Extensibility in ORDBMS Databases: An Exploration of the Data Cartridge Mechanism in Oracle9i

A thesis submitted in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

By

Tulimevava Kaunapawa Ndakunda

Computer Science Department
Rhodes University
Grahamstown
South Africa

April 2004

Abstract

To support current and emerging database applications, Object-Relational Database Management Systems (ORDBMS) provide mechanisms to extend the data storage capabilities and the functionality of the database with application-specific types and methods. Using these mechanisms, the database may contain user-defined data types, large objects (LOBs), external procedures, extensible indexing, query optimisation techniques and other features that are treated in the same way as built-in database features. The many extensibility options provided by the ORDBMS, however, raise several implementation challenges that are not always obvious. This thesis examines a few of the key challenges that arise when extending Oracle database with new functionality.

To realise the potential of extensibility in Oracle, the thesis used the problem area of image retrieval as the main test domain. Current research efforts in image retrieval are lagging behind the required retrieval, but are continuously improving. As better retrieval techniques become available, it is important that they are integrated into the available database systems to facilitate improved retrieval. The thesis also reports on the practical experiences gained from integrating an extensible indexing scenario. Sample scenarios are integrated in Oracle9i database using the data cartridge mechanism, which allows Oracle database functionality to be extended with new functional components.

The integration demonstrates how additional functionality may be effectively applied to both general and specialised domains in the database. It also reveals alternative design options that allow data cartridge developers, most of who are not database server experts, to extend the database. The thesis is concluded with some of the key observations and options that designers must consider when extending the database with new functionality.

The main challenges for developers are the learning curve required to understand the data cartridge framework and the ability to adapt already developed code within the constraints of the data cartridge using the provided extensibility APIs. Maximum reusability relies on making good choices for the basic functions, out of which specialised functions can be built.

Table of Contents

List of Figures

List of Tables

Acknowledgements

'In all your ways, acknowledge Him. He will make your paths straight'

I am highly indebted to the following people who played a significant role in this research:

1. Prof. Dave Sewry and Dr. Alfredo Terzoli for their excellent supervision, faith in my intellectual abilities and their absolute support throughout this research.

2. My fiancé Toivo for moral support and for proof reading several chapters of my thesis.

3. My dear parents for their motivation and encouragement, and believing I can reach the stars.

4. All my younger brothers and sisters for their inspiration.

5. All my friends and members of the Computer Science department for all their support.

Chapter 1:Introduction

This chapter introduces the work done in this thesis. The background to the problem is sketched, and the application domain described. The objectives and the focus of the research are also explained, and the outline of the thesis is set with a summary of the contents of each chapter.

1.1 Background and problem statement

Traditionally, relational database management systems (RDBMS) were designed to manage small and structured data types such as strings and dates. In recent years, however, application domains that require large and complex data types have increased. A typical example of such applications is found in the Computer Science Department at Rhodes University, which has collected many multimedia data for a variety of users over the last few years. The search for mechanisms that better support the design and integration of complex data in database systems is therefore a necessity today.

So far, the RDBMS is the most widely used database system [RAG00] [SUB98], and has many attractive features, such as simplicity of the query language and advanced query processing techniques. Storage and manipulation of complex data in these databases is accomplished with opaque Binary Large Objects (BLOBs) or URLs to content stored directly in operating system files. Experience has, however, revealed the inherent weakness of this approach: there is a mismatch between the nature of data and the way users and the system interact with it. Relational operations and queries require complex objects to be mapped to simple relational tables, and standard indexing approaches cannot provide content-based queries to BLOB content.

Object-Oriented Database Management Systems (OODBMS), which support the development of applications in an object-oriented manner, were developed as an alternative to managing complex data applications [KAB97]. OODBMS add DBMS functionality to object-oriented programming languages, to be able to store and provide query capabilities for complex, structured data and large, unstructured data. Despite their powerful data abstraction and modelling capabilities, however, OODBMS have not gained commercial success in the database industry [RAG00]. Today, Object Relational Database Systems (ORDBMS) are gaining in popularity for managing multimedia and

other complex data applications. ORDBMS simply add object orientation capabilities to RDBMS to store persistent objects in relations. These databases maintain the simplicity and flexibility of the relational model while at the same time providing support for objects in the database. In this way, ORDBMS support a broader set of application requirements.

1.2 Motivation

Most RDBMS vendors have today incorporated ORDBMS functionality in their products, with state-of-the-art ORDBMS such as Oracle and Informix providing facilities to support the storage, retrieval and management of different kinds of multimedia and other complex data. Even with these facilities however, the management of multimedia data, especially multimedia retrieval, remains a challenge. Application domains that require new retrieval techniques are continuously emerging, but state-of-the-art multimedia retrieval techniques are, in general, lagging behind the required retrieval capabilities. Today, no single database system is able to encapsulate all the desired retrieval functionality, and the search for improved techniques remains. As improved retrieval techniques emerge, it is important to integrate them into existing database systems to enhance their retrieval capabilities.

This integration could be achieved with ORDBMS' support for database extensibility. ORDBMS enables user-defined data types and their associated functionality to be added to the database system, and to be used in the same way as other built-in data types and functions. Even with this possibility however, it is interesting to see that most developers continue to use the ORDBMS as a relational database. This is a clear indication that the enhanced functionality of ORDBMS raises several implementation challenges that need to be addressed. A number of benchmarks for ORDBMS have been proposed to evaluate the usefulness of object-relational features in the database. These benchmarks, such as BORD [LKK00], Bucky [CWN+97] and MORD [STA03] measure the performance of typical DBMS functions that support extensibility, but do not cater for the user-defined extensions themselves. In order to effectively extend the database, it is essential to understand the implications of using these extensions. This can be done by investigating the key challenges that arise when extending the ORDBMS with new functional capabilities, and by identifying how new application domains can be built rapidly and effectively to allow the developer to exploit database extensibility to its full potential.

1.3 Focus and scope

This thesis explores the mechanisms to integrate additional functionality into ORDBMS. To do this, an initial understanding of extensibility in current database technology was necessary. A study of the object-relational features in existing ORDBMS was therefore done, highlighting commonalities and demonstrating how individual features may be used to add new functionality to the database. The explorations in this thesis use content-based image retrieval as the main test domain for extensibility, and thus also require an understanding of content-based retrieval techniques. It also used an extensible indexing scenario as an additional test domain.

A typical way to access multimedia data is to store the actual data in a database, along with a set of keywords and textual tags that describe the content of the data. Data is accessed by manually browsing the entire database according to subject categories and keywords, which are intended to capture the context of each data item in the database, as used in applications such as Chabot [OST95]. This approach, however, has several shortcomings: data items belonging in the same category to one user may not necessarily belong to the same category according to other users. Browsing the entire category can also become cumbersome and time-consuming as the amount of data in the database or category increases. Querying data based on their actual content is, naturally, a preferable solution and important feature for multimedia databases [WNM+95]. Retrieval based on content is guided by the features extracted from the data itself, such as shapes or the combinations of colours in images, for which today's technology does not provide sufficient solutions [WEK99].

The thesis does not seek to provide novel solutions to either content-based retrieval or indexing problems. Rather, it investigates how already developed techniques may be integrated into the database. The actual implementations explore the suitability of available development tools, and do not attempt to suggest radically new constructs of database extensibility. In this way, the thesis evaluates the implementation effort required from the user and presents a real life test of the extensibility mechanisms in ORDBMS, and more specifically in Oracle9*i*.

Oracle is one of the commonly used ORDBMS with multimedia management components, one of which provides support for image retrieval applications. The shortcomings of this component validate the importance of extensibility in databases. Realising the need for extensibility, Oracle has provided, since version Oracle8i, the *data cartridge* mechanism, which is used to integrate the various practical scenarios in this thesis. The data cartridge mechanism is used to demonstrate how

the database can be extended with additional functionality (such as user-defined data types, additional indexing mechanisms and query optimisation techniques) and to explore the practical implications of using such extensions.

This thesis reports on the opportunities and main challenges of extensibility in Oracle, as gathered from our practical experiences with integrating sample image retrieval and indexing techniques. The report is not intended to be an evaluation of Oracle DBMS, but rather an investigation of how the extensibility mechanisms in Oracle may be exploited to bring many of the key benefits of object-relational technology to user-defined components in the database.

1.4 Thesis organisation

The rest of the thesis is organised as follows:

Chapter 2 discusses the required features of an extensible database system. It motivates the need for extensibility and provides the object-relational features found in commonly used ORDBMS, with special emphasis on Oracle and Informix databases. The chapter is concluded with a discussion of the opportunities and challenges offered by extensibility mechanisms in these databases.

Chapter 3 introduces image retrieval as the main domain for testing extensibility in this thesis. The aim of the chapter is to provide the necessary background information to investigate issues that arise in extending the database with retrieval functionality. The chapter first introduces a general model for content-based image retrieval systems. It then discusses the specific approaches used in content-based retrieval systems, making references to previous work in the area. A common example is used to demonstrate the retrieval capabilities of image retrieval extensions in Informix and Oracle databases. Finally, the chapter is concluded with comments on the state-of-the-art image retrieval functionality.

