Int. Conf. on Data Warehousing and Knowledge Discovery DaWaK '99

Implementation of Multidimensional Index Structures for Knowledge Discovery in Relational Databases

Stefan Berchtold[1] Christian Böhm[1,2], Hans-Peter Kriegel[2] and Urs Michel[2]

[1] stb gmbh, Ulrichsplatz 6, 86150 Augsburg, Germany
[2] University of Munich, Oettingenstr. 67, D-80538 Munich, Germany
Stefan.Berchtold@stb-gmbh.de
{boehm,kriegel,michel}@informatik.uni-muenchen.de

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 structures 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 easily be 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 of up to a factor of 11.5 compared to a sequential scan of the database.

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 [EKSX 98], association rule mining [AS 94], or OLAP [HAMS 97], are often based on range search or nearest neighbor search in multidimensional feature spaces. 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 satisfactory performance.

Multidimensional index structures have been intensively investigated during the last decade. Most of the approaches [Gut 84, LS 89] 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 overcome this problem, several specialized index structures for high-dimensional query processing have been proposed that fall into two general categories: One can either solve the *d*-dimensional problem by designing a *d*-dimensional index. Examples are the TV-tree [LJF 95], the SS-tree [WJ 96], the SR-tree [KS 97] or the X-tree [BKK 96]. We refer to this class of indexing techniques as multidimensional indexes. Alternatively, one can map the *d*-dimensional problem to an equivalent 1-dimensional problem and then make use of an existing 1-dimensional index such as a B^+-tree. Thus, we provide a mapping that maps each *d*-dimensional data point into a 1-dimensional value (key). We refer to this class of indexing techniques as mapping techniques. Examples for this category are the Z-order [FB 74], the Hilbert-curve [FR 89, Jag 90], Gray-

Codes [Fal 85], or the Pyramid-tree [BBK 98]. We refer to [Böh 98] 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 knowledge discovery 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 to manage them in the database. For data-intensive applications it is necessary to implement 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 server by ORACLE, for instance, does not provide any documentation of a block-oriented interface to the database. Data cartridges are only allowed to access relations via the SQL interface. 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.

In this paper, we propose 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, we concentrate here 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 [LJF 95] or the SS-Tree

[WJ 96]. 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 simulation of mapping techniques is pretty straightforward and is therefore not explained in depth in this paper. One just stores the 1-dimensional value in an additional column of the data table and then searches this column. Obviously, a database index is used to support the search. Thus, the whole query process is done in three steps:

1. compute a set of candidates based on the 1-dimensional key
2. refine this set of candidates based on the *d*-dimensional feature vectors
3. refine this set of candidates by looking up the actual data items

2. 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 index structure, based on R-trees.

2.1 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 1 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 whether the data page overflows and if it does, 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 which causes the tree to grow in height, we have to introduce an additional table[1] 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

1. A technical problem arises here when dealing with commercial database systems: Oracle 8, for instance, ends a transaction whenever a DDL command is executed. This means that if we use Oracle 8, an insert operation on a tree that caused the root node to be split cannot be undone by simply aborting the current transaction.

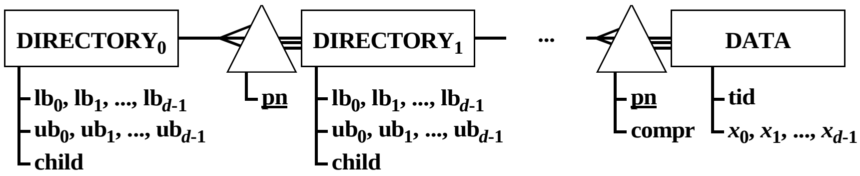

Figure 1: Relational Schema (Including B^+-Tree Indexes) of the X-Tree.

the schema of the X-tree changes due to tree growth. If we process range queries, the SQL statement is rather simple. The details are provided in section 2.3.

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 schema “DATA (x_0 FLOAT, x_1 FLOAT, ..., x_{d-1} FLOAT, *tid* NUMBER 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$_i$ is of 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 1.

