Enhancement of Oracle's Indexing Capabilities through GiST-implemented Access Methods

Mario Doeller
Harald Kosch

Institute of Information Technology
University Klagenfurt
Technical Reports
No TR/ITEC/02/2.09, April 2002

Enhancement of Oracle's Indexing Capabilities through GiST-implemented Access Methods

Mario Doeller
Harald Kosch

Institute of Information Technology
University Klagenfurt
Technical Report No TR/ITEC/02/2.09
April 2002

Abstract

Indexing is one of the most important tool for efficiency enhancement, available in DMBS today. The necessity of speeding up queries in very large database systems is unquestioned. The use of new-fashioned technologies and applications such as Multimedia, E-Commerce or Geographic Information Systems in combination with commercial database systems show the shortness of current indexing mechanism. This technical report introduces a possible way out of this dilemma. We make use of Oracle's Data Cartridge technology in combination with the GiST-framework, so that we are able to index any kind of data with any kind of balanced index trees.

Keywords: Multimedia Databases, Access Methods, Data Cartridge, Generalized Search Trees (GiST),

Contents

1 Introduction

Indexing mechanism are an important component of modern database management systems to enhance processing efficiency. Innately, most database systems only provide a limited number of integrated access methods such as B-tree or hashing facilities. These limited techniques shorten the use of database systems in new-fashioned applications such as Multimedia, E-Commerce or Geographic Information Systems (GIS). Available DBMSs provide some multimedia database extension packages such as DataBlades (Informix) or Extenders (IBM-DB2) but in most cases they do not fit to the user's need exactly. Especially, they rarely handle indexing of d-dimensional data (d>2) which is common in multimedia areas. It is not unusual to have images which are represented as color histograms with a dimensionality of 64 or higher. In this context several access methods for high dimensional data have been proposed, e.g., the SS-tree [10], SR-tree [11], M-tree [12], X-tree [13] or TV-tree [14]. A good survey on multidimensional access methods is given in [15] and [16]. Unfortunately, these access methods are not available in common DMBS. A first attempt in this direction was realized by M. Kornacker [18] by integrating an adapted GiST framework [2] into Informix Dynamic Server with Universal Data Option (IDS/UDO). This approach shows that it is feasible to enhance existing DBMS by extensible indexing mechanism. Furthermore, measurements have demonstrated that this approach can improve performance incredible.
We used for our multimedia extension the Oracle database. Besides some extension packages for text or spatial data, Oracle provides a mechanism for extending the capabilities of the server. This mechanism is called *Data Cartridge* [9] which allows a developer to extend for example the type system, server execution environment, indexing capabilities, optimizer and so on. This technical report concentrates on enhancing the indexing properties of an Oracle DBMS by the use of *Data Cartridges* and the generalized search trees (GiST) framework [2, 3]. The GiST framwork is explained in section 2. This enhancement is used for our CODAC project which stands for *Modelling and Querying* **CO***ntent* **D***escription and Quality* **A***daptation* **C***apabilites for Audio-Visual Data.* Further information can be obtained from `http://www-itec.uni-klu.ac.at/~harald/codac/`.

2 GiST Framework

The theory and implementation of the GiST framework was realized by J. H. Hellerstein and his group at the University of California, Berkeley[1]. The GiST framework enables an access method developer to build any kind of balanced index tree on any kind of data by implementing some specific methods for insertion, deletion and search. The current GiST version, 2.0, already includes source code[2] for the R-tree, R*-tree, SS-tree, SR-tree, SP-tree and B-tree. The GiST framework is widely used in the research community. The authors in [17] used an earlier version of the framework for implementing and testing their enhancement of a R-tree. A technical report about benchmarks of GiSTs can be found at the University of Alberty [1]. Furthermore the GiST framework is used in an image indexing and retrieval application, called Blobworld [6, 7, 8]. Besides, two Russian researcher[3] are currently working on an integration of the framework with the Postgres database.

[1] `http://www.berkeley.edu/`
[2] `http://gist.cs.berkeley.edu/`
[3] `http://www.sai.msu.su/~megera/postgres/gist/`

A GiST is a balanced tree with (key, RID) pairs in the leaves and (predicate, child page pointer) pairs as internal nodes. The framework and their corresponding trees have no restrictions on the key data stored within the tree or their organization within and across nodes. The framework itself is extensible by new access methods. This can be realized by implementing methods for maintaining the resulting tree consistent, consolidating tree nodes, measuring the effect of node update and assigning the distribution of data items, once a node split occurs. Once the tree is set, it can be improved by an additional tool in the GiST framework environment, the amdb [5, 4]. Amdb is a tool for designing, debugging, analyzing and performance measuring of GiST implemented access methods.
Figure 1 shows the main advantage of enhancing a DBMS with the GiST framework (right side of figure). A GiST-enhanced DBMS can easily be upgraded with any kind of newly developed access method. These new access methods can be used for adapted data types in databases and their applications.

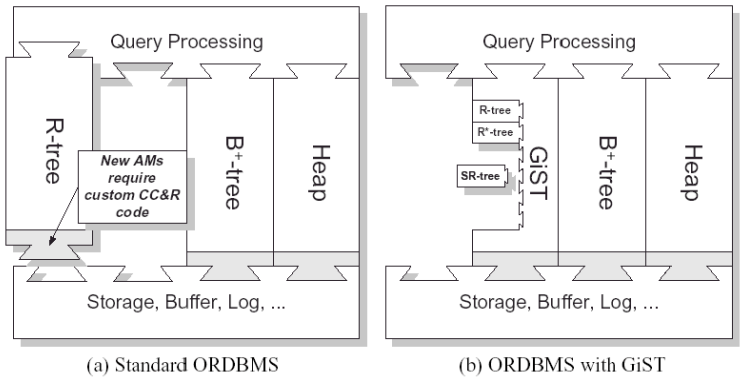

Figure 1: Perspective of a GiST enhanced ORDBMS

3 Multimedia Indexing Framework (MIF) - Architecture

The system (Figure 2), described in this paper, is windows based. The GistService component as well as the GistWrapper component uses special Windows based implementations and protocols. The main architecture is divided into three modules or parts. Each module, especially the GistService and the Oracle enhancement may be used separately and may be distributed over the network.