Chapter 4 presents the development and integration of the first data cartridge. The aim of the chapter is to show how the data cartridge mechanism can be used to add user-defined data types and their operations to the database. The demonstrated data cartridge implements a very simple colour-based image retrieval technique. The chapter also demonstrates how the same retrieval functionality can be achieved by extending the relational model. The chapter is concluded with a comparison

between the data cartridge and the relational implementation to highlight the benefits of using data cartridges.

Oracle provides a set of interfaces to extend indexing and enhance query optimisation. Chapter 5 continues the integration of the data cartridge introduced in the previous chapter to explore the use of these interfaces. A discussion on the extensibility of indexing and query optimisation is then provided, focusing on the functionality of these interfaces.

Chapter 6 develops and integrates a more complex data cartridge. The cartridge implements a well-known face recognition algorithm that uses Eigen faces for recognition, with the aim of exploring the data cartridge mechanism in greater detail. Initially, the chapter explains the architecture and objectives of the cartridge. It then presents the implementation of the cartridge, describing it as a mapping of a standalone Java application to a database application. The last section of this chapter discusses the opportunities and limitations of the object-relational mapping and features presented in this chapter.

Chapter 7 presents the final and most complex data cartridge implemented in this thesis. The cartridge implements an extensible indexing system, with the aim of exploring the mechanism of extensibility in the data cartridges themselves. Firstly, the requirement of the data cartridge is presented. The next section then demonstrates how it is implemented. The last section discusses the requirements for implementing extensible data cartridges and the issues that arise in extending them.

Chapter 8 presents our experiences with integrating data cartridges into Oracle9*i* and summarises the findings of the thesis. The chapter concludes with suggestions for future work.

Chapter 2:Extensibility in databases

Before extensibility in ORDBMS can be explored, it is important to understand the unique requirements expected from ORDBMS. This chapter presents a review of the extensibility mechanisms in ORDBMS, and introduces the themes that recur throughout this thesis. It explores the functionality offered by the extensibility features in Informix and Oracle databases and concludes with an informal discussion of the strengths and limitations of extensibility mechanisms in these two databases.

2.1 The need for extensibility

The advent of complex and unstructured data applications such as multimedia applications, have intensified the challenges faced by the DBMS. Unlike structured data, unstructured data are often very large, cannot be decomposed into simple, standard components and thus require adroit techniques to efficiently manage and access. As mentioned in chapter 1, conventional RDBMS store the pointers to this complex data inside the database, while the actual data is stored outside the database, in operating system files [RAG00]. Alternatively, data could also be stored inside the database as undifferentiated Binary Large Objects (BLOB). This simplified solution however, does not serve all the application requirements because BLOB contents are not interpretable by the DBMS.

Major DBMS vendors have integrated independent software components to manage complex application requirements and enhance the DBMS with additional capabilities. Notwithstanding the good, interim solutions offered by these components, it is only obvious that they cannot act as a panacea for all the required functionality. The revolution in information technology is continually increasing the complexity of data, and hence demanding sophisticated models to handle these data. It is thus essential for the DBMS to provide ways for users to extend it with new capabilities.

The impetus to extensibility is to add new functionality to the DBMS with user-defined data types and functions. In this way, the database can be tailored to the needs of specific application domains. The extensible architecture thus diminishes the need to develop large DBMS with multitudes of data types that attempts to keep up with new demands of complex data applications. It also

increases the potential of building useful applications since the added functionality is built around specific application requirements.

2.2 Object-Relational features

Research exploring database extensibility has been active since the mid eighties [CAR87], where efforts such as [STO86] extended the database with user-defined data types and query optimisation techniques for columns and operators in a relational database system. In defining what an extensible DBMS is and the functionality that it should provide, a number of researchers used the two-by-two matrix proposed in [STO96] and depicted in Figure 2 - 1, to classify applications according to the complexity of their data and the query capabilities they require from the system. Although this classification assumes only two discrete possibilities for each group (i.e. data is either simple or complex, and applications either require query capabilities or they do not), it captures the essence of the different applications requirements available today. The growing consensus of what an ORDBMS should provide in the database field is derived from the top right corner of this classification, and, has even been used in the design of ORDBMS benchmarks such as Bucky [CWN+97] and BORD [LKK00]. According to [STO96], the four main characteristics of an extensible DBMS, which are also used to explore the level of object-relational support offered by Informix and Oracle database in this chapter, are:

- An extensible base type system
- Complex object support
- Inheritance
- A rule system

	Simple data	Complex data
Query	e.g. emp – dept relations	e.g. video applications
No Query	e.g. text processing	e.g. c++ applications

Figure 2 - 1: A classification of data applications

2.2.1. An extensible type system

Each DBMS comes with a set of built-in data types such as integers and strings that specify the internal representations of the different kinds of data stored in the database. As already explained, these data types are not appropriate for all kinds of applications. Thus, ORDBMSs allow new base types to be defined as opaque types. Opaque types are defined in external languages such as Java or C, and their implementation is hidden in routines so that database users will not know what they look like. Access to base types is controlled by these routines, to prevent the type from being corrupted accidentally. For each distinct opaque type, the type definition specifies its view to the database using characteristics such as the storage format, constraints and valid range of data.

Since base types are semantically different from the built-in data types, they cannot, in general, be manipulated with built-in functions and operators (such as + and -). Extensible databases should therefore allow each base type to have its own defined behaviour that captures the individual semantics of specific base types. The database must also allow the internal representations of the data type to be encapsulated in the application so that it can be changed without affecting the entire operation of the DBMS. The DBMS must also provide integration mechanisms required to allow base types to be exported from the definition routines to the database.

Other required functionality that a DBMS should provide in terms of an extensible type system includes:

- No limitations on the number of base data types and functions
- Security mechanisms to prevent user-defined functions from corrupting the database
- Flexible methods of adding and dropping types and functions
- Ability to use the data types in both the client and server environment
- Seamless integration of user-defined types and functions so that they are used in the same way as built-in types and functions.

2.2.2. Complex data types

Complex types are constructed from source types such as build-in types or from other user-defined data types (UDTs). Complex types have attributes that hold the state information of the object, and methods, which implement the operations on the object type. Unlike base types, complex types are

implemented in a transparent manner, which means database users can view their implementation details. Complex types are defined as collection types or row types, where collection types group a set of values of a single data type in a single column, and the row types take a group of varying data types to store it as a single column. Each row type is declared by specifying a unique name for the type being defined, along with its unique attributes and their corresponding data types. Collection types on the other hand are declared using data type names only. As an example, a row type may be defined as:

```
CREATE ROW TYPE row_t (col1 INTEGER, col2 VARCHAR(30));
```

and a collection type may be created as:

```
CREATE TYPE col_t AS VARRAY(13) OF row_t;
```

Each attribute in the complex type is stored using a known data type, and thus assumes the operations and functions of that data type. As such, all the valid operations of the defining type are also valid on the attributes of the same type in the complex type. However, methods that implement the operations on the complex type as a single entity still need to be defined. These methods are functions or stored procedures, which may be written in a language external to the database such as C or Java.

Some of the required characteristics of a DBMS that supports complex types include:

- Definition of type constructors – the system should either automatically assign a constructor method or allow the user to define one for each object type created
- Sets to allow objects to be compared using traditional set methods
- Arrays to allow users to store and access items at particular offsets
- Nested tables
- References to allow pointers to row objects to model association among objects and hence reduce the need to use foreign keys

2.2.3. Inheritance

Inheritance is another necessary characteristic for extensibility in databases. Inheritance defines a type of type hierarchies so that subtypes can be derived from supertypes previously defined. In

some databases, such as Oracle, a supertype should be declared using the keyword NOT FINAL in order to permit subtypes or NOT INSTANTIABLE to prevent them from being instantiated. Types are declared FINAL by default, which implies that they cannot have subtypes.

A subtype must not be allowed to be entirely different from its supertype. It is therefore necessary that the inheritance link between the subtype and supertypes is maintained throughout the inheritance hierarchy, so that any changes made on the supertype is also reflected on the subtype, unless the subtype is re-implemented. To support inheritance, the DBMS must have capabilities to:

- add attributes that are not found in the supertype
- add new methods that are not found in the supertype
- allow method overloading
- allows subtype methods to override supertype methods
- provide security restrictions to ensure that the inheritance relationship is maintained.

2.2.4. Rules and triggers

The DBMS must allow the following features to support rules and triggers:

- Events and actions – Rules and triggers are used to ensure the consistency of the database. Rules must provide the capability to execute a task just before or just after an event has occurred, while triggers must support update, insert and delete queries as necessary.
- Integration of rules with inheritance and type extension
- Rich execution semantic for rules to support different kinds of actions

2.3 Extensibility in Informix

Informix has been one of the leading ORDBMS since its acquisition of the Illustra Server in 1996 [INF97]. Described as one of the most extensible databases [KOR99] [WEK99], Informix database offers the DataBlade extension to provide an API that supports a wide range of object-relational capabilities. A *DataBlade Module* is thought of as a software component that can be plugged into the database to extend the server's functionality with new data types and their methods [INF01]. DataBlade modules are completely integrated with the server, but can contain components that are

executed from the client side. Once a DataBlade module is built and plugged into the database server, it is used at par with built-in server components. This section discusses the four main extensibility components of the DataBlade Module API that can be enhanced to enhance Informix database with new functionality.