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, as 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 [WSB 98] 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 only based on the compressed representation.

In our implementation of the X-tree, we suggest the 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 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.

2.2 Index Creation

There are two situations when one intends to insert 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 are supposed to 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 preprocess the data (e.g. sort) such that an index can be built very efficiently. 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, when necessary, a local restructuring of the tree. There are basically two alternatives for the implementation: An implementation of the whole insert algorithm (e.g. using embedded SQL), or directly inserting the data point into the DATA relation and then to raise triggers which perform the restructuring operation.

In any implementation, we first have to determine an appropriate data page to insert the data item. Therefore, we recursively look-up the directory tables as we would handle 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 [BKK 96].

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, such as proposed in [BBK 98a]. This technique can also be implemented in embedded SQL or stored procedures.

2.3 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

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$_0$ $\leq$ qub$_0$ANDqlb$_0$ $\leq$ dir$_0$.ub$_0$
        AND ...
        AND dir$_0$.lb$_{d\text{-}1}$ $\leq$ qub$_{d\text{-}1}$ANDqlb$_{d\text{-}1}$ $\leq$ dir$_0$.ub$_{d\text{-}1}$
    /* 2nd step */
        AND dir$_1$.lb$_0$ $\leq$ qub$_0$ANDqlb$_0$ $\leq$ dir$_1$.ub$_0$
        AND ...
        AND dir$_1$.lb$_{d\text{-}1}$ $\leq$ qub$_{d\text{-}1}$ANDqlb$_{d\text{-}1}$ $\leq$ dir$_1$.ub$_{d\text{-}1}$
    /* 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$_{d\text{-}1}$ and qcub$_{d\text{-}1}$
    /* 4th step */
        AND data.x$_0$ BETWEEN qlb$_0$ AND qub$_0$
        AND ...
        AND data.x$_{d\text{-}1}$ BETWEEN qlb$_{d\text{-}1}$ AND qub$_{d\text{-}1}$

Figure 2: An Example for an SQL Statement Processing a Range Query.

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, thus filtering between pages which are affected or not. Once 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 point whose quantized approximation is intersected by 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 (c.f. Figure 2 for a 2-level directory). It forms an equi-join between each pair of subsequent directory levels (DIRECTORY$_j$ and DIRECTORY$_{j+1}$, $0 \leq j \leq k - 2$) and an additional equi-join between the last directory level DIRECTORY$_i$ and the DATA relation. It 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 substring-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.

The particularity of our approach is that processing of joins between $(k + 1)$ tables is more efficient than a single scan of the data relation provided that the SQL statement is transformed into a suitable query evaluation plan (*QEP*). This can be guaranteed by hints to the query optimizer. 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 which are 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 on 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.

3. Experimental Evaluation

In order to verify our claims that the suggested 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,
2. sequential scan on the data relation using the COMPRESSED attributes technique
3. standard index (B-tree) on the first attribute
4. standard index on all attributes concatenated in a single index
5. standard indexes on each attribute (inverted-list approach)
6. X-tree-simulation with and without COMPRESSED attributes technique

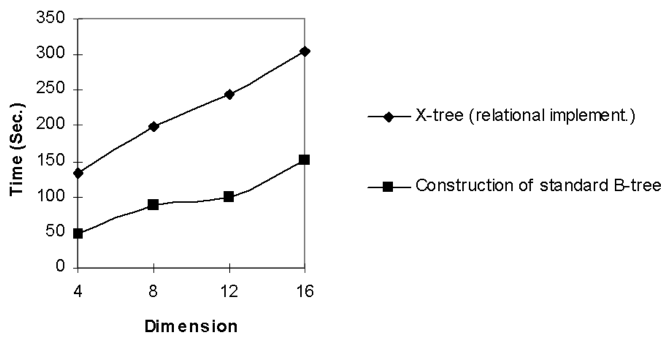

Figure 3: Times to Create an X-tree in Oracle 8.