3.1 GistService

The GistService (figure 3) is the main part in the external address space and is implemented in C++. It runs as an own process (called service) in the Windows Operating System environment and manages an environment for Generalized Search Trees (GiST). The service is split up in two main components: The *GistCommunicator* and the *GistHolder*. The *GistCommunicator* is a COM-object (Component Object Model) and offers the following IDL (Interface Definition Language) interface.

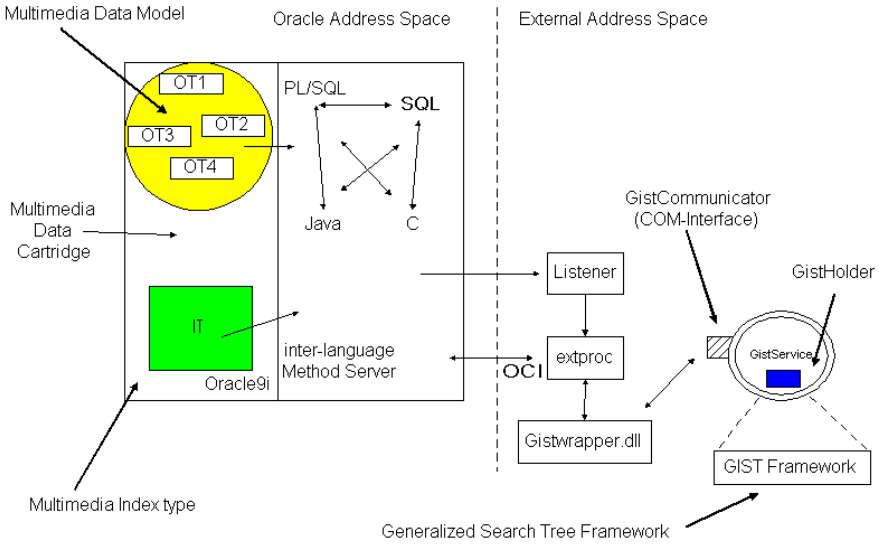

Figure 2: Overview of GistService Architecture

```
interface IMyService : IDispatch
  {
      struct queryTypeBSTR {
          BSTR name;
          int dimension;
          VARIANT coords;
      }
      struct queryOp {
          int operation;
          int argument;
          int number;
          int tree\_no;
      }

      [id(2), helpstring("Methode gistCreate")] HRESULT gistCreate([in] BSTR file_name,
                                                                   [in] int ext,
                                                                   [out] int *number);

      [id(3), helpstring("Methode gistClose")] HRESULT gistClose([in] int number);

      [id(4), helpstring("Methode gistInsert")] HRESULT gistInsert([in] BSTR key,
                                                                   [in] BSTR data,
                                                                   [in] int tree_no);

      [id(5), helpstring("Methode gistRemove")] HRESULT gistRemove([in] BSTR query,
                                                                   [in] int tree_no);

      [id(6), helpstring("Methode gistFlush")] HRESULT gistFlush([in] int tree_no);

      [id(7), helpstring("Methode gistCheck")] HRESULT gistCheck([in] int tree_no);

      [id(8), helpstring("Methode gistIsEmpty")] HRESULT gistIsEmpty([in] int tree_no,
                                                                     [out] int *empty);

      [id(9), helpstring("Methode gistOpen")] HRESULT gistOpen([in] BSTR file_name,
                                                               [out] int *tree_no);

      [id(10), helpstring("Methode gistQuery")] HRESULT gistQuery([in] struct queryOp query,
                                                                  [in] struct queryTypeBSTR type,
                                                                  [out] VARIANT* result);
```

```
}
```

All defined structures are used for our query representation, whereas the *queryTypeBSTR* structure defines first the query name and tree type (e.g., *rt_point* stands for a R-tree that stores point data), second the dimension of the stored data and finally the search values. The *queryOp* structure specifies in its arguments (1) what kind of *operation* (e.g., equals, nearest-neighbor, overlap), (2) what kind of *argument* is represented through the search values (e.g., point, rectangle), (3) how many results are desired (*number*) (4) and which tree is addressed. For an advanced description please refer to the GistService documentation.

The *GistCommunicator* is used for inter-process communication between the database (the

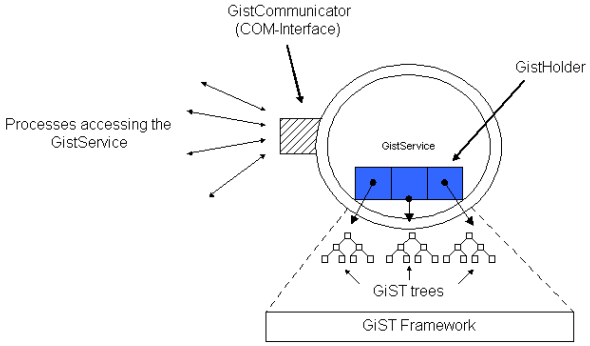

Figure 3: GistService process

GistWrapper shared library) and our GiST implemented access methods. It offers, defined in the IDL, the necessary functionality (e.g., creating, inserting, deleting), that common access methods provide. The currently available architecture eliminates the disadvantage of the actual GiST framework, namely the possibility to administrate more than one index tree at time. This is done through the *GistHolder*. It manages all currently running index trees and their accesses. Each index tree is identified through a global unique ID which is forwarded to the accessing process. Each access is defined through the appropriate ID. For simplicity, the index trees are internally stored in an array, but this data structure can be replaced by any other structure.

3.2 GistWrapper

The GistWrapper module is a C++ implemented shared library that is used by the database for connecting to the GistService. The library has two main tasks. The first one makes the GistService accessible for database procedures. An Oracle database has the possibility to call external C/C++ code via shared libraries. The GistWrapper acts as a wrapper for the GistService. The second task is the transformation of the input and output data to make it useable for either the GistService or the database. For instance, a simple C Char type has to be transformed into a BSTR string, or a VARIANT type into a String and so on. The GistWrapper module offers the following interface:

```
extern "C" int __declspec(dllexport) gistlibInit();
extern "C" int __declspec(dllexport) gistlibCreate(char *,int);
extern "C" int __declspec(dllexport) gistlibInsert(char *, char *, int);
```

```
extern "C" int __declspec(dllexport) gistlibClose(int);
extern "C" int __declspec(dllexport) gistlibRemove(char *, int);
extern "C" int __declspec(dllexport) gistlibCheck(int);
extern "C" int __declspec(dllexport) gistlibFlush(int);
extern "C" int __declspec(dllexport) gistlibOpen(char *);
extern "C" bool __declspec(dllexport) gistlibIsEmpty(int);
extern "C" int __declspec(dllexport) gistlibQuery(char *, int, char *, int, int, int, int, char*);
```

The meaning of the different input and output parameters is simply the same as in the GistService interface (see the documentation for a precise explanation of all methods and their parameters).

