specific data types (called *data cartridges* or *data blades*). The general idea is to define data cartridges for multidimensional attributes and thus to manage them in the database. For data-intensive applications, it is necessary to implement the multidimensional index structures in the database. This requires the access to the block-manager of the database system, which is not granted by most commercial database systems. The current universal servers by ORACLE, Informix, and DB2, for instance, do not provide any documentation of a block-oriented interface to the database. Data cartridges (data blades) are only allowed to access relations via the SQL interface. Thus, current object-relational database systems are thus not very helpful for our integration problem.

We can summarize that using current object-relational database systems or pure relational database systems, the only possible way to store multidimensional attributes inside the database is to map them into the relational model. An early solution for the management of multidimensional data in relations is based on space-filling curves. Space-filling curves map points of a multidimensional space to one-dimensional values. This mapping is distance preserving in the sense that points which are close to each other in the multidimensional space are *likely* to be close to each other in the one-dimensional space. Although distance-preservation is not strict in

this concept, the search for matching objects is usually restricted to a limited area in the embedding space. Unfortunately, space filling curves are not very efficient in high-dimensional data spaces, particularly if the data points are highly skewed, correlated or clustered (Berchtold, *et al.* 1998a; Berchtold, *et al.* 1997a).

Therefore, we propose in this paper a technique which allows a direct mapping of the concepts of specialized index structures for high-dimensional data spaces into the relational model. For concreteness, here we concentrate on a relational implementation of the X-tree on top of Oracle-8. The X-tree, an R-tree variant for high-dimensional data spaces, is described in detail in section 4.1. The presented techniques, however, can also be applied to other indexing approaches such as the TV-Tree (Lin, *et al.* 1995) or the SS-Tree (White & Jain, 1996). Similarly, the underlying database system can be exchanged using the same concept we suggest.

The general idea is to model the structure of the relevant components of the index (such as data pages, data items, directory pages etc.) in the relational model and to simulate the query processing algorithms defined on these structures using corresponding SQL statements.

The rest of this paper is organized as follows: In the next section we explain the nature of multidimensional search and the corresponding problems. In section 3 and 4, we show how to implement the proposed solutions to multidimensional search in a commercial database system. Section 5 contains experimental results supporting our technique and section 6 concludes the paper.

2. Related Work

The problem of similarity search in a database can be stated as follows: We are given a large set of N data objects, stored in a relational database systems. The objects belong to a specific application domain such as multimedia, medical imaging etc. The objects are usually complex and can be stored as binary large objects (BLOB), i.e. from the DBS point of view, a data object is an uninterpreted sequence of bytes. The similarity of the objects is defined in an application-specific way and is coded in the application program. For instance, the similarity of color images may be determined by comparing the color distributions. Several similarity measures for various domains have been proposed in the past (Wallace & Wintz, 1980; Mehrotra & Gary, 1993; Agrawal, *et al.* 1995). The (dis-)similarity between two objects is also defined as an application-specific measure called *object distance*. The similarity search is defined as the retrieval of all objects with a specified maximal object distance to a given query object. A naive approach to this problem is to compare each object in the database with the query object. As we assume the number of objects to be large and the comparison operation to be expensive this naive approach lacks efficiency.

To handle similarity queries efficiently, usually a so-called feature transformation is applied. This approach extracts important properties from the objects in the database and transforms them into vectors of a d-dimensional vector space, the so-called feature space. Usually, the feature transformation is defined such that the distance between the feature vectors (the *feature distance*) either corresponds to the object distance or is, at least a lower bound thereof (*"lower bounding property"*). This way, the similarity search is naturally translated into a range query in the feature space. If the feature distance does not directly correspond to the object distance,

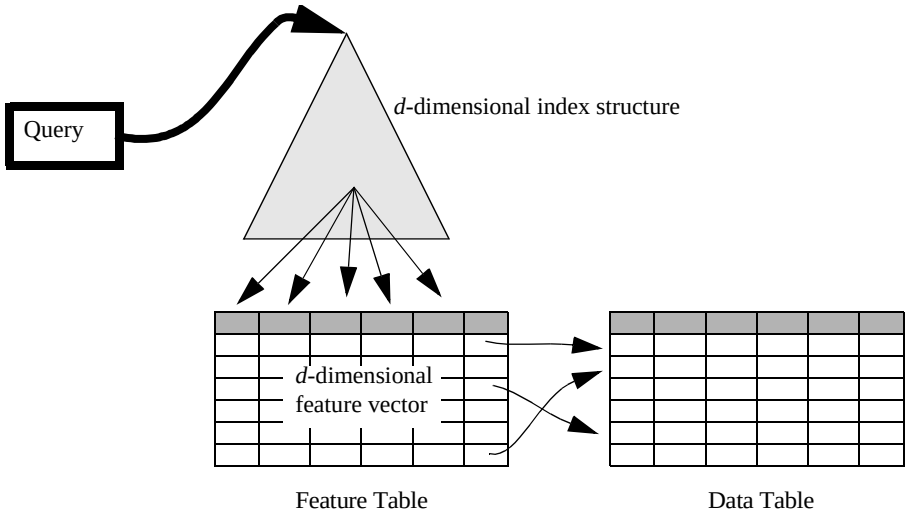

Figure 1: The Problem of Multidimensional Search

but is only a lower bound, the result of the range query is a set of candidates. It is guaranteed that each object satisfying the query is contained in the candidate set (*no false dismissals*) but there may be candidates which are not actual answers to the similarity query. As the candidates have to be tested in a so-called refinement step, the corresponding paradigm is called multi-step query processing. In case of color images, we might, for instance, use *d*-dimensional color histograms as feature vectors. In a relational implementation, we will typically hold the feature vectors in a separate table, the *feature table*. Figure 1 depicts this setup. Additionally, we will have an index on the primary key of the data table in order to speed up the join between the feature table and the data table. As this is self-evident, we omitted the index from the figure.

In order to search the database, we first query the feature table and produce a set of candidates that is complete, however might be too large. Thus, in a refinement step, we have to remove all the false hits from the answer set. This means that for each of the candidates we have to load the corresponding data item from the data table and perform some domain-dependent operation to test if the candidate is an actual hit. The performance of the whole search process is determined by two factors: the performance of the search on the feature table and the performance of the test for each candidate. As the second step is domain-dependent, the DBMS has only little influence on its performance. However, the first step is identical for any application and therefore, should be supported by the database system.