2.3.1 User-defined types

There are three varieties of user-defined types in Informix, namely [INF97]:

- **Distinct**, which inherit their physical characteristics from existing types but can have different behaviours. These types are usually used to tailor and give more meaning to the application data type. For example, the definition of the Rand currency may be given as:

```
CREATE DISTINCT TYPE rand AS MONEY;
```

- **Row data types**, which stores a group of built-in data types into a single data type. Row data types are used for data types whose individual fields require exclusive access. As an example, a data type, `student`, may be created as:

```
CREATE ROW TYPE STUDENT (student_number INTEGER, name CHAR(40),
phone CHAR(15));
```

- **OPAQUE data types,** which are structures (for example, C structures) used to create indivisible objects whose actual structures are not transparent to the user. These data types are used for large objects and objects whose representation should be hidden from users. Each definition of an opaque type specifies the appearance of the data type (how it is displayed to the user), its internal representation (how it is stored in the database), and its standard functions (how the DBMS applies the standard SQL functions and internal functions to the stored data type).

In addition, Informix supports single inheritance among named row types. Tables based on row types can also become typed tables, and thus are part of the inheritance hierarchy.

2.3.2 User-defined functions

User-defined functions (UDFs) in Informix are used to define and implement routines, aggregates, casts, errors and user-defined interfaces [INF98]. The definitions specify the function name, the input and output parameters, and particulars of how it might be activated. The implementation on the other hand, specifies the actual operations of the function, and may be done using SQL, Stored Procedure Language, C, C++ or Java. UDF can be called in SQL statements from any place where built-in functions can be called.

2.3.3 Extensible indexing

Indexing methods in Informix are called *access methods* [KOR00][INF01]. An access method consists of a collection of *purpose functions*, which are routines used to perform indexing tasks such as inserting, updating and fetching data from the index. The access method can either be a *primary access method* or the *secondary access method*. Primary access methods are developed using a Virtual Table Interface (VTI), which is an extension to the DataBlade API, and are used to manage external data as if it were stored inside the database. Secondary access methods on the other hand, manage index data and are used to add new access methods to the database. These methods consist of functions that the Informix Database Server calls to perform several index definition tasks including creating, opening, scanning, closing, and dropping the index, and index maintaining routines such as inserting, updating, and deleting index entries. The access method is created using the following two steps:

- Define and code functions that implement the operations that are eventually associated with the purpose functions. As an example, a method to create the index may be defined as:

```
create function idx_ create (pointer)
returns int
external name
              '$IDXDIR/idx_demo.bld(ix_create)'
language c;
```

The external name variable refers to the location and name of the library with the implementation code for the function. This library, called `idx_demo.bld` in this example, has a corresponding function called `ix_create`.

- Bind all purpose functions to their corresponding access tasks such as:

```
create secondary access_method idx_am
(
am_create     = ix_create,
am_insert     = ix_insert,
...
am_sptype='S');
```

The index is now ready for use. When a statement of the form "`create index ... using idx_am`" is issued to the database, the server automatically executes the `idx_create` UDF that is registered for the `am_create` task.

2.3.4 Extensible optimiser

Built-in data types have built-in relative cost functions and statistics routines, which are used to calculate the cost and selectivity of executing built-in functions. For user-defined functions however, user-defined relative cost functions and user-defined statistics can be defined. The relative function costs is used to determine the order of processing the `WHERE` clause in a `SELECT` statement, while the selectivity function is used to determine the number of rows that might be returned by a function, given the arguments. The routines with a lowest cost are performed first, and the most expensive routines are performed last. In general, the cost of the routine is computed using the formula [INF01]:

```
Lines_of_code + (I/O operations * 100),
```

while selectivity is determined by computing the statistics on the data columns. However, optimiser can only be extended for user-defined functions coded in C.

Informix Database Server also allows user-defined *commutator* and *negator* functions to be defined. Commutator and negator functions are executed when the commutation or negation of the query is much faster to execute than the original function. Commutator functions return the same results as the original function, with arguments in the reverse order, while negator functions return the opposite results with arguments in the same order.

2.4 Extensibility in Oracle

The *data cartridge mechanism* is Oracle database's main mechanism for extending its capabilities [GIE02a]. With data cartridges, the existing database functionality can be extended with new functionality, by allowing new data types and their methods to be added to the database. The other features that can be extended include the indexing system and the query optimiser as shown in Figure 2 - 2 below.

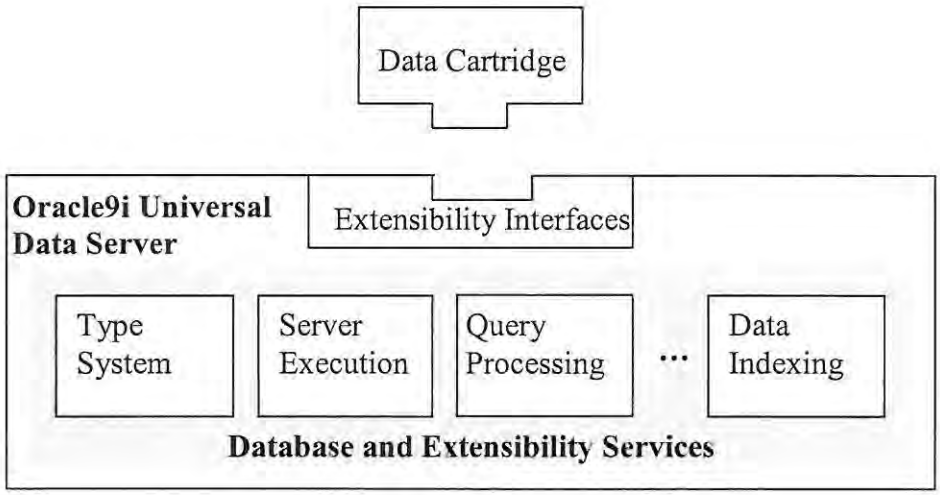

Figure 2 - 2 : Oracle data cartridge services (*Based on Figure 1 - 1 [GIE02a]*)

The main characteristics of data cartridges are the following [GIE02a]:

- Data cartridges are server-based

The main cartridge components, such as the data types and their functions, are stored in the server. Although other additional components, such as external libraries and other resources can be stored outside the database server, they are primarily accessed from the server, where they are dispatched as external routines.

- Data cartridges extend the server

The new data types and their methods introduced by the data cartridge are previously unavailable in the server, and hence extend the server capabilities. User-defined functions can also be directly added to the general database machinery, extending the capabilities of the overall database.

- Data cartridges are integrated with the server

Once new data types and methods are defined and loaded into the server, they can be used at par with built-in data types and functions. New data types and functions are also integrated within the server engine, so that the server mechanisms such as the query optimiser and the indexer can recognise and respond to the new extensions in the same way it responds to built-in data types.

- Data cartridges are packed

Data cartridges can be packed as a unit, which can easily be installed in the database. Packing data cartridges as a unit ensures that the necessary cartridge components are installed in the database under the same schema and with the necessary privileges. It also ensures consistency among the cartridge components in different databases.

The rest of this section briefly summarises the standard Oracle server services that can be extended by the data cartridge developer to enhance the capabilities of the database.

2.4.1 Extensible type system

Apart from the standard SQL data types such as integers and dates, Oracle also offers support for additional data types such as collections (VARRAY and nested tables), internal objects (BLOB, CLOB), References, external files (BFILE) and user-defined object types [GIT02]. The extensible type system is the gist behind adding new user-defined object types to the database, which are primarily defined using built-in data types and other user-defined objects previously defined. An object type must have at least one attribute and can have methods that are used to manipulate the object. The attribute specifies the name of the attribute and its type. The following example illustrates how a new type is created in Oracle:

```
CREATE TYPE Student AS OBJECT
(
        StudentNumber     NUMBER,
        name          VARCHAR2 (30),

MEMBER FUNCTION register(stud IN Student) RETURN NUMBER,
MAP MEMBER FUNCTION StudentToInt RETURN Integer
);
```

The `Student` object declared above contains an attribute `StudentNumber` of type `NUMBER` and the attribute `name` of type `VARCHAR2`, which holds variable characters. In addition, it has two methods declared as `MEMBER` functions.

2.4.2 Extensible server execution environment

The components of the data cartridge may be developed using PL/SQL, Java or C. This offers two main advantages. Firstly, it offers flexibility in the language that can be used to develop the components. Secondly, it fosters code reuse as other applications that are not integrated with the database can use the same code. Oracle's PL/SQL and Java are both interpreted languages, and thus safely run in the database as the DBMS ensures that each statement is safe before executing it, at the cost of restricting the power of the language. C on the other hand is compiled, and is not considered safe to run in the database. As a result, code written in C is implemented as external routines, which run outside the database's server space. This insulates the server's functionality from program failures resulting from C routines. However, external routines incur a dispatch overhead because they are executed outside the database.