3.3 Multimedia Data Cartridge (MDC)

3.3.1 Oracle Data Cartridges - What they are!

Oracle offers with its *Data Cartridge* technology a mechanism and possibility for extending the capabilities of an Oracle database. Oracle databases are build as a modular architecture with extensible services. These extensible services enable database designer and programmer to extend e.g., the type system, the query processing or the data indexing (see figure 4). Every extensible service offers an extensibility interface which can be used to enhance and modify the database for the users needs. The main characteristics of data cartridges are the following:

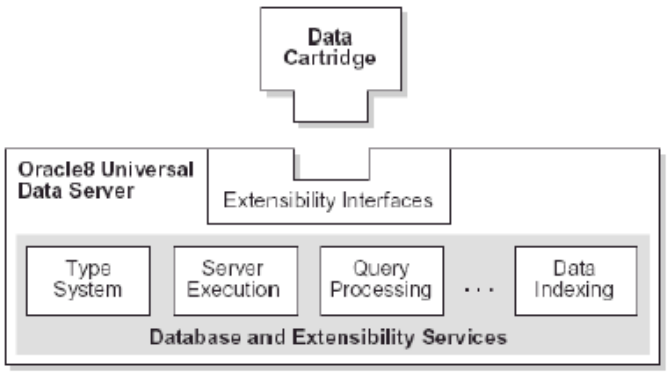

Figure 4: Oracle Data Cartridge

- **Data Cartridges are server-based.**
  Their components reside at the server and are primary accessed from the server.

- **Data Cartridges extend the server.**
  With the possibility of defining new types, one is able to create a solution-oriented image of a real world problem. For example, one can create a table *Company* with a column *Person*. Each *Person* object contains additional information in the form of a photo of type *Image*. All new types can be enhanced by behavior e.g., for transforming the image, for calculation of the persons salary, and so on.

- **Data Cartridges are integrated with the server.**
  The extensions made to the server are integrated within the server engine, so that the optimizer, query parser, indexer and other server mechanisms recognize and respond to these extensions. By implementing all necessary interface methods of the extensibility interface, one can be sure that e.g., the new domain-specific index is used for better performance and for dealing with new types.

- **Data Cartridges are packed.**
  A Data Cartridge can be seen as a unit. All extensions can be packed to a unit and easily installed on every Oracle database. In this scenario, no additional implementation has to be done and every Oracle database can be equipped with the new functionality without further maintenance.

3.3.2 Oracle's Extensibility Services

Since Oracle8i, the server provides services for basic storage, query processing, optimization and indexing. These services have been made *extensible* for application developer to trim the database to their special needs. The usual way for using these services is to implement a Data Cartridge that extends the extensibility interface. This section describes the most basic and common used extensible services. It is not necessary for a Data Cartridge developer to implement all methods in the extensible interfaces.

- **Extensible Type System**
  Beside the common native SQL data types, such as `INTEGER, NUMBER, DATE and VARCHAR`, Oracle adds support for new types including user-defined objects, collections (`VARRAY`, nested tables), internal large object types (`BLOB, CLOB`) or `BFILE`.
  The extensible type system is in most cases the main feature for Data Cartridge developments. Here, one may define new domain-specific object types. These types specifies the underlying persistent data (which is represented through their *attributes*) and are enhanced by their relevant behavior (*methods*). Object types are used to extend the modelling capabilities in common with native data types. Every object type can have one or more *attributes*. These *attributes* consists of a name and a type. This type can again consist of domain-specific object types or native types. The *method* is a procedure or a function which has directly access to the object's *attributes*. The reader is referred to [9] for further information on other data types.

- **Extensible Server Execution Environment**
  Oracle's server execution environment provides two main advantages. First, the components of a data cartridge and other database procedures and functions can be implemented in any popular programming language such as PL/SQL, Java or external C language routines. Where every combination of the set of programming languages is possible. Thus, one object method can be implemented in Java another in PL/SQL and the Java method in turn can call, via a PL/SQL function, an external C routine. The use of external C routines is useful for computation-intensive operations. These routines are executed in a separate address space than the server. This fact ensures that the database server is insulated under all circumstances from program failures. Nevertheless external routines are able to *call back* to the Oracle Server using OCI (Oracle Call Interface).

Second, the type system decouples the implementation of an object's method from its specification. The specification just defines the head of the method with appropriate in and out parameters. It is not defined what kind of implementation is behind the object's method. Therefore, a data cartridge developer has the possibility for, a simple exchange of different implementations, using different programming languages for the same method specification.

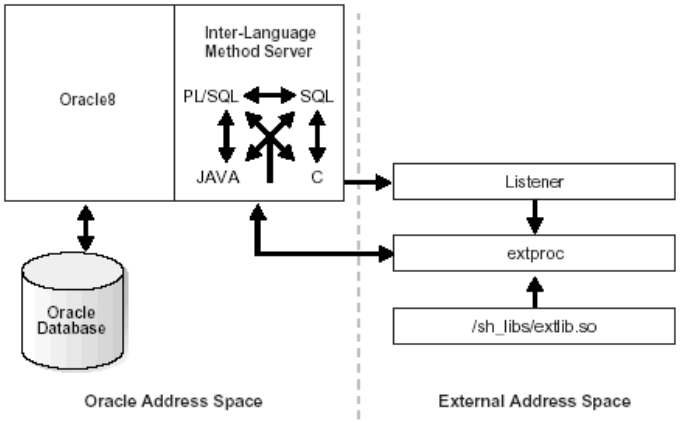

Figure 5: Oracle Server Execution Environment

- **Extensible Indexing**
  Typical DBMS, including Oracle, supports just a few types of access methods (B+Trees, Hash Indexes, Text Indexes) on a limited number of data types (number, strings). However, commercial DBMS have to store and handle multimedia data such as images, videos or other user-defined objects. In addition, one needs efficient query and retrieval systems for that kind of data. Indexing plays a major role in this context. But database companies cannot provide access methods for all user needs. Therefore, Oracle builds an extensible server which allows developer defining new index types as required. The data cartridge is responsible for defining the index structure, maintaining the index content during load and update operations and searching the index during query processing. The index itself can be stored internally as heap-organized or index-organized table or externally as an operating system file.
  Oracle introduces the concept of an *indextype* for user-defined access methods. Each *indextype* has specific operators and uses an object that is responsible for the implementation. The object has to implement all necessary functions which are provided by the ODCI (Oracle Data Cartridge Interface) for indexing e.g. ODCIIndexCreate, ODCIIndexInsert or ODCIIndexDrop.

```
CREATE OR REPLACE INDEXTYPE <indexname> FOR
  <operator1>,
  <operator2>,
  ...
USING <object>;
```

  Typical databases do not support extensible indexing. Therefore, application developer has to maintain their own file-based indexes for complex data. It needs, of course, a great effort to retain consistency between external indexes and the related data in the database. This additional effort and complexity is not needed in the Oracle extensible indexing concept.