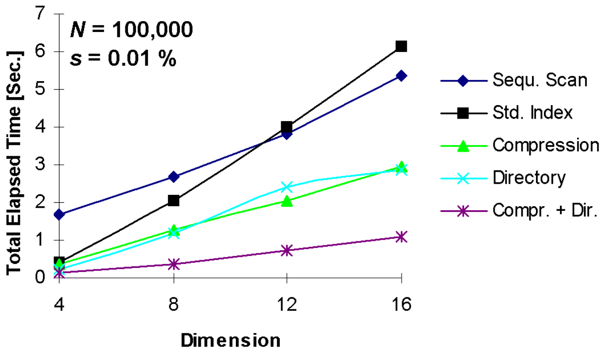

Figure 4: Performance for Range Queries (Synthetic Data) for Varying Dimensions.

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

In our first experiment, we determine the times for creating an X-tree on a large database. Therefore, we bulk-load the index using different techniques. The results of this experiment are shown in figure 3. The relational implementation requires, depending on the dimensionality of the data set, between one and five minutes to build an index on a 100,000 record 16-dimensional database. For this experiment, we use the algorithms described in [BBK 98a] for bulk-loading the X-tree caching intermediate results in a operating system file. The times for the standard B-tree approach and the X-tree approach show that a standard B-tree can be built about 2.5 times faster. However, both techniques yield a good overall performance.

In the next experiment, we compare 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 4. The performance of the inverted lists approach and the standard index on a single attribute is not presented due to bad performance. It can be seen that both, the compressed attributes technique and the X-tree simulation yield high performance gains over all experiments. Moreover, the combination of both these techniques outperforms the sequential scan and the standard index for

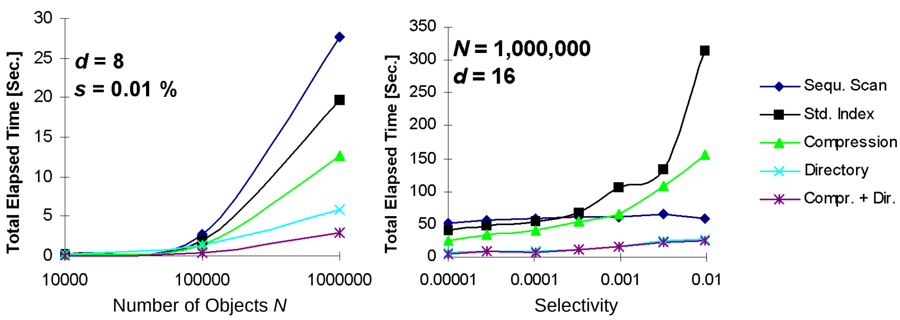

Figure 5: Performance for Range Queries (Synthetic Data) for Varying Database Size (a) and for Varying Selectivity (b).

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 11.5.

In the experiment depicted in figure 5a, we investigate the performance of the implementations when varying the size of the database. Again, the relational implementation of the X-tree with compressed attributes outperforms all other techniques by far. The acceleration even improves with growing database size.

In the last experiment on real data, we investigate the performance for varying selectivities. The results of this experiment on 1,000,000 16-dimensional feature vectors are shown in figure 5b. The data comes from a similarity search system of a car manufacturer and each feature vector describes the shape of a part. As we can observe from the chart, our technique outperforms all other techniques. The effect of the compressed attributes, however, was almost negligible. Thus, the performance of the X-tree with and without compressed attributes is almost identical. This confirms our claim that implementing index structures on top of a commercial relational database system shows very good performance for both, synthetic and real data.

4. 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 11.5 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.

References

[ALSS 95] Agrawal R., Lin K., Sawhney H., Shim K.: *'Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases'*, Proc. of the 21st Conf. on Very Large Databases, 1995, pp. 490-501.

[AS 94] Agrawal R., Srikant R.: '*Fast Algorithms for Mining Association Rules*', Proc. of the 20st Conf. on Very Large Databases, Chile, 1995, pp. 487-499.

[BBB+ 97] Berchtold S., Böhm C., Braunmueller B., Keim D. A., Kriegel H.-P.: *'Fast Similarity Search in Multimedia Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona.