2.4.3 Extensible indexing

Oracle has standard indexing methods such as text indexes, B-trees and Hash indexes. Oracle cannot handle index values derived from user-defined data types and their operations, and do not permit indexing column with LOB values. For data that require complex indexing techniques, the user must define and implement the new access method in the database. Oracle provides an extensible indexing API called *Oracle Data Cartridge Interface Indexing* (ODCIIndex), which allows user-defined indexes to operate in the same way as built-in indexes. The index can either be stored inside the database as a database table, or outside the database as an external file. Regardless of how it is stored, the user must define all the necessary indexing functions such as how the index will be searched during query processing and how it should be maintained when new content are inserted and during update operations.

Oracle also offers function-based indexes, which are used to improve performance on computational intensive query expressions. The query expressions can be computed once and stored in an index so that when access to the expression is required again, the values are already computed.

Function-based indexes can also be used to build indexes on object type columns and object references, which standard Oracle indexing techniques cannot do.

2.4.4 Extensible optimiser

The optimiser determines the efficient way to process a statement. The choices available to the query optimiser are also maximised in Oracle with the introduction of user-defined functions (UDF) and new access methods. Oracle provides an interface called *Oracle Data Cartridge Interface Statistics* (ODCIStats), which is used to extend the optimiser with user-defined selectivity and cost functions. The optimiser considers the objects and conditions specified in the statement to process a query, and thus require the knowledge of the defined functionality so that it can use it appropriately.

Optimising queries involves enumerating all the possible plans for evaluating the statements and estimating the cost of each enumerated plan so that the plan with the least estimated cost can be chosen. As such, statistics that quantify the data cost of accessing database objects such as tables, columns, indexes and partitions must first be generated. Statistics are collected using the system provided *DBMS_STATS* package. To optimise queries, Oracle stipulates that the optimiser must evaluate the query expressions and conditions and transform the expressions into equivalent join statements if necessary. The optimiser then chooses the cost-based or rule-based approach and determines the goals of the optimisation i.e. whether the statement is optimised for best throughput or for best response time. Depending on the goal, the optimiser then uses the collected statistics to calculate the selectivity of predicates, estimate the cost of each execution plan and choose the path with the least associated cost to execute the statement.

Although it is not mandatory to extend all the data cartridge extensibility services described in this section, a data cartridge should, at minimum, define a single object type. If the extensible component only defines new functionality without the definition of new data types, Oracle recommends the development of stored procedures packed as database packages instead.

2.5 Conclusions

DBMS extensibility addresses the need for extending the database with new functionality. Both Informix and Oracle database successfully present some degree of extensibility, and are therefore potential databases for the integration of additional retrieval techniques. This chapter discussed the features of Informix and Oracle databases that can be extended to provide extensibility in four key areas: user-defined data types, user-defined functions, index structures and the query optimisation techniques, to meet the extensibility requirements discussed in [STO96].

In addition to the discussed features, Oracle and Informix databases also support extensions not discussed in this chapter, including additional languages for writing database extensions and server procedures (PL/SQL for Oracle and SPL for Informix), enhanced support for large objects and support for access to external data. Oracle provides data cartridges as the extension packages that allow user-defined functionality to be packaged while Informix offers datablades. All these features are essential to enable ORDBMS to support a broader class of application requirements.

The extensibility provided by these two databases reduces the need to process complex data in client application environments. Server-centric processing promotes code reuse, as most of the code is stored in the DBMS server and can be shared among all applications accessing the database. When extending database components, the actual functional implementations are decoupled from the code interfaces, allowing the functional implementation to change without affecting the entire database application. Extensibility mechanisms in these two databases therefore accrue the benefits of modularity.

The number of object-oriented and relational features provided by both Oracle and Informix is comparable. One of the differences between the two systems, however, is at the level of integration with C functions. Informix datablades are more tightly integrated into the DBMS in comparison to data cartridges. C functions are compiled and executed inside the database server space in Informix, although some common language features are only used within certain limitations because their implementation uses programming techniques that are not permitted in server-based routines. In Oracle, C functions are stored as external procedures, and although called from the server, they have to be dispatched and executed in an external address space. A main drawback of running external routines outside the database is that, external routines are not covered by the DBMS's support for concurrency and recovery. If the code is defective however, it does not crash the server.

With Informix DataBlades, flawed code may block other users, returning incorrect results or even in extreme cases, bringing down the server.

Objects in these databases are still stored in tables, although in cases where data does not neatly fit into tables such as BLOBs, only a locator to the actual data may be stored in a table. Just like classes in object-oriented programming, objects similar in behaviour and attributes, are stored under one type. Declaring an object type does not allocate storage, so, object types must be instantiated to create object instances. The concept of types, supertypes and sub types allow for inheritance and impose the abstraction for types just like classes in object-oriented programming. Intrinsically, objects are represented as a collection of other objects, which ultimately when followed to the super class object, ends up in a set of simple attributes stored in tables. Thus, the concept of an object type in these databases is realised by nesting tables within other tables.

Chapter 3:Test domain

This chapter describes content-based image retrieval as a test domain for database extensibility. Its discussion is aimed at providing a framework that is sufficient to explore the integration of content-based image retrieval techniques into database systems. A summary of the general approaches to content-based image retrieval proposed in the literature is provided. The level of support offered by content-based image retrieval extensions in Informix and Oracle9i is also explored using a common example. The chapter concludes with a highlight of the limitations of content-based image retrieval.

3.1 Introduction

In this thesis, database extensibility is investigated by exploring mechanisms that integrate content-based image retrieval techniques into the DBMS. In traditional RDBMSs, multimedia data such as images are stored using a Binary Large Object (BLOB), along with textual tags that identify and describe the relevant contents of the data. The database is oblivious to the actual BLOB content, which is usually accessed by using the textual tags. Any BLOB processing is done in the program application environments such as C++ or Visual Basic. Due to the many problems with this approach, current database systems moved a step further from mere BLOB storage, by providing components that offer multimedia data processing.

Using textual tags to retrieve images from large databases has several drawbacks. Apart from the basic image attributes such as the width and height of the image, textual tags are usually added manually by the annotator, as images do not come with accompanying textual information that describe the content of the image. The annotation process thus does not only become cumbersome and time consuming for large, heterogeneous image databases, but can also be very subjective, and limited to the vocabulary of the annotator. In databases with complex image patterns, it is sometimes extremely difficult to capture the meaning of images using keywords. This complicates image querying, often resulting in keyword mismatches, especially when the database users are from various domains of expertise.

Another alternative approach to image retrieval is by browsing the database manually. Browsing allows images to be viewed one by one, increasing the likelihood of finding the required images from the database. Browsing is, however time consuming for large databases. Recent image retrieval techniques focus on techniques that attempt to provide content-based access to images. Content-based image retrieval uses features that are automatically derived from the images themselves, to compare and match images for similarity.

The aim of a content-based image retrieval system (CBIRS) is to mimic human recognition capabilities when searching for a desired image in a large and varied collection. Various research fields, including computer vision and pattern recognition, have combined in search of more effective techniques that make image collections easier to search. The extent to which this collaboration is currently being realised in various image retrieval technologies is discussed in this chapter, using a demonstration of the functionality supported by image retrieval extensions in database systems. In particular, the chapter demonstrates how image retrieval applications can be built in Informix and Oracle9i using database-specific extensions. It highlights the functionality offered by these extensions using a common example, and gives a summary of the limitations of these image retrieval extensions. The chapter then concludes with comments on the state-of-the art image retrieval techniques.

3.2 Content-based image retrieval model

Assuming a large number of images stored in the database, the main problem that content-based image retrieval seeks to solve is as follows: "*Given a query image, retrieve a set of images from the database that are most similar to the query image*". Several approaches have been proposed to solve this problem. In general, these approaches all seek to:

- Identify the features that capture similarity between images
- Isolate and extract these features from the image
- Efficiently index the features to facilitate fast retrieval
- Compare and match stored features with query features
- Compute and determine image similarity

Content-based image retrieval systems define methods for defining and computing similarity in images. This is better explained using the following general concept of image retrieval:

Assume $I = (I_1, I_2, I_3, \ldots\ldots I_n)$ is a database containing n images. Each database image I_i must be associated with a feature vector f_i that captures the meaning of the contents of the image. If A is the algorithm for extracting the feature description of the database image I_i, then the mapping from the image data to the feature vector is described as:

$$A : I \rightarrow f$$

where $f = (f_1, f_2, f_3, \ldots\ldots f_n)$.

Given a query image I_q, the aim is to retrieve images from the database that are most similar to I_q. To solve this problem, the image I_q is also mapped to a corresponding feature vector f_q using A. The retrieval systems then determines the similarity between the two images I_i and I_q by computing the matching function $d(f_i, f_q)$, which is a distance measure between the feature vectors f_i and f_q. A similarity threshold, which can be tuned to achieve the different levels of similarity, can also be specified.

The image retrieval strategy explained above is depicted Figure 3 - 1.