- **Extensible Optimizer**
  With the extensible optimizer functionality, one is able to create statistic collections, selectivity and cost functions for the user-defined indexes, functions and operators. This information is used by the optimizer in choosing the query plan. An application developer has only the possibility to enhance the cost-based optimizer. There is no way in adding functionality to the rule-based optimizer.

3.3.3 Multimedia Data Cartridge - Indexing

This section introduces to the indexing part of our Multimedia Data Cartridge. We present our methodology step-by-step. The Multimedia Data Cartridge contains three main parts (GistService (section 3.1), GistWrapper (section 3.2) and Oracle Enhancement (Figure 2)).
The Oracle enhancement is defined through following steps: First one has to enable the database for using our external C routines that are stored in the *GistWrapper* shared library. This is done by defining the location of the shared library and by creating respective PL/SQL functions that correspond to the external C routines. The location is indicated with the following statement:

```
 CREATE LIBRARY libgist_lib AS
'e:\temp\gist_service\gistwrapp.dll';
```

Examples for corresponding functions are the following, whereas the first is necessary for creating a new index and the second one is used for inserting in the index tree:

```
 CREATE OR REPLACE FUNCTION PLS_libCreate(f IN
VARCHAR2, ext IN BINARY_INTEGER) RETURN BINARY_INTEGER AS
   EXTERNAL LIBRARY libgist_lib
   NAME "gistlibCreate"  -- Name of function call. Quotes preserve lower case.
   LANGUAGE C
   PARAMETERS(f string, ext int, return int);
/

CREATE OR REPLACE FUNCTION PLS_libInsert(key IN CLOB, data IN
VARCHAR2, tree_no IN BINARY_INTEGER) RETURN BINARY_INTEGER AS
   EXTERNAL LIBRARY libgist_lib
   NAME "gistlibInsert"  -- Name of function call. Quotes preserve lower case.
   LANGUAGE C
   PARAMETERS(key string, data string, tree_no int, return int);
/
```

The next step is to define functions for every index operator. An index operator in a B-tree, for example, is the equality, lower or larger operation. In our case, we defined all necessary functions for the R-tree family. These two operator functions define an equality search for d-dimensional points and a nearest-neighbor search for d-dimensional points. They are only used iff the index type is defined, but no index is created on a specific table. If an index exists on a queried table, then the required methods of the use-defined object are executed.

```
CREATE OR REPLACE FUNCTION rt_equal_point_func(a CLOB, b CLOB, count number, dimension number) RETURN NUMBER AS
 BEGIN
  if a = b then
    return 1;
  else
    return 0;
  end if;
END; /

CREATE OR REPLACE FUNCTION rt_nearest_point_func(a CLOB, b CLOB,
count number, dimension number) RETURN NUMBER AS BEGIN
   ....
    return 1;
END; /
```

Every operator function has a corresponding operator definition. For instance:

```
CREATE OR REPLACE OPERATOR rt_equal_point BINDING (CLOB, CLOB,
number, number) RETURN NUMBER USING rt_equal_point_func;

CREATE OR REPLACE OPERATOR rt_nearest_point BINDING (CLOB, CLOB,
number, number) RETURN NUMBER USING rt_nearest_point_func;
```

These two operator definitions use the operator functions above.
The next step is to create an user-defined object that implements all necessary index methods supported by ODCI. In our case, the object is termed *GistRTIndex* and offers implementations for the following methods:

- ODCIGetInterfaces
- ODCIIndexCreate
- ODCIIndexDrop
- ODCIIndexInsert
- ODCIIndexDelete
- ODCIIndexUpdate
- ODCIIndexStart
- ODCIIndexFetch
- ODCIIndexClose

As mentioned in subsubsection 3.3.2, there is a decoupling between the method specification and implementation. The code fragment below shows the specification of all methods pointed out above. The decoupling can be seen for example in the *ODCIIndexCreate* method, where the implementation is realized in a java class called *GistRTIndex* or in the *ODCIIndexDrop* method which is implemented in PL/SQL. The PL/SQL implementations are collected in the so called *BODY* (not shown in this paper) of an object. The java class in turn calls via PL/SQL functions some external C routines for accessing the *GistService* that handles all index trees.

```
CREATE OR REPLACE TYPE GistRTIndex AS OBJECT (
   scanctx integer,
  STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList) RETURN NUMBER,
  STATIC FUNCTION ODCIIndexCreate (ia sys.odciindexinfo, tr_tp_dim varchar2)
                                    RETURN NUMBER AS LANGUAGE JAVA NAME
 'GistRTIndex.ODCICreate(oracle.ODCI.ODCIIndexInfo,java.lang.String) return java.math.BigDecimal',
  static function ODCIIndexDrop(ia sys.odciindexinfo) RETURN NUMBER,

  STATIC FUNCTION ODCIIndexInsert(ia sys.odciindexinfo, rid VARCHAR2,
                                                  newval CLOB)
                                  RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIInsert(oracle.ODCI.ODCIIndexInfo, java.lang.String,
        java.lang.String) return java.math.BigDecimal',

  STATIC FUNCTION ODCIIndexDelete(ia sys.odciindexinfo, rid VARCHAR2,
                                                  oldval CLOB)
                                  RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIDelete(oracle.ODCI.ODCIIndexInfo, java.lang.String,
        java.lang.String) return java.math.BigDecimal',

  STATIC FUNCTION ODCIIndexUpdate(ia sys.odciindexinfo, rid VARCHAR2,
                                  oldval CLOB, newval CLOB)
                                  RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIUpdate(oracle.ODCI.ODCIIndexInfo, java.lang.String,
        java.lang.String, java.lang.String) return
```

```
        java.math.BigDecimal',

  STATIC FUNCTION ODCIIndexStart(sctx in out GistRTIndex, ia sys.odciindexinfo,
                        op sys.ODCIPredInfo,
                        qi sys.ODCIQueryInfo,
                        strt number,
                        stop number,
                        cmpval CLOB, count number, dimension number)
                        RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIStart(GistRTIndex [], oracle.ODCI.ODCIIndexInfo,
        oracle.ODCI.ODCIPredInfo,
        oracle.ODCI.ODCIQueryInfo, java.math.BigDecimal,
        java.math.BigDecimal,
                java.lang.String, java.math.BigDecimal, java.math.BigDecimal) return java.math.BigDecimal',


  MEMBER FUNCTION ODCIIndexFetch(nrows number, rids OUT sys.odciridlist)
   RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIFetch(java.math.BigDecimal,
    oracle.ODCI.ODCIRidList[]) return java.math.BigDecimal',

  MEMBER FUNCTION ODCIIndexClose RETURN NUMBER AS LANGUAGE JAVA NAME
    'GistRTIndex.ODCIClose() return java.math.BigDecimal'
); /
```