This allows us particularly to focus on the following problem: Given a set of N points in a d-dimensional space, how can we quickly search for points that fulfill a given query condition. The query condition could either be a multidimensional interval in which all points have to be located (range query), or it could be a query point and we are looking for the nearest neighbor

of this query point (nearest neighbor query). Both query types are useful for multimedia databases and it depends on the specific application which one will be used. In the following, we restrict our considerations to query processing on the feature table. The task we are focusing on in this paper is to efficiently determine the candidate set for the refinement step. The refinement step itself is ignored because it is too domain-specific.

Various solutions to the problem of multidimensional search have been proposed. First, we may classify the solutions according to the value *d*. If *d* is sufficiently small, e.g. 3, we are able to use index structures such as the grid file (Nievergelt, *et al.* 1984), the hB-tree (Lomet & Salzberg, 1989), the kd-tree (Bentley, 1975; Bentley, 1979) or the R^*-tree (Beckmann, *et al.* 1990). However, if *d* is quite large, e.g. 16, these index structures do not provide adequate performance. The reasons for this degeneration of performance are subsumed by the term "curse of dimensionality". The major problem in high-dimensional spaces is that most of the measures one can define in a *d*-dimensional vector space such as volume, area, or perimeter are exponentially depending on the dimensionality of the space. Thus, many techniques work only in low-dimensional spaces where we still have an exponential dependency. The exponent, however, is small enough.

To overcome these problems, a variety of specialized new index structures have been proposed in the past years, dealing with the problem of high-dimensional indexing. Examples are the TV-tree (Lin, *et al.* 1995), the SS-tree (White & Jain, 1996), the SR-tree (Katayama & Satoh, 1997), the X-tree (Berchtold, *et al.* 1996) or the Pyramid-tree (Berchtold, *et al.* 1998a).

Lin, Jagadish and Faloutsos (1995) presented the TV-tree which is an R-tree-like index structure. The central concept of the TV-tree is the telescope vector (*TV*). Telescope vectors divide attributes into three classes: attributes which are common to all data items in a subtree, attributes which are ignored and attributes which are used for branching in the directory. The major drawback of the TV-tree is that information about the behavior of single attributes, e.g. their selectivity, is required.

Another R-tree-like high-dimensional index structure is the SS-tree (White & Jain, 1996) which uses spheres instead of bounding boxes in the directory. Although the SS-tree clearly outperforms the R^*-tree, spheres tend to overlap in high-dimensional spaces.

The SR-tree, a variant of the SS-tree has been proposed by Katayama and Satoh (1997). The basic idea of the SR-tree is to use both hyperrectangles and hyperspheres as an approximation in the directory. Thus, a page region is described by the intersection of both objects.

Jain and White (1996) introduced the VAM-Split R-tree and the VAM-Split kD-tree. VAM-Split trees are rather similar to kD-trees, however in contrast to kD-trees, split dimensions are not chosen in a round robin fashion but depending on the maximum variance. VAM Split trees are constructed in main memory and then stored on secondary storage. Therefore, the size of a VAM-Split tree is limited by the available main memory.

In (Berchtold, *et al.* 1996), the X-tree has been proposed which is an index structure adapting the algorithms of R^*-trees to high-dimensional data using two techniques: First, the X-tree introduces an overlap-free split algorithm which is based on the split history of the tree. Second, if the overlap-free split algorithm would lead to an unbalanced directory, the X-tree omits the

split and the corresponding directory node becomes a so-called supernode. Supernodes are directory nodes which are enlarged by a multiple of the block size. A detailed description of the X-tree will be provided in section 4.

Most of those indexing techniques still are affected by the curse of dimensionality, but they allow to shift the edge of efficient indexing towards higher dimensions. In other words, although a low-dimensional index structure such as the R^*-tree only works efficiently up to 6-dimensions a high-dimensional index structure such as the X-tree works fine up to 12 dimensions. Only the Pyramid-tree (described below) is able to provide indexing power for an arbitrary dimensionality of space because its performance is independent of the dimension. However, this is only true for uniform data and fully specified range queries. For nearest neighbor queries, so far, no index structure has been proposed the performance of which is independent from the dimensionality of space. Rather, there seems to be analytical and experimental evidence (Berchtold, *et al.* 1997b) that, at least in the case of uniform data, there cannot exist an index structure that works equally well in all dimensions.

As a result, most recently, the VA-file (Weber, *et al.* 1998) was developed, an index structure that actually is not an index structure. The authors prove in the paper that under certain assumptions, above a certain dimensionality no index structure can process a nearest neighbor query efficiently. Thus, they suggest to speed-up the sequential scan rather than trying to fight a war that already is lost. The basic idea of the VA-file is to keep two files: a bit-compressed (quantized) version of the points and the exact representation of the points. Both files are unsorted, however, the ordering of the points in the two files is identical. Query processing is equivalent to a sequential scan of the compressed file with some look-ups to the second file whenever this is necessary. In particular, a look-up occurs if a point cannot be pruned from the nearest neighbor search which is only based on the compressed representation.

The idea of quantization based compression has also been integrated to index based query processing. The IQ-tree (Berchtold, *et al.* 2000) combines the ideas of a tree, a scan, and the quantization. The technique performs an I/O optimizing scan through the data pages if the index selectivity is not high enough to compensate for the cost of seek operations. In contrast to the VA-file, the quantization grid of the IQ tree is related to the data page regions and its resolution is automatically optimized during index construction and maintenance.

The Pyramid-Tree (Berchtold, *et al.* 1998a) is an index structure that, similarly to the Hilbert-technique (Faloutsos & Roseman, 1989; Jagadish, 1990), maps a *d*-dimensional point into a one-dimensional point and uses a B^+-tree to index the one-dimensional points. Obviously, queries have to be translated in the same way. In the data pages of the B^+-tree, the Pyramid-tree stores both, the *d*-dimensional points and the one-dimensional key. Thus, no inverse transformation is required and the refinement step can be done without look-ups to another file. The specific mapping used by the Pyramid-tree is called Pyramid-mapping. It is based on a special partitioning strategy that is optimized for range queries on high-dimensional data. The basic idea is to divide the data space such that the resulting partitions are shaped like peels of an onion. Such partitions cannot be efficiently stored by R-tree-like or kD-tree-like index structures. However, the Pyramid-tree achieves the partitioning by first dividing the *d*-dimensional space into $2d$ pyramids having the center point of the space as their top. In a second step, the

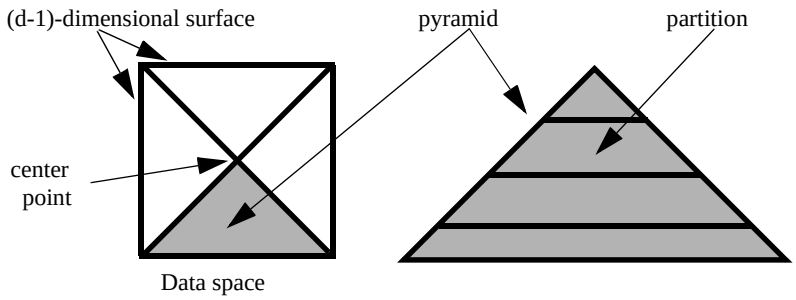

Figure 2: Partitioning the Data Space into Pyramids

single pyramids are cut into slices parallel to the basis of the pyramid forming the data pages. Figure 2 depicts this partitioning technique.