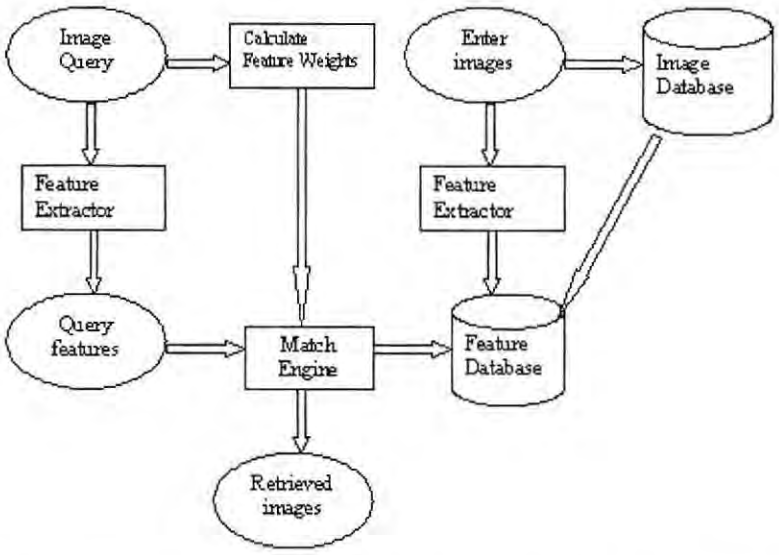

Figure 3 - 1 : The retrieval strategy of a standard content-based image retrieval system

3.3 Approaches to image retrieval

Different approaches have been developed to implement the retrieval strategy of a content-based image retrieval system. This section provides some of the methods used in feature extraction, indexing structures and similarity computations.

3.3.1 Feature extraction

The features that describe the content in images are chosen in such a way that they capture the desired similarity between the images, so that images that differ slightly correspond to a much more similar representation than images that are completely different. Features are automatically extracted from the image itself, and are stored in a feature vector. Some of the commonly used features are:

- **Colour with/without spatial constraints** - Literature describes several colour extraction approaches [SMC96] [RCH99], but most are a variation capturing the content of different colours within an image and constructing a representation technique that captures this distribution. A comprehensive review of representative colour feature extraction methods and their implications can be found in [RCH99], while some representative studies of colour spaces and colour perception can are found in [WYA97].

- **Texture** - Texture is another feature that can represent the content of an image. Texture describes the low-level arrangement of structures that hold a common homogenous content such as graininess, coarseness or smoothness. A variety of texture-based techniques calculate the relative grey levels in pairs of pixels from each image, which are used to determine certain textures features such as coarseness and roughness. A study of texture features is found in [TMY78].

- **Shape** - Various schemes for shape-based features have been proposed in literature, but [MKL97] categorised them as information preserving (or unambiguous) and non-information preserving (or ambiguous) depending on their ability to reconstruct the approximate shape from the extracted features. In general, shape segmentation techniques are applied to database images to represent shapes using either *boundary-based* or *region-based* shapes. *Boundary-based* shapes consider the extreme borderline of the shape, while

the *region-based* shape considers the whole region covered by the shape. A discussion on shape features is given in [MKL97].

- **Combination of Colour, Texture, Shape and other features** - A number of image retrieval systems extract multiple features to describe the content in images. Some of the most popular retrieval systems that use multiple features include Query by Image Content (QBIC) which extracts colour and texture features [FS95], BlobWORLD which extracts colour, texture and spatial information [CTB+99] and MARS which uses colour, texture, shape and spatial location features [POM99]. Other features found in literature include spatial and topological relationships, semantic associations such as aggregation and generalisation and pattern recognition methods summarised in [AKJ02] and [RCH99].

3.3.2 Similarity computations

To determine the degree of similarity between a query and a set of database images, the features of the query image must also be analysed and extracted. Similarity between the images is then determined by matching the features of the query image against the features of the images in the database. This similarity is usually determined using a similarity measure, which is a matching function that computes the distance between the extracted features (also called signatures) of the query and comparison image. The similarity measure should be a metric with properties such as symmetry, transitivity and linearity.

The distance similarity measures used depend on the features extracted from the image. As an example, some of the image similarity measures used to compare images for colour similarity include the *L1-Distance*, which calculates the sum of the absolute value of differences in colour histograms, the *L2-Distance*, which compares the sum of squared differences of the colour histograms, and the *Quadratic Distance Metric*, which assumes that the perpetual distance between two points in the feature space corresponds to the Euclidean metric. The *L1-Distance,* commonly for the query image q and the database image I is computed using the formula:

$$D_{color}^{(q,i)} = \sum_{j=1}^{n} | H_j^q - H_j^i |,$$

where H_j is the j^{th} bin in the histogram.

The Euclidean Distance between the query image and the i^{th} image on the other hand is calculated as:

$$D_{color}^{(q,i)} = \sum_{j=1}^{n} \sqrt{(H_j^q - H_j^i)^2}$$

Using the above distance measures, the distance between two identical images is zero (0). More similar images have smaller values of the distance measure, while less similar images have bigger values. To find the similarity between the query and all images in the database, the distance measure must be applied between the query and all the images in the database. Distance values can then be ranked in increasing order of importance so that more relevant images appear earlier than less irrelevant images in the result set.

3.3.3 Feature indexing

The features extracted from the image are stored as points in a multidimensional feature space. For smaller databases, image similarity can be computed by sequentially comparing the feature vector of the query image to the feature vectors of all images stored in the database. Sequential access, however, becomes time consuming as the number of dimensions and the size of the database increases, requiring efficient methods to index and facilitate access to data. Typical access structures used to support this indexing are known as spatial access methods and metric trees, and include SS-tree [WHJ96] [FUT99], R-trees [GAG98], SR-trees [KAS97] and M-trees [CPZ97]. These access structures however, have exponential time and space complexity as the number of dimensions increases, making them almost similar to scanning the database sequentially. Multidimensional indexing is therefore still a major problem today.

In general, efficient indexing structures also depend on the features extracted from images. Retrieval systems that use multiple features require a separate indexing structure for each feature, such as colour and shape, to be built. Using this approach, however, does not sufficiently support queries involving composite features such as queries that require both shape and texture features simultaneously. These queries are therefore processed using a hierarchical approach, where each feature is applied against the appropriate index and the similarity function is determined for each feature. The results of individual feature types are then merged to answer the query.

3.4 Content-based image retrieval in ORDBMS

Image retrieval functionality is supported to varying degrees in commercial ORDBMS. To demonstrate some of the functionality offered by these components, a simple student application requiring content-based access to identify the student and retrieve particular student details (such as student number, student name, course and current year of study) is used. This application is stored using a single table shown in Figure 3 - 2 below:

Students		
PK	**StudentNumber**	**SHORT**
	StudentName Course StudentPhoto PhotoSignature	CHAR(50) CHAR(50) LONGBINARY LONGBINARY

Figure 3 - 2 : Representation of the student table

It is the implementation and level of support offered over the `StudentPhoto` and `PhotoSignature` columns that are of particular interest in this section. While all the other columns can be stored using common data types, the `StudentPhoto` and `PhotoSignature` columns (implemented as LONGBINARY types in Figure 3 - 2) require a product specific data type in each database. The following two sections describe implementation of the application in Informix and Oracle9i.

3.4.1 Informix Excalibur image retrieval DataBlade

Informix has several datablade modules to manage digital content. Available datablades include modules for indexing and searching text information, for managing geospatial information, for the management of time series and temporal data and for storing and managing image information. The *Excalibur Image DataBlade* [INF99] and the *Image Foundation DataBlade* [INF00] are the two DataBlade modules used to manage images in Informix. In addition to storing and providing basic access to images, the *Image Foundation DataBlade* module has functions to convert data among several image formats using industry-standard *CVT* command functions. Image Foundation also supports image transformation operations such as scaling, cropping, rotation and selection of particular regions of interest from the image.

Content-based search in Informix is provided by Excalibur's Image retrieval module. Image retrieval is performed based on colour, shape and texture attributes. The contents of the Excalibur Image DataBlade Module are shown in Figure 3 - 3.

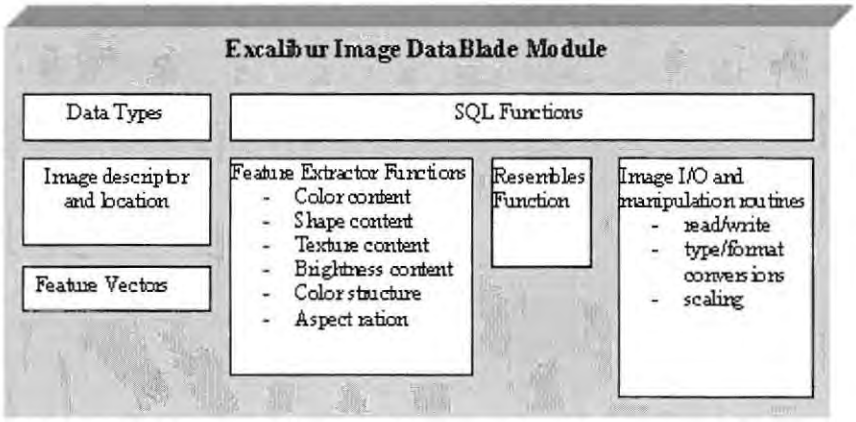

Figure 3 - 3 : Excalibur Image DataBlade (*Based on Figure 1 - 2 [INF99]*)

Assuming Informix Dynamic Server with Universal Server Option is installed with the Informix Large Object Locator DataBlade Module and Excalibur Visual Retrievalware DataBlade product, the steps to create, store and retrieve supported raster image formats in the database according to Figure 3 – 2 are:

1. Create the table for storing the images.

```
CREATE TABLE students
 (
StudentNumber  NUMBER     PRIMARY KEY    NOT NULL,
StudentName           VARCHAR(50)        NOT NULL,
Course                VARCHAR(50)        NOT NULL,
StudentPhoto          IFDIMGDESC  NOT NULL,
PhotoSignature        IFDFEATVECT
);
```

The `StudentPhoto` is of type `IfdImgDesc`, which is used to store image attributes including the location, the format, pixel type, pixel width and height, and other image-relevant parameters in Informix. The `PhotoSignature` is assigned to the `IfdFeatVect` type, which stores the signature as a combined feature vector and is used to perform content-based search.