The last step for creating an own index is to define an *indextype* as shown in subsubsection 3.3.2. We have to specify each index type separately. The R-tree index type, for example, requires an corresponding *name* (e.g. rtree), available *operators* (e.g. rt_equal_point) and the used *object type* (e.g GistRTCLOBIndex). This can be interpreted with the following command.

```
CREATE OR REPLACE INDEXTYPE rtree FOR
    rt_equal_point(CLOB, CLOB, number, number),
    rt_equal_rect(CLOB, CLOB, number, number),
    rt_nearest_point(CLOB, CLOB, number, number),
    rt_nearest_rect(CLOB, CLOB, number, number),
    rt_overlap_rect(CLOB, CLOB, number, number),
    rt_contains_rect(CLOB, CLOB, number, number),
    rt_contained_rect(CLOB, CLOB, number, number)
USING GistRTCLOBIndex;
```

Finally one is able to create an index of the new index type like:

```
CREATE INDEX it1 ON lob(f2) INDEXTYPE IS rtree
PARAMETERS('rt_point');
```

Here we created an R-tree index called *it1* on a table *lob* for a column of type `CLOB` (*f2*). Every operation (select, insert, update, ...) on table *lob* will initiate the corresponding method of our index type implementation.

4 Installation

This section describes the installation process of the indexing part of our Multimedia Data Cartridge. The process contains two parts. The first one concerns all necessary steps on the operating system, the second one the database enhancements.

- **Operating system**
  As mentioned earlier, our implementation is restricted to the windows operating system. An extension to linux or unix is with additional costs possible. First, we have to store

our shared library (*gistwrapp.dll*) and the windows service (*GistService.exe*) in a specified folder. There exists no installer for our service at the moment, therefore one has to adapt all path entries in our two installation files manually. The first file which has to be manipulated is called *GistService.reg*. This file directly adds all necessary entries to the windows registry. These entries are for example the *uuid* that represents our *GistService* process, some path entries where the service can be found and so on. By double-clicking, all entries will be added automatically to the system's registry. Further, one has to use two Microsoft tools for installing new services (*instsrv.exe and srvany.exe*).

- **Database System**
  The installation of the indexing part of our Multimedia Data Cartridge is semi-automatic. There are two files which one needs to adapt. The first one is called *security.sql* and contains some file permission rules. Necessary modifications in *security.sql* are the database user where the data cartridge should be installed and the path entries where our debugging information should be stored. The second file is called *gistwrapper.sql* and contains all information and commands necessary for creating the index types, their components and functions for accessing the *GistService* via the shared library *gistwrapp.dll*. In the *gistwrapper.sql* file, one has to modify the database user and the correct path of the *gistwrapp.dll*. The next step is to load the necessary java classes into the database. This is realized with the *lp.bat* batch file. Before, one needs to load the ODCI.jar and CartridgeServices.jar packages.

5 Measurements

This section describes a significant part of the series of experiments we performed in order to evaluate the effectiveness of our *Multimedia Indexing Framework* (MIF). The tests were carried out on two distinct datasets, one *synthetic (uniform dataset)* and one *real*. The experimental settings are as follows:

- The synthetic dataset contains 64 and 96 dimensional feature vectors that are represented as strings. The values were generated uniformly over the normalized [0..1] space.

- The real dataset was generated from an 1 hour and 43 minutes long movie, encoded with DIV3. The movie contained 141.000 frames of size 576*372 pixel. From the movie, we extracted by 64 dimensional color histogram, by retaining the two most significant bits in the RGB space. The generated feature vectors were inserted into the database (twice to increase the database size).

In order to compare the built-in indexing mechanism and our MIF for efficiency we have to carry out exact match queries. The remaining supporting MIF retrieval functions, like range search, NN-search and overlap search, are in addition to the build-in index functions.

The insertion and retrieval was carried out through server-sided JDBC, i.e., the java class resides in the database and is executed through Oracle's own JVM (Java Virtual Machine). We always used the same java classes for the measurements, thus the results are comparable.

5.1 Indexing Details

The feature vectors are stored in a column of type *Character Large Object (CLOB)*. We had to use the *CLOB* data type, because of the high dimensionality of the required data (e.g., multi-level feature vector of frames in MPEG movies). As a consequence, the build-in B-tree can not be used, as it does not handle CLOBs. In theory, the build-in B-tree is able to index data up to 160 dimension (limited through the VARCHAR2 data type of Oracle), but in practice the dimension handled depends on the number of data points, for instance for a dimension of 2 (!) we are able to insert 2.8 millions, after this value the database crashed. Instead, we used the *Oracle Text Index* which is able to index CLOB string representations without severe limitations. Based on these index techniques we compared the response time between a normal (no index) solution, the Oracle Text Index and *MIF*. The response time was measured for insertion and query operations. The query operation was limited to exact match queries, because of the limited functionality of current database indexing mechanisms. As mentioned above this shortcoming can be compensated with MIF.
The different access methods were created with the following statements. The first statement creates an R-tree index based on our *GistService* implementation. The second one creates a text index that uses Oracle Text Index capabilities.

```
CREATE INDEX lob ON lob(f2) INDEXTYPE IS rtree
PARAMETERS('rt_point');


CREATE INDEX lob ON lob(f2) INDEXTYPE IS ctxsys.context;
```