Summarizing all this work, there are two general approaches to multidimensional indexing: One can either solve the *d*-dimensional problem by designing a *d*-dimensional index. We refer to this class of indexing techniques as multidimensional indexes. Alternatively, one can map the *d*-dimensional problem to an equivalent one-dimensional problem and then make use of an existing one-dimensional index such as a B^+-tree. Thus, we provide a mapping that maps each *d*-dimensional data point into a one-dimensional value (key). We refer to this class of indexing techniques as mapping techniques.

All the above high-dimensional index structures except for the Pyramid-tree belong to the first category. Examples for the second category are the Z-order (Finkel & Bentley, 1974), the Hilbert-curve (Faloutsos & Roseman, 1989; Jagadish, 1990), Gray-Codes (Faloutsos, 1985) or the Pyramid-tree (Berchtold, *et al.* 1998a). Usually, the performance of the *d*-dimensional approach is slightly better, however, as the second category makes use of existing and proven technology, there exist some advantages, too: The mapping techniques can be implemented much easier and important issues such as recovery or concurrency control can be considered solved problems, as the techniques make use of existing B^+-tree indexes. The decision which index structure is appropriate for a given application is very complex and depends on the data distribution, the query mix, the size and dimensionality of the database. Therefore, this decision is not addressed in this paper.

3. Simulation of Mapping Techniques

In this chapter, we describe how to implement a multidimensional index structure based on a mapping to a one-dimensional space. The basic idea is to simulate the index structure by simply adding a new field for the key to the feature table and maintaining an index on this field. In order to process a query, we first determine a set of candidates based on the one-dimensional key. Then, we have an additional filter step based on the feature table. Finally, we consult the data table to compute the exact answer.

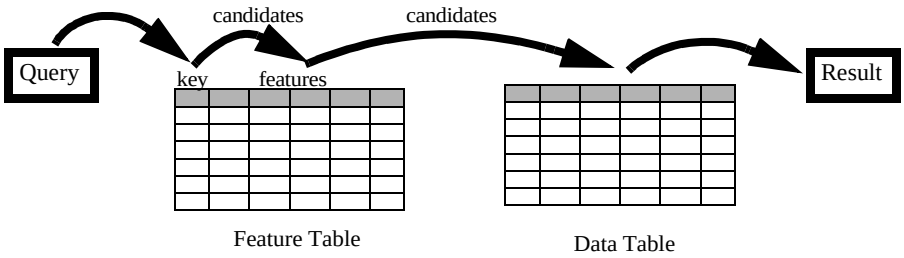

Figure 3: Query Processing using a Mapping Technique

3.1 The Principle of Mapping Techniques

Mathematically, there are two classes of mappings from a *d*-dimensional point to a one-dimensional key: bijective and non-bijective mappings. Bijective means that given a *d*-dimensional point it is possible to uniquely determine the corresponding one-dimensional value *and* vice versa. Examples for bijective mappings are the Z-order, the Hilbert-curve, and Gray codes. An example for a non-bijective mapping is the Pyramid mapping. A non-bijective mapping is a mapping that allows us to uniquely determine the one-dimensional key of a *d*-dimensional point, however, each key is shared by a set of *d*-dimensional points and thus, the inverse mapping is not unique.

In practice, this difference between bijective and non-bijective does not exist because bijective mappings are only bijective if infinite precision is applied. For practical reasons, the one-dimensional key is always restricted to a precision manageable by a single CPU command i.e. 64 bit maximum. To optimize query processing, the precision is further restricted. Therefore, keys are shared by a set of *d*-dimensional points and thus, all mentioned mappings are not invertible. As an implication, we cannot process a given query only using the one-dimensional key. But fortunately, one can at least guarantee that all the correct answers to any query are contained in the computed answers based on the one-dimensional keys although this answer set of candidates may be larger. Therefore, we have to refine the candidate set by taking the *d*-dimensional feature vectors into account.

Thus, the whole query process, also depicted in Figure 3, is done in three steps:

1. compute a set of candidates based on the one-dimensional key (1st filter step)
2. reduce this set of candidates based on the *d*-dimensional feature vectors (2nd filter step)
3. refine this set of candidates by looking up the actual data items (refinement step)

3.1 Relational Architecture

As shown in Figure 3, the only change we have to do in addition to the setup shown in Figure 1 is to add a single field "key" to the feature table. Furthermore, we have to add a trigger to the feature table such that whenever a tuple is added to (or updated in) the feature table, the corresponding key value will be set correctly. The complexity of this function is index-specific. Typically, the mappings can be implemented rather easily. As a second step, we add an (B^+-tree) index to the key field of the feature table to accelerate the access.

In order to process a range query, we first translate the query into a set of range queries on the key domain. Then, we are able to formulate the whole query processing in a single SQL statement:

select	"data fields"
from	"data table" d, "feature table" f
where	f.ptr=d.key **and**
	((f.key $\geq lb_1$ **and** f.key $\leq ub_1$) **or**
	...
	(f.key $\geq lb_k$ **and** f.key $\leq ub_k$))

In order to maintain the index structure in case of insert, update and delete operations, we simply add a trigger to the table that fires if any of these operations appears. The stored procedure invoked by the trigger simply computes the mapping and stores the result in the key field of the tuple. Thus, the implementation of the index structure is reduced to the pure implementation of the mapping itself. All the mappings proposed so far are simple bitmap operations or consist of only a few statements in a high-level programming language. Thus, they can easily be implemented using an functional extension of SQL such as PL/SQL in case of Oracle 8.