2. Insert a record into the table.

```
INSERT INTO STUDENTS (StudentNumber, StudentName, Course,
StudentPhoto) VALUES
(6011760,'Kandeshi','Doctor',IfdImgDescFromFile('/imagedirectory/ka
ndeshi.gif'));
```

Note that the `PhotoSignature` field holds a NULL value.

3. Populate the signature by extracting feature vectors from the inserted image.

```
UPDATE Students
SET photosignature = GetFeatureVector( studentphoto)
      WHERE studentnumber =6011760;
```

The `GetFeatureVector` function computes the image signature by extracting the following six features:

- *Colour content*, that captures the colour and its location in an image
- *Shape content*, which measures the relative orientation, curvature and contrast of lines within an image
- *Texture*, which measures the flow and roughness of an image
- *Brightness structure*, which measures the brightness at each point in the image
- *Colour structure*, which measures the hue, saturation and brightness of the image and
- *Aspect Ratio*, which is a measure of width to height in an image

The signature returned by the `GetFeatureVector` function is then used to store the signature into the `PhotoSignature` column.

If several image records were inserted as shown in step 2, multiple image signatures can be inserted by setting the `WHERE` clause as:

```
      WHERE photosignature IS NULL;
```

4. Search the database for images similar to the image inserted in step 2.

```
SELECT c.studentname, RANK
FROM students c, students q
 WHERE
  q.studentnumber = 6011760
 AND
  RESEMBLES ( c.photosignature, q.photosignature, 0.70,1,1,1,0,0,0,
rank #REAL)
ORDER BY RANK;
```

The RESEMBLES function is of particular interest in this discussion because it performs content-based image retrieval. The function accepts the following arguments:

- The signature of the comparison image
- The signature of the query image
- The threshold parameter, which must be a real number between 0 and 1. Only images that are above the threshold are considered similar to the query. Images with the similarity score less than the threshold value are therefore not included in the answer set.
- A set of the features weights used in matching. The cumulative sum of weights should not exceed 100. Features that do not contribute to the final resemblance score should be set to 0.
- An output variable that holds the final matching score

The above statement compares a query image (q) with the comparison image (c) already stored in the database. Since a threshold of 0.7 is specified, only images with a match score above 0.7 are returned as an answer to the query.

3.4.2 Oracle image data cartridge

Oracle has a single integral feature that can be used to store and manage media-rich content in the database. This feature, called *inter*Media, is designed to manage specific multimedia data applications including geographic location information, images, audio and video in an integrated manner with other standard data types [WAR01]. The services provided by *inter*Media include the ability to store, manage and retrieve data. *inter*Media also supports web technologies and annotation services for multimedia data.

*inter*Media provides various object types to manage different data types. One such data type is `OrdImage`, which supports the storage, management and retrieval of image data. `OrdImage` data type has an associated `OrdImageSignature` type, which supports content-based image retrieval based on colour, shape, texture and location. Colour captures the distributions of colours in the image regardless of the location, but can be used in conjunction with location to capture the spatial distributions of colour in an image. Texture represents the low-level structures such as graininess and smoothness, while shape represents the shapes that appear in the image as characterised by colour-based segmentation techniques.

Provided that Oracle9i has been installed with the *inter*Media option, it can support the `student` table of Figure 3 - 2 using the following steps:

1. Create the `students` table with the necessary columns.

```
CREATE TABLE students
 (
StudentNumber  NUMBER     PRIMARY KEY    NOT NULL,
StudentName           VARCHAR(50)          NOT NULL,
Course          VARCHAR(50)        NOT NULL,
StudentPhoto          ORDSYS.ORDIMAGE,
PhotoSignature        ORDSYS.ORDIMAGESIGNATURE
);
```

`StudentPhoto` is stored using `OrdImage` data type, which contains basic image attributes such as width, height, source location and the actual image data. `PhotoSignature` is stored under `OrdImageSignature` type, which stores the image signature as a combined feature vector. Since `OrdImage` and `OrdImageSignature` are stored under the `ORDSYS` schema of the database, users from other schemas must write these data types as `ORDSYS.ORDIMAGE` and `ORDSYS.ORDIMAGESIGNATURE` to specify their defining schema.

2. Create a directory that contains image files and grant read rights to users.

```
CREATE DIRECTORY IMAGEDIRECTORY AS 'C:\PHOTOS';
GRANT READ ON DIRECTORY IMAGEDIRECTORY TO PUBLIC;
```

The above statement only works when the specified directory resides in the server system, and must always be in capital letters. If images from external sources such as a web server are used, an http location must be specified. External file systems can alternatively use web forms to specify the file locations using *inter*Media Java classes.

3. Use PL/SQL to insert a record in the table.

```
DECLARE
  image ORDSYS.ORDIMAGE;
  signature ORDSYS.ORDIMAGESIGNATURE;
  ctx RAW(4000):=NULL;
BEGIN

INSERT INTO STUDENTS VALUES
(6011760,'Kandeshi','Doctor',ORDSYS.ORDIMAGE.INIT(),
ORDSYS.ORDIMAGESIGNATURE.INIT());

SELECT Studentphoto, Photosignature
INTO image, signature
FROM STUDENTS
WHERE STUDENTNUMBER = 6011760 FOR UPDATE;

  image.SETSOURCE('FILE','IMAGEDIRECTORY','kauna.gif');
  image.setProperties;
  image.IMPORT(CTX);
  signature.generateSignature(image);
UPDATE Students
SET studentphoto = image, photosignature = signature
            WHERE studentnumber =6011760;
  COMMIT;
END;
```

The first `INSERT` command in the above code initialises the `studentphoto` and the `photosignature` columns with empty values (which is different from the null value, since columns initialised with empty values can be selected for update while null value columns cannot). The `SELECT FOR UPDATE` statement then locks the row for the student with student number *6011760* for update, and sets the image source location using the `SETSOURCE` method of the `OrdImage` object. Before loading the image in the database, its properties are set using the `setproperties` method.

The image is then imported into the database using the `import` method, and the image signature is created using the `generateSignature` method.

- The `generateSignature` method creates the signature of the image using *region-based* signatures. *Region-based* signatures are found by dividing the image into regions based on colour, texture, shape and location. Location is used to describe the exact position of the colour, texture and shape attributes.

The signature returned from `generateSignature` is stored in the `photoSignature` column.

Finally, the `students` table is updated with the new values stored in the `Image` and `Signature` variables.

4. Create an index on the signature to speed up retrieval.

*inter*Media defines an index type called `ORDIMAGEINDEX`, to build and maintain an index for image data. In the following statement, an index `PHOTOINDEX` is created on the `students` table based on the data in the `photosignature` column.

```
CREATE INDEX PHOTOINDEX ON STUDENTS (PHOTOSIGNATURE) INDEXTYPE IS
ORDSYS.ORDIMAGEINDEX
  PARAMETERS ('ORDIMAGE_data_Tablespace=<tabname>,
ORDIMAGE_Index_Tablespace= <intname>');
```

The `tabname` argument in the `PARAMETERS` function is the name of the `tablespace` that will contain the actual index data, while `intname` is the `tablespace` that will contain the internal index created on the data.

5. Perform content-based retrieval.

Assuming that the `students` table has been populated with a number of images, a query for images that looks similar to the student with student number *6011760* can be issued using:

```
SELECT c.studentname, c.studentphoto, ORDSYS.IMGScore(123) SCORE
FROM students c, students q
WHERE q.studentnumber = 6011760 AND
```

```
ORDSYS.IMGSimilar ( c.photosignature, q.photosignature,
'COLOUR="0.2" TEXTURE="0.1" SHAPE ="0.4" LOCATION="0.3"',10,123)=
1;
```

The `IMGSimilar` function, which performs content-based image retrieval, accepts the following five arguments:

- The signature (`c.photosignature`) of the image to compare
- The signature (`q.photosignature`) of query image
- A string with an arbitrary combination of the features weights used in matching. The string assigns the weight to each feature specified feature, which ranges between 0 and 1. Unlike the `RESEMBLES` function which requires all feature weights to be assigned, features that are not important to the search can be ignored in the statement. The total feature weight should add up to 1.
- The threshold parameter, which determines the relevance of images to the query. Images with a similarity score less than or equal to the threshold value are considered relevant to the query, and are returned as matching to the query image. The threshold value ranges from 0 to 100, with a threshold of 0 returning exact matches, while a threshold of 100 returns all images in the database.

- Optional value for auxiliary operator – this value must be the same in the `IMGScore` and `IMGSimilar` operator to indicate that the matching score returned in both operators is the same.