Based on these index techniques we produced a measurement equation between a normal no index solution, the Oracle Text Index and our *GistService* index. The select statement is the following:

```
select * from lob where
rt_equal_point(f2,'0.539777921054799 0.6924431242553102
0.9597020855625487 0.09410592024055775 0.5315271296051458
0.3983927095907065 0.9210868758838193 0.5679278446751121
0.07328062897625498 0.14025569697362295 0.9203892718791543
0.1464733769423443 0.40359949162271747 0.3226128494765611
0.01516395855245145 0.9099703305283804 0.7292131517427908
0.8057646999980167 0.9026170611665935 0.9904572202943244
0.3225747348638268 0.5784491526885225 0.8995128345783748
0.7109777570571694 0.6627525418329804 0.920771523252903
0.33823456214052583 0.7749395930114605 0.2639748023486367
0.16382280205410582 0.13708625319989498 0.807913929245711
0.7589974052014429 0.5518762403021286 0.7413992338741358
0.09419157512846821 0.008564902549407338 0.6437158252065578
0.8517338479628374 0.92170024252968 0.49358747362533706
0.15156507487838478 0.35644301168229675 0.5514269974827135
0.013257470813778038 0.10924795716082669 0.6680911036909487
0.46127841604503406 0.7601811018290506 0.7856282112518719
0.22669817356001576 0.17935082831586557 0.17270222075863306
0.8150372133196819 0.2775059111563083 0.7178711194860397
0.5749468381639644 0.6033146713754611 0.15693868337613637
0.3466085849166787 0.5067183692188529 0.18625047302678122
0.22765893180111296 0.20334661208137417',3,64) = 1
```

The *rt_equal_point* operation is used for an equality search. If no index exists, the operation is executed with a PL/SQL procedure that sequentially compares all column entries with the given one. During a *GistService* operation, this function call is transferred to the correct *GistService* index type and the corresponding java object. The first parameter is the column name, the second is the search string, the third is the amount of results and the last one is the degree of dimensionality.
A possible select statement for an Oracle Text Index looks like follows.

```
 SELECT * FROM lob WHERE
CONTAINS(f2,'0.539777921054799 0.6924431242553102
0.9597020855625487 0.09410592024055775 0.5315271296051458
0.3983927095907065 0.9210868758838193 0.5679278446751121
0.07328062897625498 0.14025569697362295 0.9203892718791543
...',1)>0
```

5.2 Detailed Results

The comparison of MIF using R-trees and Oracle 9i build-in Text Index [19] shows that MIF trees show less insertion efficiency, but offer significantly higher query performance.

Indexing – Synthetic Dataset The following figures show the results for the insertion process: Figure 6 for 64 dimensional point entries and Figure 7 for 96 dimensional point entries. These figures depict that the MIF introduces higher insertion times than the related solutions. However, with a higher dimension, i.e., 96 dimensions, the difference to the Oracle Text Index is lower than for a smaller dimension which shows advantages for the MIF for higher dimensions. The extra time for the MIF is caused by the overhead from switching between the Oracle address space and the external address space.

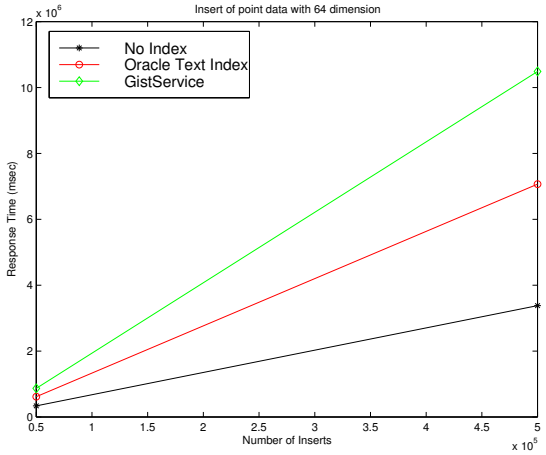

Figure 6: Response Time of the Insert Statements (50000 to 200000 points with 64 dimension)

It has to be noted that the memory consumption of the *Oracle Text Index* is enormous. The table space consumed is in the worst case over 3.5 GB for an insertion operation of 200.000 point elements with 96 dimensions. Compared to this, the memory consumption of the *MIF* is significantly smaller, e.g., for the same insertion operation as above, it requires only a table space of about 400 MB.

Retrieval Query – Synthetic Dataset The results for the query evaluation show that our framework MIF beats clearly the related solutions. This is important, as the query process is far more often used than the indexing process in typical ad-hoc scenarios.

Figure 8 shows that for exact match queries involving 64 dimensions, the *MIF* environment outperforms clearly both related solutions. Every bar in our figure represents the accumulated response time of 10 randomized select statements. The representation is logarithmic, because

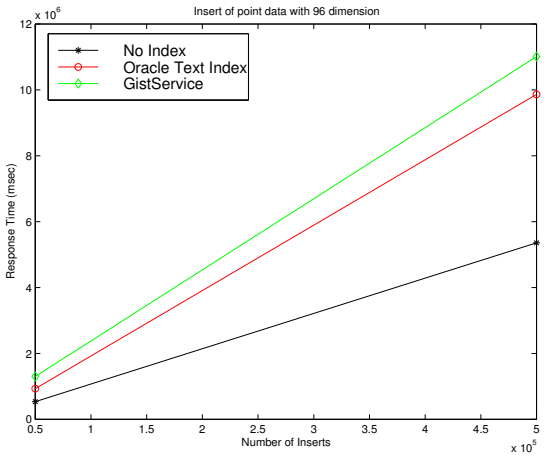

Figure 7: Response Time of Insert Statements (50000 to 200000 point entries with 96 dimension)

of the large gap between the no index solution and the two others. A sequential search is, of course, significantly slower. Positively, the *MIF* environment is from 7 to 21 times faster than the Oracle Text Index for 200.000 data entries of 65 respectively 96 dimensions. Furthermore, MIF performs from 883 to 2405 times better than the no index solution for the same amount of point entries. The difference in performance between the indexes are independent of the amount of data entries (please compare the left and right sides of figure 8 and 9.