4. Simulation of Hierarchical Index Structures

The implementation of hierarchical index structures is much more complex than the implementation of mapping techniques. This applies to any implementation strategy. The reason for this is that hierarchical index structures have a complex structure that dynamically changes when inserting new data items. Thus, algorithms do not run on a previously given structure and have to be implemented recursively. To demonstrate that even in this complex scenario, an implementation of an index structure on top of a commercial database system can be done relatively easy and is preferable compared to a legacy implementation, we implemented the X-tree, a high-dimensional R-tree-based index structure. In the next section of this chapter we describe the most relevant details of the X-tree. Then, we show how an implementation can be done using a relational database system.

4.1 The X-Tree

The X-tree (Berchtold, *et al.* 1996) is structurally very similar to the family of R-trees (Guttman, 1984). As with R-trees, the directory of an X-tree is hierarchically organized. Each directory node contains a set of minimum bounding rectangles and a set of pointers to subnodes. The lowest level of the hierarchy consists of data pages. The data pages contain the actual data, in our case the data points. The minimum bounding rectangles are allowed to over-

lap in general, however, all rectangles in a subtree must be totally contained in the minimum bounding rectangle of the root node of this subtree. Thus, in order to process a query, say a range query, one simply follows all paths for which the minimum bounding rectangles intersect the range. In order to insert a data item, a specific strategy is used, depending on the type of the index structures. The problem thereby is that as the minimum bounding rectangles do not cover the whole space and are allowed to overlap, there exists some degree of freedom into which node a given new data item should be inserted. Another important algorithm for an R-tree-like index structure is the so-called split algorithm. The split algorithm is invoked, whenever a node overflows. In this case, we have to create a new node and we have to decide which entries of the old node stay in the node and which entries are transformed to the newly created node.

During the last years, a variety of R-tree-based index structures such as the R-tree, the R^*-tree, the R^+-tree, and others have been proposed. All these index structures have primarily been designed for the management of spatially extended, 2-dimensional objects, but have also been used for high-dimensional point data. Empirical studies (Berchtold, *et al.* 1996; Wallace & Wintz, 1980), however, showed a deteriorated performance of R^*-trees (the best index structure for 2-dimensional data) for high-dimensional data. The major problem of R-tree-based index structures in high-dimensional data spaces is overlap. In contrast to low-dimensional spaces, there exist only few degrees of freedom for splits in the directory. In fact, in most situations there exists only a single “good” split axis. An index structure that does not use this split axis will produce highly overlapping MBRs in the directory and thus show a deteriorated performance in high-dimensional spaces. Unfortunately, this specific split axis might lead to unbalanced partitions. In this cases, a split should be prevented in order to avoid underfilled nodes.

The X-tree is an extension of the R^*-tree which is particularly designed for the management of high-dimensional objects and based on the analysis of problems arising in high-dimensional data spaces. It extends the R^*-tree by two concepts:

- overlap-free split according to a split-history
- supernodes with an enlarged page capacity

The X-tree shows a high performance gain compared to R^*-trees for all query types in medium-dimensional spaces. For small dimensions, the X-Tree shows a behavior almost identical to R^*-trees, for higher dimensions the X-tree also has to visit such a large number of nodes that a linear scan is less expensive. It is impossible to provide exact values here because many factors such as the number of data items, the dimensionality, the distribution, and the query type have a high influence on the performance of an index structure.

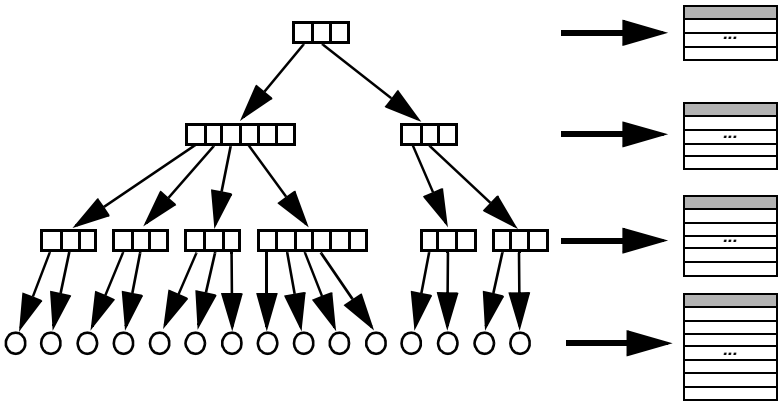

Figure 4: Relational Simulation of an X-tree

4.2 Simulation

The basic idea of our technique is to simulate the X-tree within the relational schema. Thus, we keep a separate table for each level of the tree. One of those tables stores the data points (simulating the data pages), the other tables store minimum bounding boxes and pointers (simulating the directory pages). Figure 4 depicts this scenario. In order to insert a data item, we first determine the data page in which the item has to be inserted. Then, we check if the data page overflows and if so, we split the page according to the X-tree split strategy. Note that a split might also cause the parent page in the directory to overflow. If we have to split the root node of the tree, i.e. the tree has to grow in height, we have to introduce an additional table and thus change the schema. A practical alternative is to pre-define tables for a three or four level directory. As only in case of very large databases, an X-tree grows beyond height four. By doing so, we can handle a split of the root node as an exception that has to be handled separately. Thus, the schema of the tree becomes static. All these actions are implemented in stored procedures.

In order to search the tree, we have to join all tables and generate a single SQL statement that queries the entire tree. This statement has to be created dynamically whenever the schema of the X-tree changes due to tree growth. If we process range queries, the statement is rather simple. The details are provided in section 4.4.

Relational Schema

All information usually held in the data pages of the X-tree is modeled in a relation called DATA. A tuple in DATA contains a *d*-dimensional data vector which is held in a set of *d* numerical attributes $x_0, x_1, ..., x_{d-1}$, a unique tuple identifier (*tid*), and the page number (*pn*) of the data page. Thus, DATA has the scheme “DATA (x_0 FLOAT, x_1 FLOAT, ..., x_{d-1} FLOAT, *tid* NUM-

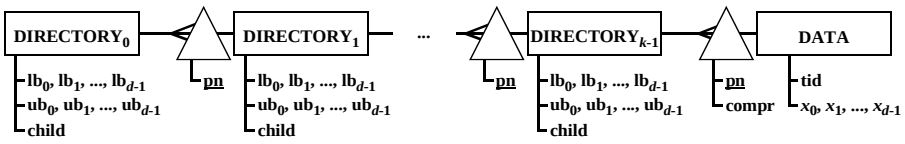

Figure 5: Relational Schema of the X-Tree

BER NOT NULL, *pn* NUMBER NOT NULL)". Intuitively, all data items located in the same data page of the X-Tree share the same value *pn*.