The query statement compares the query image (q) and the comparison image (c) already stored in the database to find the student number, student photo, and the similarity score of students with photos that looks similar to the image of a student with the student number *6011760*. As demonstrated with the `RESEMBLES` function, the similarity measures returned by the matching function can be easily sorted in decreasing order of importance using the ORDER BY operator on the score attribute of the select statement, so that images more similar to the query images appear earlier than less similar images in the matched list. Since a threshold of 10 is specified, only images with a match score less than or equal to 10 are returned.

3.5 Conclusions

The previous section demonstrated how an application requiring content-based retrieval of images might be built in Oracle and Informix. The demonstration used database-specific implementations to give a summarised comparison of the retrieval capabilities offered by the image retrieval extensions in these two databases. Apart from the attribute and method names used by the different databases, the functionality offered by these two database systems is comparable.

Both Informix and Oracle image retrieval extensions are able to store and provide content-based retrieval of images from the database. Content-based searches are performed on a feature vector, which are constructed based on colour, texture and shape information of the image, although Informix has additional features such as brightness and aspect ratio. However, the effectiveness of these retrieval extensions ultimately depends on their ability to identify the required images from the database when required by users. In this regard, the retrieval extensions in these databases are subject to a number of shortcomings:

- Both retrieval systems use low-level features, such as colour and texture, to find similar images, while users usually search for high-level features, such as the presence of a particular object.

- Similarity retrieval is based on general features. No mechanisms exist by which to specify individual characteristics within a particular feature. It is possible, for example, to search for images similar in colour compositions, but a search cannot be made for objects that look like the red object in a given image. Similarly, it is possible to search for images with similar texture, but one cannot specify that the desired similarity be in terms of graininess.

- There is no intuitive set of weights that guarantee satisfactory retrieval. The user has to keep tuning the weights and the retrieval threshold, to achieve different retrieval results.

- There is also no intuitive reason given as to why particular images are returned as a match to the query. Because there is a difference between the perception of the user and that of the system, this makes it difficult to structure queries.

- These retrieval systems are not capable of automatically classifying and recognising individual objects in the images.

The above shortcomings are, however, not unique to image retrieval extensions in Oracle and Informix databases. Research in content-based image retrieval has, in general, mostly concentrated on identifying low-level features that describe the contents of images, allowing users to retrieve images based on features such as colour, texture and shape. Users, however, usually want to retrieve images based on high-level features such as occurrences of specific events, objects and phenomenon, rather than low-level appearances. This weakness, described as the *semantic gap* drawback in [SMC96] [SAJ98] and [SAJ99], is evident in almost all CBIRS available today. The search for high-level features requires complex knowledge and reasoning capabilities, for which current machine technology cannot provide to a satisfactory extent. In fact, [SAJ99] argues that current content-based retrieval features will never represent sufficient degree of similarity for high-level features. As such, the mismatch between the capabilities of the content-based technology and the needs of the users is not likely to be resolved in the near future, exemplifying the need for extensible databases to integrate improved retrieval mechanisms into databases as they become available.

Chapter 4:A sample data cartridge: the type system

This chapter demonstrates how the data cartridge mechanism discussed in chapter 2 can be used to integrate additional retrieval techniques previously unavailable in the database. It uses a simple colour-based image retrieval technique based on the framework set in chapter 3 to explore the cartridge development process and identify the challenges that must be negotiated during the data cartridge development. As seen in the implementation, there are multiple design decisions to make when integrating the technique, partly because the cartridge shown here can be implemented by simply extending the relational model with additional functions. The chapter discusses the implications of using both the cartridge and the relational implementations.

4.1 Introduction

Oracle's image retrieval functionality discussed in chapter 3 requires objects in images to occupy almost the entire image space or at least to occupy the same size and position on each image, to guarantee successful matches [WAR01]. In real life, however, a scenario where the user may desire to retrieve images containing extraneous objects with different sizes and occupying different positions is not uncommon. Even when images are prepared in accordance with the criteria that claim to guarantee successful matches, the user might still want to query the image with new retrieval requirements, such as retrieving images based on specific colour compositions. Chapter 3 revealed that Oracle's image retrieval support does not have provisions for specifying the colour compositions in this manner. As noted in chapter 2, however, extensibility capabilities can be leveraged to integrate new retrieval mechanisms in databases. This chapter makes use of this extensibility to demonstrate how the support required in specifying the colour compositions of the retrieval scenario described above may be achieved.

The prototype data cartridge implemented in this chapter is called the colour-based image retrieval cartridge (CBIRC). In addition to giving the guidelines on how to integrate a content-based image retrieval cartridge in Oracle, the chapter also reports on the implications of the choices made to integrate the cartridge. Even though the chosen retrieval scenario has sufficient data representation to demonstrate the integration of a complex retrieval setting, it exhibits a high degree of simplicity

so that it can easily be implemented as an extension to the relational model. The chapter thus, demonstrates how the implementation of the same scenario can be achieved under the relational model, drawing attention to the relevant relational issues, such as stored procedures, which are used to attain equivalent retrieval. The implementations discussed in this chapter cover the existing development tools and coding tasks available to the data cartridge developer as outlined in [GIE02a] to explore the suitability of the data cartridge in integrating new retrieval techniques.

4.2 Objectives

The aim of CBIRC is to retrieve images with particular colour compositions. In this context, the cartridge is able to answer questions such as "*retrieve all images that contains blue and black colours*" and "*retrieve images that are mostly blue*". The data cartridge can also be used to retrieve images that have similar colours to the query image, and thus have the ability to answer queries such as "*retrieve all images that have the following colour compositions*". When any query of the above nature is issued, the cartridge retrieves all the images that satisfy the specified query condition. The results of the query can then be sorted, so that images are retrieved in a decreasing order of relevance, with the images more relevant to the query appearing earlier in the query answer.

As noted earlier, the objective of building the CBIRC is to demonstrate how the data cartridge construct may be used to integrate additional retrieval techniques in Oracle using existing tools. In this context, the coverage of the implementation discussed in this chapter encompasses the steps available to the data cartridge developer from the development to the deployment process as explained in chapter 2 and the data cartridge developer's guide [GIE02a]. Section 3 of this chapter briefly discusses the architecture of the cartridge using the retrieval framework set in chapter 3. In section 4, a step-by-step implementation of the CBIRC is given. Section 5 demonstrates how equivalent functionality to CBIRC capabilities can be achieved under the relational model. Finally, section 6 describes the issues that must be addressed during data cartridge development and concludes with an evaluation of the data cartridge development process.

4.3 The data cartridge development process

As explained in chapter 2, the data cartridge mechanism allows the database to be extended in four key areas: the type system, the server execution environment, the indexing structures and the query optimiser. In general, it is not necessary to extend all these services in a data cartridge; the optimal approach to developing and assembling the cartridge components depends on the particular needs of the application. This section describes the main tasks that are necessary to develop data cartridges. The entire cartridge development process is depicted in Figure 4 – 1.