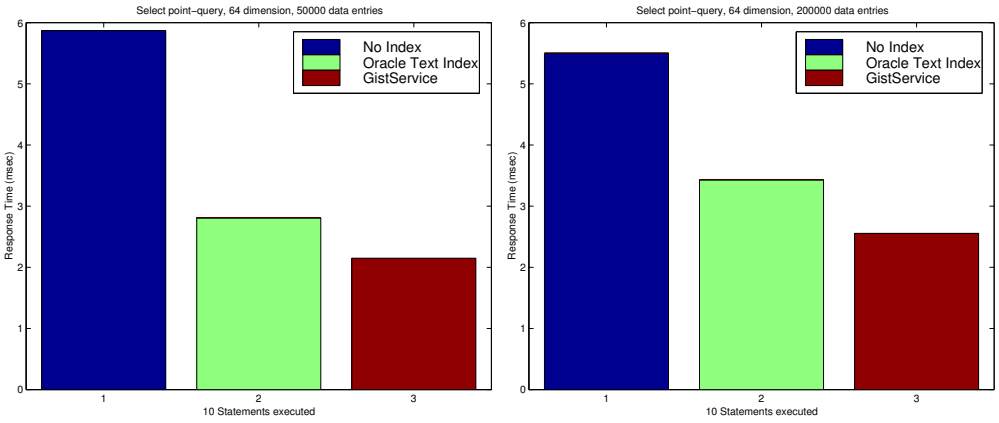

Figure 8: Response Time of Select Statements with 50000 (left) and 200000 points (right) and 64 dimension

Indexing and Retrieval – Real Dataset The response time for the insertion process with the real dataset was similar to the results obtained from the synthetic dataset test series and is not presented in this paper. The response time is again measured as sum of 10 successive executed randomized select queries. In reason of the great discrepancy between the index and non-index solutions, the results are again in an logarithmic representations.

Figure 10 shows that the Oracle Text Index performs better than in the previous test series, however, on average we are still a factor of 2 faster than the Oracle Index (and this even with the call to an external address space). An explanation for this behavior derives from the differences in

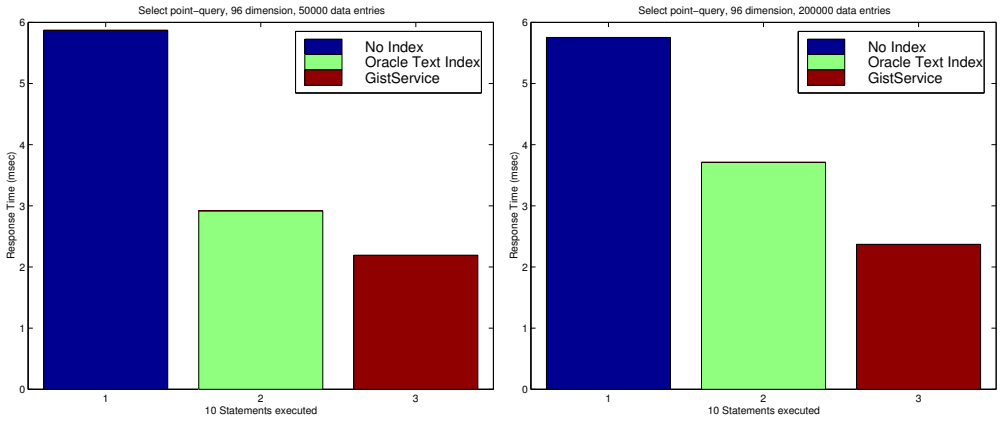

Figure 9: Response Time of Select Statements with 50000 (left) and 200000 points (right) and 96 dimension

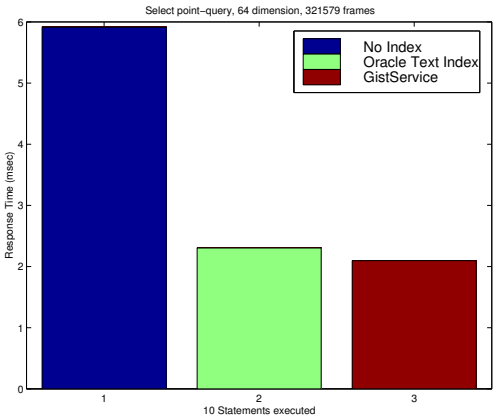

Figure 10: Response Time of Select Statements for color histograms (real dataset) with 64 dimension

the data entries. The data entries of the first test series consists of randomized values generated uniformly over the normalized [0..1] space. Contrarily, the color histogram derived from this special movie contains more zero values than in the uniform case. This fact enables the Oracle Text Index to apply compression more efficiently than in the previous testing series. Thus, we put ourself into a highly disadvantageous case, however, we still outperform the Oracle Text Index. It is important to note again, that we mainly extend the functionality of the index services to multimedia specific operations and that the purpose of this test series is to show that our implementation processes efficiently.

Figure 11 shows the capacity properties of current available GiST implemented trees for High-dimensional point entries. The maximum dimensionality is internally limited to 510 (equivalent to 4 KB). The page size is per default 8 KB, so one page is able to take 2 entries with 510 dimensions. The split function shows for high dimensionality (d>230) in most access methods some malfunctions. With the exception of the SS-tree and the R*-tree for rectangle data, index trees are only able to insert data entries for the root page. Whereas for the SP-tree the split method is not implemented.

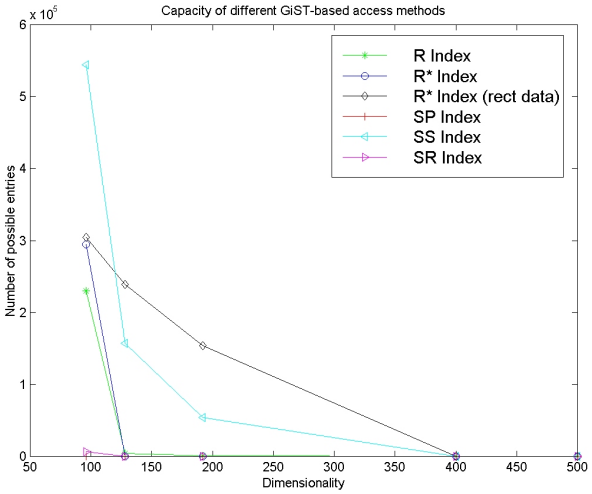

Figure 11: Capacity properties of current available GiST implemented trees.