The *k* levels of the X-tree directory are modeled using *k* relations DIRECTORY$_0$, ..., DIRECTORY$_{k-1}$. Each tuple in a relation DIRECTORY$_i$ belongs to one entry of a directory node in level *i* consisting of a bounding box and a pointer to a child node. Therefore, DIRECTORY$_i$ follows the scheme "DIRECTORY$_i$ (lb_0 FLOAT, ub_0 FLOAT, ..., lb_{d-1} FLOAT, ub_{d-1} FLOAT, *child* NUMBER NOT NULL, *pn* NUMBER NOT NULL)". The additional attribute *child* represents the pointer to the child node which, in case of DIRECTORY$_{k-1}$, references a data page and *pn* identifies the directory node the entry belongs to. Thus, the two relations DIRECTORY$_{k-1}$ and DATA can be joined via the attributes *child* and *pn* which actually form a 1:*n*-relationship between DIRECTORY$_{k-1}$ and DATA. The same relationship exists for two subsequent directory levels DIRECTORY$_i$ and DIRECTORY$_{i+1}$. Obviously, it is important to make the join between two subsequent levels of the directory efficient. To facilitate index-based join methods, we create indexes using the *pn* attribute as the ordering criterion. The same observation holds for the join between DIRECTORY$_{k-1}$ and DATA. To save table accesses, we also added the quantized version of the feature vectors to the index. The resulting relational schema of the X-Tree enhanced by the required indexes (triangles) is depicted in Figure 5.

Compressed Attributes

If we assume a high-dimensional data space, the location of a point in this space is defined in terms of *d* floating point values. If *d* increases, the amount of information, we are keeping, also increases linearly. Intuitively however, it should be possible to keep the amount of information stored for a single data item almost constant for any dimension. An obvious way to achieve this is to reduce the number of bits used for storing a single coordinate linearly if the number of coordinates increases. In other words, because in a high-dimensional space, we have so much information about the location of a point, it should be sufficient to use a coarser resolution to represent the data space. This technique successfully has been applied in the VA-file (Weber, *et al.* 1998) to compute nearest neighbors. In the VA-file, a compressed version of the data points is stored in one file and the exact data is stored in another file. Both files are unsorted, however, the ordering of the points in the two files is identical. Query processing is equivalent to a sequential scan of the compressed file with some look-ups to the second file whenever this is necessary. In particular, a look-up occurs if a point cannot be pruned from the nearest neighbor search based on the compressed representation, only.

In our implementation of the X-tree, we suggest a similar technique of *compressed attributes*. A compressed attribute summarizes the *d*-dimensional information of an entry in the DATA table in a single-value representation. Thus, the resolution of the data space is reduced to 1 byte per coordinate. Then, the 1-byte coordinates are concatenated and stored in a single attribute called *comp*. Thus the scheme of DATA changes to DATA (REAL x_0, REAL x_1, REAL x_{d-1}, RAW[*d*] *comp*, INT *tid*, INT *pn*). To guarantee an efficient access to the compressed attributes, we store *comp* in the index assigned to DATA. Thus, in order to exclude a data item from the search, we first can use the compressed representation of the data item stored in the index and only if this is not sufficient, we have to make a look-up to the actual DATA table. This further reduces the number of accesses to the DATA table because most accesses are only to the index.

4.3 Index Creation

There are two situations for inserting new data into an index structure: Inserting a single data item, and building an index from scratch given a large amount of data (bulk-load). We will handle these two situations separately due to efficiency considerations. The reason for this is that a dynamic insert of a single data item is usually relatively slow, however, knowing all the data items to be inserted in advance, we are able to pre-process the data (e.g. sort) such that an index can be built immediately. This applies to almost all multidimensional index structures and their implementations.

The dynamic insertion of a single data item involves two steps: determining an insertion path and, possibly, a local restructuring of the tree. There are basically two alternatives for the implementation: An implementation of the complete insertion algorithm (e.g. using embedded SQL), or directly inserting the data point into the DATA relation and then raising triggers which perform the restructuring operation.

In any implementation, first we have to determine an appropriate data page to insert the data item into. Therefore, we recursively look-up the directory tables as we would do it in a legacy implementation. Using a stored procedure, we load all affected node entries into main memory and process them as described above. Then, we insert the data item into the page. In case of an overflow, we recursively update the directory, according to (Berchtold, *et al.* 1996).

If an X-Tree has to be created from scratch for a large data set, it is more efficient to provide a bulk-load operation, as proposed in (Berchtold, *et al.* 1998b). This technique can also be implemented in embedded SQL or stored procedures.

4.4 Processing Range Queries

Processing a range query using our X-Tree implementation with a *k*-level-directory involves (k + 2) steps. The first step reads the root level of the directory ($DIRECTORY_0$) and determines all pages of the next deeper level ($DIRECTORY_1$) which are intersected by the query window. These pages are loaded in the second step and used for determining the qualifying pages in the subsequent level. The following steps read all *k* levels of the directory in the same way, each filtering between pages which are affected or not. When the bottom level of the directory has been processed, the page numbers of all qualifying data pages are known. The data pages in our implementation contain the compressed (i.e. quantized) versions of the data vectors. Step number (k + 1), the last filter step loads these data pages and determines candidates (a candidate is a

```
SELECT data.*
FROM directory_0 dir_0, directory_1 dir_1, data
WHERE
    /* JOIN */
                dir_0.child = dir_1.pn
          AND   dir_1.child = data.pn
    /* 1st step*/
          AND   dir_0.lb_0 ≤ qub_0          AND   qlb_0 ≤ dir_0.ub_0
          AND   ...
          AND   dir_0.lb_d-1 ≤ qub_d-1      AND   qlb_d-1 ≤ dir_0.ub_d-1
    /* 2nd step */
          AND   dir_1.lb_0 ≤ qub_0          AND   qlb_0 ≤ dir_1.ub_0
          AND   ...
          AND   dir_1.lb_d-1 ≤ qub_d-1      AND   qlb_d-1 ≤ dir_1.ub_d-1
    /* 3rd step */
          AND   ASCII(SUBSTR(data.comp,1,1)) BETWEEN qclb_0 and qcub_0
          AND   ...
          AND   ASCII(SUBSTR(data.comp,1,1)) BETWEEN qclb_d-1 and qcub_d-1
    /* 4th step */
          AND   data.x_0 BETWEEN qlb_0 AND qub_0
          AND   ...
          AND   data.x_d-1 BETWEEN qlb_d-1 AND qub_d-1
```

Figure 6: An Example for an SQL Statement Processing a Range Query

point whose quantized approximation is contained in the query window). In the refinement step (k + 2), the candidates are directly accessed (the position in the data file is known) and tested for containment in the query window.