[BBK 98] Berchtold S., Böhm C., Kriegel H.-P.: *'The Pyramid-Technique: Towards indexing beyond the Curse of Dimensionality'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Seattle, pp. 142-153,1998.

[BBK 98a] Berchtold S., Böhm C., Kriegel H.-P.: *'Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations'*, 6th. Int. Conf. on Extending Database Technology, Valencia, 1998.

[BBKK 97] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: '*A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*', ACM PODS Symposium on Principles of Database Systems, 1997, Tucson, Arizona.

[Ben 75] Bentley J.L.: '*Multidimensional Search Trees Used for Associative Searching*', Communications of the ACM, Vol. 18, No. 9, pp. 509-517, 1975.

[Ben 79] Bentley J. L.: *'Multidimensiuonal Binary Search in Database Applications'*, IEEE Trans. Software Eng. 4(5), 1979, pp. 397-409.

[BKK 96] Berchtold S., Keim D., Kriegel H.-P.: '*The X-tree: An Index Structure for High-Dimensional Data*', 22nd Conf. on Very Large Databases, 1996.

[BKSS 90] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990.

[Böh 98] Böhm C.: *'Efficiently Indexing High-Dimensional Data Spaces'*, Ph.D. Thesis, Faculty for Mathematics and Computer Science, University of Munich, 1998.

[EKSX 98] Ester M., Kriegel H.-P., Sander J., Xu X.: *'Incremental Clustering for Mining in a Data Warehousing Environment*', Proc. 24th Int. Conf. on Very Large Databases (VLDB '98), NY, 1998, pp. 323-333.

[Fal 85] Faloutsos C.: *'Multiattribute Hashing Using Gray Codes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1985, pp. 227-238.

[FB 74] Finkel R, Bentley J.L. *'Quad Trees: A Data Structure for Retrieval of Composite Keys'*, Acta Informatica 4(1), 1974, pp. 1-9.

[FR 89] Faloutsos C., Roseman S.: *'Fractals for Secondary Key Retrieval'*, Proc. 8th ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, 1989, pp. 247-252.

[Gut 84] Guttman A.: '*R-trees: A Dynamic Index Structure for Spatial Searching*', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1984.

[HAMS 97] Ho C.T., Agrawal R., Megiddo N., Srikant R.: Range Queries in OLAP Data Cubes. SIGMOD Conference 1997: 73-88

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

[Jag 90] Jagadish H. V.: *'Linear Clustering of Objects with Multiple Attributes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 332-342.

[JW 96] Jain R, White D.A.: *'Similarity Indexing: Algorithms and Performance'*, Proc. SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670, San Jose, CA, 1996, pp. 62-75.

[KS 97] Katayama N., Satoh S.: *'The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 369-380.

[LJF 95] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-Tree: An Index Structure for High-Dimensional Data'*, VLDB Journal, Vol. 3, pp. 517-542, 1995.

[LS 89] Lomet D., Salzberg B.: *'The hB-tree: A Robust Multiattribute Search Structure'*, Proc. 5th IEEE Int. Conf. on Data Eng., 1989, pp. 296-304.

[MG 93] Mehrotra R., Gary J.: *'Feature-Based Retrieval of Similar Shapes'*, Proc. 9th Int. Conf. on Data Engeneering, 1993.

[NHS 84] Nievergelt J., Hinterberger H., Sevcik K. C.: *'The Grid File: An Adaptable, Symmetric Multikey File Structure'*, ACM Trans. on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[WJ 96] White D.A., Jain R.: *'Similarity indexing with the SS-tree'*, Proc. 12th Int. Conf on Data Engineering, New Orleans, LA, 1996.

[WSB 98] Weber R., Scheck H.-J., Blott S.: '*A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces*', Proc. Int. Conf. on Very Large Databases, New York, 1998.

[WW 80] Wallace T., Wintz P.: *'An Efficient Three-Dimensional Aircraft Recognition Algorithm Using Normalized Fourier Descriptors'*, Computer Graphics and Image Processing, Vol. 13, pp. 99-126, 1980.