6 Conclusion

This technical report has introduced an integration of GiST-implemented access methods into Oracle databases with the help of Data Cartridges. We have shown that a new index type can be instantiated with few steps using the Data Cartridge technology in combination with the GiST-framework. Furthermore, our measurements have demonstrated that such an enhancement improves performance of queries for user-defined data types. We queried in our testing environment a table of color histograms with different dimensionality (64,96) stored as CLOBs. The results were compared between our GistService, the Oracle Text Index and a sequential search. The newly created framework builds the frame for efficient querying in the CODAC project.
Future research will focus on integrating additional access methods such as LPC-file [27] for better NN-search support in high dimensional vector spaces or *DependencyStructure* DS of the MPEG-7 standard [26] for indexing text annotations.

References

[1] Nathan G. Colossi and Mario A. Nascimento. Benchmarking Access Structures for High-Dimensional Multimedia Data. Technical Report 99-05. Department of Computing Science University of Alberta, Canada. 1999.

[2] Joseph M. Hellerstein, Jeffrey F. Naughton and Avi Pfeffer. Generalized Search Trees for Database Systems. In Proceedings of the 21st International Conference of Very Large Databases VLDB, pages 562-573, Zurich Switzerland 1995.

[3] Marcel Kornacker. PHD Thesis: Access Methods for Next-Generation Database Systems. University of California at Berkley, 2000.

[4] M. Kornacker, M. Shah, J. M. Hellerstein. Amdb: A Design Tool for Access Methods. Technical Report UCB//CSD-99-1051, University of California at Berkeley, 1999.

[5] M. Shah, M. Kornacker, J. M. Hellerstein. Amdb: A Visual Access Method Development Tool. In Proc. User Interfaces To Data Intensive Systems, pages 130-140, Edinburgh, Scotland, 1999.

[6] C. Carson, M. Thomas, S. Belongie, J. M. Hellerstein and J. Malik. Blobworld: A System for Region-Based Image Indexing and Retrieval. Third International Conference on Visual Information Systems, pages 509-517, Amsterdam, The Netherlands, June 1999. Springer Verlag ISBN:3-540-66079-8.

[7] C. Carson, S. Belongie, H. Greenspan and J. Malik. Blobworld: Image Segmentation using Expectation-Maximization and its Application to Image Querying. IEEE Transactions on Pattern Analysis and Machine Intelligence. SUB.

[8] M. Thomas, C. Carson and J. M. Hellerstein. Creating a Customized Access Method for Blobworld, Proc. of the 16th Int. IEEE Conf. on Data Engineering, page 82, San Diego, CA, March 2000, IEEE Computer Society 2000.

[9] Data Cartridge Developer‘s Guide. Oracle 8i, Part No.: A76937-01, December 1999. `http://technet.oracle.com/doc/oracle8i_816/appdev.816/a76937/title.htm`.

[10] D. A. White and R. Jain. Similarity Indexing with the SS-tree. In Proc. of the 12th Int. IEEE Conf. on Data Engineering, pages 516-523, New Orleans, Louisiana 1996. IEEE Computer Society 1996, ISBN 0-8186-7240-4.

[11] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In Proc. of the 1997 ACM SIGMOND Int. Conf. on Management of Data, pages 369-380, 1997.

[12] P. Ciaccia, M. Patella and P. Zezula. M-tree: An efficient Access Method for Similarity Search in Metric Spaces. In Proc. of the 23rd Int. Conf. on Very Large Data Bases, pages 426-435, Athens, Greece 1997. Morgan Kaufmann 1997, ISBN 1-55860-470-7.

[13] S. Berchtold, D. A. Keim and H. P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. Proceedings of the 22nd VLDB, pages 28-39, Mumbai (Bombay), India August 1996. Morgan Kaufmann 1996, ISBN 1-55860-382-4.

[14] K. I. Lin, H. V. Jagadish and C. Faloutsos. The TV-tree: An Index Structure for High-Dimensional Data. VLDB Journal, 3, 1994.

[15] Volker Graede and Oliver Günther. Multidimensional Access Methods. ACM Computing Surveys, 30(2), 1998.

[16] C. Böhm, S. Berchtold and D. A. Keim. Searching in High-Dimensional Spaces - Index Structures for Improving the Performance of Multimedia Databases. In ACM Computing Surveys, 2001.

[17] R. Bliujute, C. S. Jensen, S. Saltenis, G. Slivinskas. R-tree Based Indexing of Now-Relative Bitemporal Data. In Proc. of the 24th VLDB Conference, pages 345-356, New York, USA, 1998. Morgan Kaufmann 1998, ISBN 1-55860-566-5.

[18] M. Kornacker. High-Performance Extensible Indexing. In Proc. of the 25th VLDB Conference, pages 699-708 Edinburgh, Scotland, 1999. Morgan Kaufmann 1999, ISBN 1-55860-615-7.

[19] Oracle Text. An Oracle Technical White Paper, May 2001. `http://technet.oracle.com/products/text/`.

[20] J. M. Hellerstein, E. Koutsoupias and C. H. Papadimitriou. On the Analysis of Indexing Schemes. In Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 249-256, Tucson, May 1997.

[21] Guojun Lu. Multimedia Database Management System. Artech House, Boston - London, 1999, ISBN 0-89006-342-7.

[22] Marius Tico, Taneli Haverinen and Pauli Kuosmanen. A Method of Color Histogram Creation for Image Retrieval. In IEEE Proc. of 4th Nordic Signal Processing Symposium (NORSIG), Norrkobing, Sweden, 2000.

[23] G. Cha, X. Zhu, D. Petkovic and C. Chung. An Efficient Indexing Method for Nearest Neighbor Searches in High-Dimensional Image Databases. IEEE Transaction on Multimedia, Vol. 4, No. 1, pages 76-87, March 2002.

[24] R. Leornardi and P. Migliorati. Semantic Indexing of Multimedia Documents. IEEE Multimedia, Vol. 9, No. 2, pages 44-51, April-June 2002.

[25] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley Publishing Company, 1989, ISBN 0-201-50255-0.

[26] J.M. Martinez. Overview of the MPEG-7 standard, v5.0. ISO/MPEG N4509, MPEG Requirements Group, Dezember 2001. Available at `http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm`.

[27] Marius Tico, Taneli Haverinen and Pauli Kuosmanen. A Method of Color Histogram Creation for Image Retrieval. In IEEE Proc. of 4th Nordic Signal Processing Symposium (NORSIG), Norrkobing, Sweden, 2000.

Institute of Information Technology
University Klagenfurt
Universitaetsstr. 35
A-9020 Klagenfurt
Austria

http://www.itec.uni-klu.ac.at

University Klagenfurt