In our relational implementation, all these steps are comprised in a single SQL statement (cf. Figure 6 for a 2-level directory). It forms an equi-join between each pair of subsequent directory levels (DIRECTORY$_j$ and DIRECTORY$_{j+1}$, $0 \le j \le k-2$) and an additional equi-join between the last directory level DIRECTORY$_i$ and the DATA relation. The SQL statement consists of (k + 2) AND-connected blocks in the WHERE-clause. The blocks refer to the steps of range query processing as described above. For example, the first block filters all page numbers of the second directory level qualifying for the query. Block number (k + 1) contains various sub-string-operations. The reason is that we had to pack the compressed attributes into a string due to restrictions on the number of attributes which can be stored in an index. The last block forms the refinement step. Note that it is important to translate the query into a single SQL statement, because client-/server communication involving costly context switches or round-trip delays can be clearly reduced.

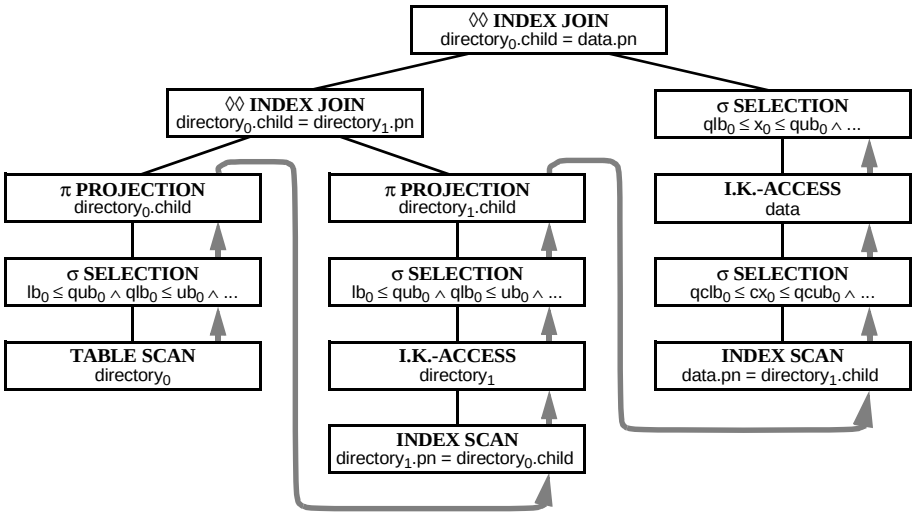

Figure 7: Sample Query Evaluation Plan for a Range Query

The particularity of our approach is that processing of joins between $(k + 1)$ tables is even more efficient than a single scan of the data relation provided that the SQL statement is transformed in a suitable query evaluation plan (*QEP*). This can be guaranteed by hints to the query optimizer. The *QEP* for our 2-level example is depicted in figure 7. For clarity, the information flow is marked by grey arrows. Query processing starts with a table scan of the root table. The page regions intersecting the query window are selected and the result is projected to the foreign key attribute *child*. The value of this result is used in the index join to efficiently search for the entries in DIRECTORY_1 contained in the corresponding page region. For this purpose, an index range scan is performed. The corresponding directory entries are retrieved by internal-key-accesses in the corresponding base table DIRECTORY_1. The qualifying data page numbers are again determined by selection and projection to the *child*-attribute. An index range scan similar to the index scan above is performed on the index of the DATA-table containing the page number, and the quantized version of the data points. Before accessing the exact representation of the data points, a selection based on the compressed attribute is performed to determine a suitable candidate set. The last step is the selection based on the exact geometry of the data points.

As it can be seen, the query evaluation plan is very similar to that for range query processing in the standard X-tree. In particular, each page access in the standard algorithm corresponds to the access of a sequence of tuples which is contiguously ordered in an index. The only difference is that a single page access to the X-tree does not correspond to a single page access in the relational index for two reasons: First, the assumed effective capacity of the X-tree may not be

identical to the effective capacity of the relational index. Second, even if the capacities are identical, the information on one X-tree page may be distributed among two pages of the relational index because we have no means to control the assignment of tuples to pages. Generally, one page access in the X-tree corresponds to *m* page accesses in the relational implementation, where

$$m = \frac{C_{\text{eff,X-tree}}}{C_{\text{eff,RelIndex}}} + 1 .$$

A relatively new feature of the used database system, clusters, should enable us to tackle this problem. First experimental results, however, did not yield substantial improvements over non-cluster indexes. In spite of this disadvantage, our technique outperforms standard query processing techniques such as the sequential table scan or the inverted list approach by an order of magnitude, as our experiments show.

4.5 Limitations

Our solution implements a multidimensional index by modelling it in a relational way. It is necessary to make assumptions about the physical organization of the relational model, i.e. certain order-preserving (one-dimensional) indexes must be created. The multidimensional query is transformed into an SQL query on that model. To perform efficiently, it is necessary that the appropriate indexes are used. To immitate the behavior of the multidimensional index, the join between the tables modelling the directory and the table modelling the data pages also must be evaluated using a particular join algorithm, the *indexed nested loop join* in an appropriate join order. In our experiments, the query optimizer generated a suitable query evaluation plan. Other optimizers, for instance rule-based query optimizers, could easily fail to generate good plans, since they could apply the push-selection rule. In such cases, the optimizer must be forced to generate a suitable plan by the use of optimizer-hints. These steps, however, destroy the principle of physical and logical data independence which is an important principle for database modeling. The corresponding code must be integrated into the application program, i.e. the use of the multidimensional index cannot be hidden from the user in the same way as the use of relational indexes is hidden from the user/application programmer.

A second limitation regards the efficiency of update operations which is out of the scope of the current paper. The use of triggers to update directory relations after insertion of new points or deletion of database points introduces a substantial overhead which makes the database application slow in the presence of a heavy update load. These problems can be tackled by collecting updates into separate relations or into separate areas of the current relations (a *default data page*). Then, the index could be reorganized from time to time.

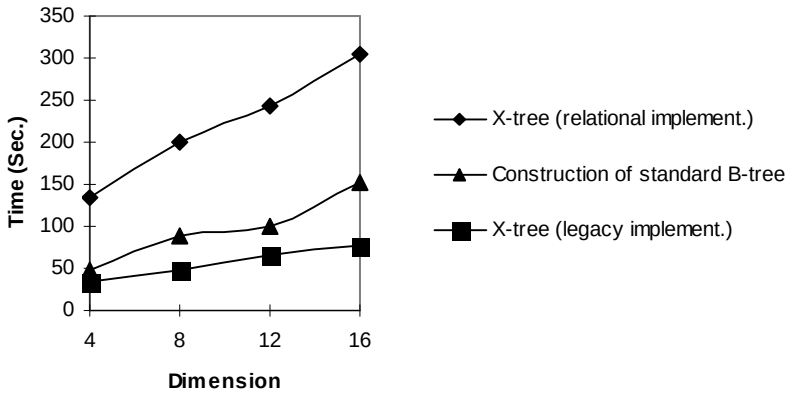

Figure 8: Times to Create an X-tree in Oracle 8

5. Experimental Evaluation

In order to verify our claims that an implementation of multidimensional index structures does not only provide advantages from a software engineering point of view but also in terms of performance, we actually implemented the X-tree on top of Oracle 8 and performed a comprehensive experimental evaluation on both, synthetic and real data. Therefore, we compared various query processing techniques for high-dimensional range queries in relational databases:

1. sequential scan on the data relation,
1. sequential scan on the data relation using the COMPRESSED attributes technique
1. standard index (B-tree) on the first attribute
1. standard index on all attributes concatenated in a single index
1. standard indexes on each attribute (also known as inverted-lists-approach)
1. X-tree-simulation with and without COMPRESSED attributes technique

As first experimental results showed, the variants 4. and 5. have a performance much worse than all other variants. We will therefore not show detailed results for these techniques.

In a first experiment, we determined the times for creating an X-tree on a large database. Therefore, we bulk-loaded the index using different techniques. The results of this experiment are shown in figure 8. Depending on the dimensionality of the data set, the relational implementation required, between one and five minutes to build an index on a 100,000 record 16-dimensional database. For this experiment, we used the algorithms described in (Berchtold, *et al.* 1998b) in order to bulk-load the X-tree caching intermediate results in a operating system file. The series “X-tree netto” shows the time consumed for the pre-processing of the bulk-loaded X-tree. The times for the standard B-tree approach and the X-tree approach

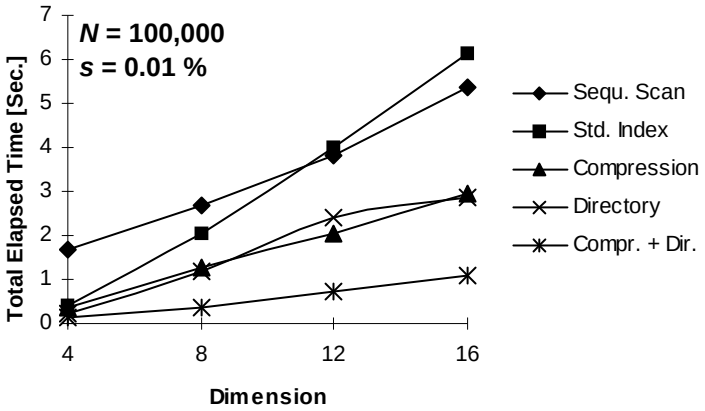

Figure 9: Performance for Range Queries (Synthetic Data) for Varying Dimensions

showed that a standard B-tree can be built about 2.5 times faster. However, both techniques showed a good overall performance.

In the next experiment, we compared the query performance of the different implementations on synthetic data. The result of an experiment on 100,000 data items of varying dimensionality is presented in figure 9 for a selectivity of s=0.01% (i.e. 10 results). Again, the total elapsed times of the inverted lists approach and the standard index on a single attribute are not presented due to their bad performance. It can be seen, that both, the compressed attributes technique

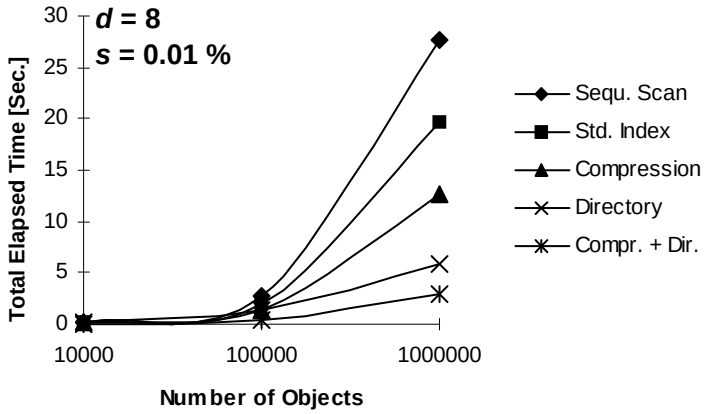

Figure 10: Performance for Range Queries (Synthetic Data) for Varying Database Sizes

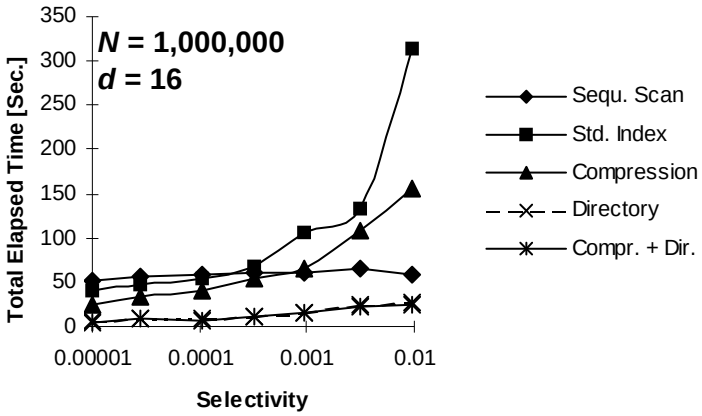

Figure 11: Performance for Range Queries (Synthetic Data) for Varying Selectivities

(denoted by *compression*) and the X-tree simulation (denoted by *directory*) yield high performance gains over all experiments. Moreover, the combination of both outperforms the sequential scan and the standard index for all types of data over all dimensions. It can also be seen that the combination of the directory and the compressed attributes technique yields a much better improvement factor than each single technique. The factor even improves for higher dimensions, the best observed improvement factor in our experiments was 11.46.

In the experiment depicted in figure 10, we investigated the performance of the implementations when varying the size of the database. Again, the relational implementation of the X-tree with compressed attributes (denoted by *compr.+dir.*) outperformed all other techniques by far. The acceleration even improves with growing database size.

In a last experiment on real data, we looked at the performance for varying selectivities. The results of this experiment on 1,000,000 16-dimensional feature vectors are shown in figure 11. We used data from a similarity search system of a car manufacturer where each feature vector describes the shape of a part used in one of the cars of the manufacturer. As we can see from the chart, our technique outperformed all other techniques, however, the effect of the compressed attributes was almost negligible. Thus, the performance of the X-tree with and without compressed attributes was almost identical. Summarizing, this confirms our assumption that implementing index structures on top of a commercial relational database system shows very good performance for both, synthetic and real data.

6. Conclusions

In this paper, we proposed a new approach to implement an index structure on top of a commercial relational database system. We map the particular index structure to a relational database design and simulate the behavior of the index structure using triggers and stored procedures. We showed that this can be done easily for a very large class of multidimensional index structures. To demonstrate the feasibility and efficiency, we implemented an X-tree on top of Oracle 8. We ran several experiments on large databases and recorded a performance improvement of up to a factor of 11.46 compared to a sequential scan of the database.
In addition to the performance gain, our approach has all the advantages of using a fully-fledged database system including recovery, multi-user support and transactions. Furthermore, the development times are significantly shorter than in a legacy implementation of an index.

7. References

Agrawal R., Lin K., Sawhney H., Shim K. (1995). “Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases” in {Proc. 21st Int. Conf. on Very Large Databases}, pp. 490-501.

Agrawal R., Srikant R. (1994). "Fast Algorithms for Mining Association Rules" in {Proc. 20st Int. Conf. on Very Large Databases}, Chile, pp. 487-499.

Beckmann N., Kriegel H.-P., Schneider R., Seeger B. (1990). "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Atlantic City, NJ, pp. 322-331

Bentley J.L. (1975). "Multidimensional Search Trees Used for Associative Searching”, {Communications of the ACM}, Vol. 18, No. 9, pp. 509-517.

Bentley J. L. (1979). "Multidimensiuonal Binary Search in Database Applications", {IEEE Trans. Software Eng.}, Vol. 4, No. 5, pp. 397-409.

Berchtold S., Böhm C., Braunmüller B., Keim D. A., Kriegel H.-P. (1997a). "Fast Parallel Similarity Search in Multimedia Databases" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Tucson, AZ, pp. 1-12.

Berchtold S., Böhm C., Jagadish H.V., Kriegel H.-P., Sander J. (2000). “Independent Quantization: An Index Compression Technique for High-Dimensional Spaces” in {Proc. Int. Conf. on Data Engineering}

Berchtold S., Böhm C., Keim D., Kriegel H.-P. (1997b). "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space" in {ACM PODS Symposium on Principles of Database Systems}, Tucson, AZ, pp. 78-86.

Berchtold S., Böhm C., Kriegel H.-P. (1998a). "The Pyramid-Technique: Towards indexing beyond the Curse of Dimensionality" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Seattle, WA, pp. 142-153.

Berchtold S., Böhm C., Kriegel H.-P. (1998b). "Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations" in {6th. Int. Conf. on Extending Database Technology}, Valencia, Spain, pp. 216-230.

Berchtold S., Keim D., Kriegel H.-P. (1996). "The X-tree: An Index Structure for High-Dimensional Data" in {Proc. 22nd Int. Conf. on Very Large Data Bases}, Mumbay, India, pp. 28-39.

Böhm C. (1998). "Efficiently Indexing High-Dimensional Data Spaces”, Ph.D. Thesis, Faculty for Mathematics and Computer Science, University of Munich.

Ester M., Kriegel H.-P., Sander J., Xu X. (1996). "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Nois" in {Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining}, Portland, OR, pp. 226-231.

Faloutsos C. (1985). "Multiattribute Hashing Using Gray Codes" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, pp. 227-238.

Faloutsos C., Roseman S. (1989). "Fractals for Secondary Key Retrieval" in {Proc. 8th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems}, pp. 247-252.

Finkel R., Bentley J. L. (1974). “Quad Trees: A Data Structure for Retrieval of Composite Keys", {Acta Informatica}, Vol. 4, No. 1, pp. 1-9.

Guttman A. (1984). "R-trees: A Dynamic Index Structure for Spatial Searching" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Boston, MA, pp. 47-57.

Hjaltason G. R., Samet H. (1995). "Ranking in Spatial Databases" in {Proc. 4th Int. Symp. on Large Spatial Databases}, Portland, ME, pp. 83-95.

Ho C. T., Agrawal R., Megiddo N., Srikant R. (1997). “Range Queries in OLAP Data Cubes” in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Tucson, AZ, pp. 73-88

Jagadish H. V. (1990). "Linear Clustering of Objects with Multiple Attributes" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, Atlantic City, NJ, pp. 332-342.

Jain R, White D. A. (1996). "Similarity Indexing: Algorithms and Performance" in {Proc. SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670}, San Jose, CA, pp. 62-75.

Katayama N., Satoh S. (1997). "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries" in {Proc. ACM SIGMOD Int. Conf. on Management of Data}, pp. 369-380.

Knorr E.M. and Ng R.T. (1998). "Algorithms for Mining Distance-Based Outliers in Large Datasets" in {Proc. 24th Int. Conf. on Very Large Databases}, New York City, pp. 392-403.

Lin K., Jagadish H. V., Faloutsos C. (1995). "The TV-Tree: An Index Structure for High-Dimensional Data", {VLDB Journal}, Vol. 3, pp. 517-542.

Lomet D., Salzberg B. (1989). "The hB-tree: A Robust Multiattribute Search Structure" in {Proc. 5th IEEE Int. Conf. on Data Engineering}, Los Angeles, CA, pp. 296-304.

Mehrotra R., Gary J. (1993). "Feature-Based Retrieval of Similar Shapes" in {Proc. 9th Int. Conf. on Data Engeneering}, Vienna, Austria, pp. 108-115.

Nievergelt J., Hinterberger H., Sevcik K. C. (1984). "The Grid File: An Adaptable, Symmetric Multikey File Structure" in {ACM Trans. on Database Systems}, Vol. 9, No. 1, pp. 38-71.

Sander J., Ester M., Kriegel H.-P., Xu X.: “Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and its Applications”, in: {Data Mining and Knowledge Discovery}, Kluwer Academic Publishers, Vol. 2, No. 2.

Wallace T., Wintz P. (1980). "An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors", {Computer Graphics and Image Processing}, Vol. 13, pp. 99-126.

Weber R., Schek H.-J., Blott S. (1998). "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces" in {Proc. Int. Conf. on Very Large Databases}, New York, pp.194-205.

White D.A., Jain R. (1996). "Similarity indexing with the SS-tree" in {Proc. 12th Int. Conf on Data Engineering}, New Orleans, LA, pp.516-523.