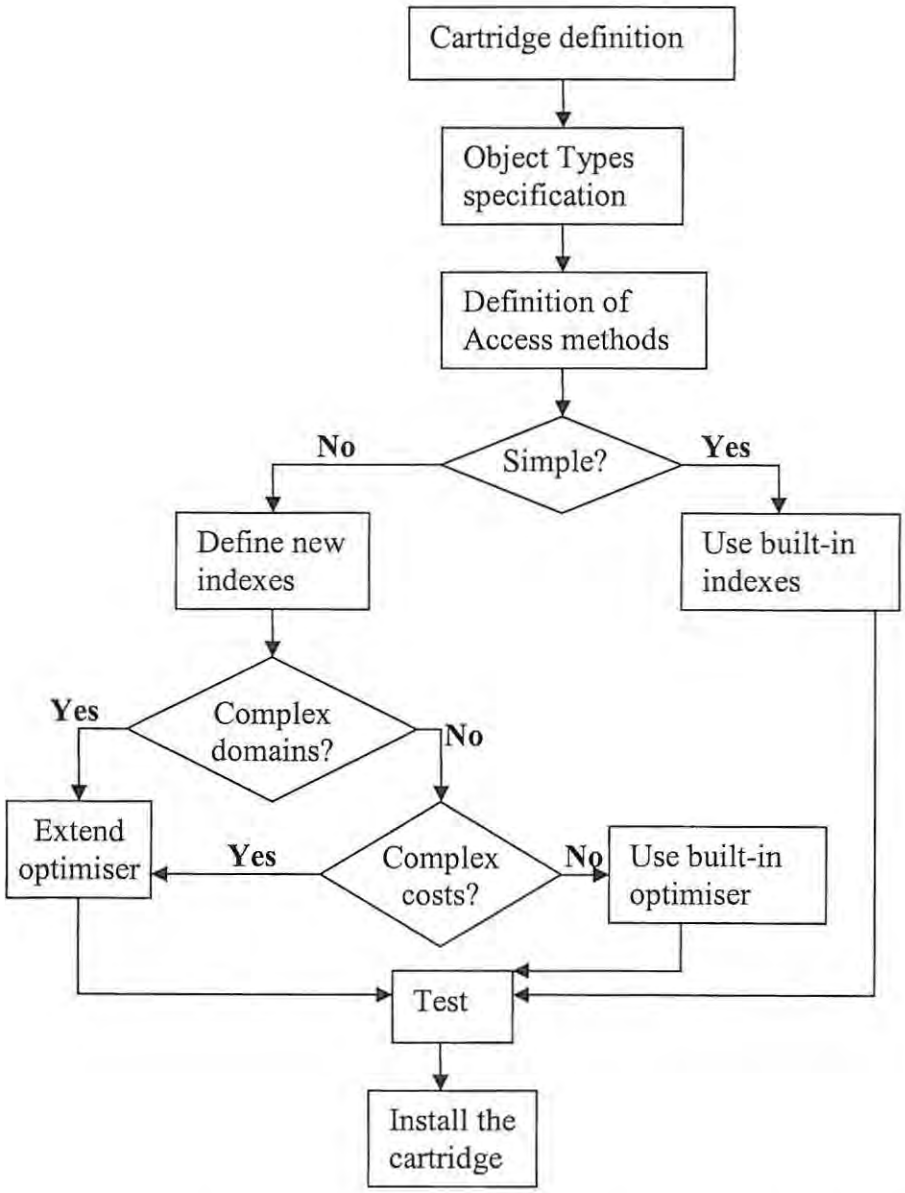

Figure 4 - 1 : The data cartridge development process – (*Adapted from Figure 2 - 1 [GIE02]*)

The first step to building a data cartridge is the cartridge task definition, which requires the new features that the new cartridge will provide for end users to be specified. Once the cartridge specification is defined, the object types and their methods can then be described and implemented in the database using the allowed database languages such as Java and C. Since methods written in C/C++ are executed in an external address space, their code must first be packaged into a DLL, and an interface that defines the interaction between the SQL statements and these methods must be defined. This also involves specifying the path and the file name of the DLL, along with the function name that will be used as an alias to calling the library. Since using C/C++ functions requires tedious and exhaustive database set up, the simplified set up process is given in Appendix D.

After implementing the objects and their methods, access methods to be used to index the cartridge data can then be defined. Applications with simple data types (such as queries requiring numerical range queries) do not need user-defined access methods, as they can be indexed using built-in indexing techniques. Applications requiring complex indexing mechanisms on the other hand, can have indexing mechanisms defined for them. Finally, the optimisation requirements are determined, with queries requiring complex optimisation techniques using user-defined optimiser and other queries using the built-in optimiser. The test program that uses the cartridge can then be written to debug and test the cartridge for usability and correctness before installing the cartridge.

The cartridge components must be installed in a schema, which has the same name as the cartridge before they can be used from the database. Like other schema objects, cartridge components are only accessible by the owner and by users to whom specific access privileges have been granted. Cartridge components can also be declared as `global`, which makes them visible to all database users. New error names and codes can also be defined for use in data cartridges. Oracle has reserved the error codes in the range 20000 to 20999 to cartridge specific error messages, which cartridge developers can use to define unique cartridge errors in the form `ORA20000:xxx`.

4.4 The cartridge definition

This section describes the components of CBIRC. It describes how the image object is represented in the database, how the content is extracted from the image and stored in the database, and describes the similarity metrics adopted to realise the application-specific and the new functional capabilities that the data cartridge intends to provide.

4.4.1 Image object representation

The content of the image in the colour-based image retrieval cartridge is represented as a collection of colour features directly extracted from the image. The representation of the image object in the database is better explained using the following model:

An image database table $I = \{i_1, i_2, i_3, \ldots\ldots i_n\}$ is assumed to store a set of *n* images. When an image $i_j \in I$ is added to the database, its basic attributes $A = (a_1, a_2, a_3 \ldots\ldots a_7)$ (representing the seven attributes such as width, height, etc as shown in Figure 4 - 2) are extracted. The amount of each of the 13 colours (shown in table 4-1) represented in an image is also calculated and stored as a 10-bin histogram $H = (h_1, h_2, h_3 \ldots\ldots h_{10})$, where h_j is the j^{th} bin of the histogram. A complete image object is thus represented in the database as:

```
I = O (D, A, H)
```

where *D* is the actual image data. The image database entities are depicted in Figure 4 - 2.

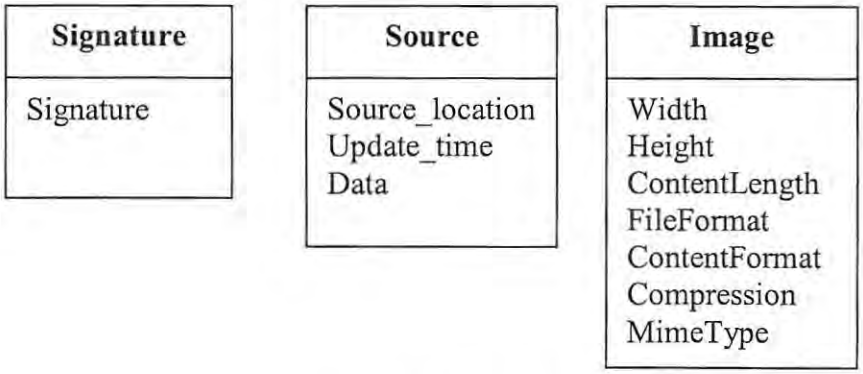

Figure 4 - 2 : Entities of the CBIRC

4.4.2 Image content representation

CBIRC uses colour to describe the contents of images, because colour is easier to extract than other features. Colour is also widely used in retrieval systems because it is invariant to image size and rotation, and partial occlusion [SWB91]. CBIRC uses the standard RGB colour space explained in [SAC+96] to describe the content stored in images. When an image is initially entered in the

database, CBIRC computes the total numbers of pixels that correspond to each of the 13 colours shown in table 4 – 1.

Colour Name	Red	Green	Blue
White	255	255	255
Light grey	192	192	192
Grey	128	128	128
Dark grey	64	64	64
Black	0	0	0
Red	255	0	0
Pink	255	175	175
Orange	255	200	0
Yellow	255	255	0
Green	0	255	0
Magenta	255	0	255
Cyan	0	255	255
Blue	0	0	255

Table 4 - 1 : The default Colour Space as proposed in sRGB

CBIRC only uses 13 colours to avoid using a very high dimensional colour vector. The total number of pixels for each colour are normalised to percentages and stored as the cumulative frequency of the colours in a histogram, which is used to construct a signature to the image. This signature is then stored as a 10-bin histogram, constructed using a Constant-Bin Allocation discussed in [CHIT01].

To explain the signature concept, consider the simple image shown in Figure 4 - 3.

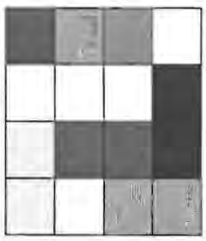

Figure 4 - 3 : Sample image

The cumulative frequencies of colours can be extracted from the image according to the colour descriptions of table 4-1 are shown in table 4-2 below:

Colour	Amount
Red	19%
Blue	13%
White	31%
Yellow	13%
Grey	6%
Green	18%

Table 4 - 2 : Colours found in sample image

Since there are 10 bins in the histogram, each bin holds a 10% capacity of the total colour representation. Bin, B1, for example records the colour percentage from 0 – 10%, while B10 holds 90-100%. The sample image's signature is represented as:

Colour	Amount	Histogram bins									
		B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
Red	19%	0	1	0	0	0	0	0	0	0	0
Blue	13%	0	1	0	0	0	0	0	0	0	0
White	31%	0	0	1	0	0	0	0	0	0	0
Yellow	13%	0	1	0	0	0	0	0	0	0	0
Grey	6%	1	0	0	0	0	0	0	0	0	0
Green	18%	0	1	0	0	0	0	0	0	0	0

Table 4 - 3 : Creation of the signature of the sample image

4.4.3 Similarity metrics

The similarity function between images is based on image signatures rather than the images themselves. The comparison is defined using a distance metric measure between the two signatures (feature vectors). CBIRC determines using the *L1-Distance* measure, which is commonly used for histogram comparison. Similarity between the query image q and the database image i is therefore computed as follows:

$$D_{color}^{(q,i)} = \sum_{j=1}^{10} | set(H_j^q) - set(H_j^i) |,$$

where H_j is the j^{th} bin in the histogram and `set` (H_j) indicates the bin whose value is set to a value whose bin is being compared.

4.5 Implementation of the cartridge components

This section describes how object types are created to meet the retrieval scenario described above.

Since UDTs are arbitrarily defined, it is not possible for the database to provide all the required types. Object types are used to define UDTs, and consist of a specification and a body, which are defined using SQL Data Definition Language (DDL). The specification is the interface to applications: it declares the set of attributes and methods that the applications can call to manipulate the object data. The body implements the specification of the object, and hence, provides the actual definition of the methods [RUS02a]. Methods can be written in PL/SQL or other external languages such as C, Java or C++. The type specification is decoupled from the body, so that the implementation can change without affecting the entire operations of the object. Each cartridge entity shown in Figure 4 - 1 is implemented as an object with relevant data structures and operations, so that the attributes of the `source` and `signature` objects are defined using built-in database types, while the `image` object uses a combination of built-in data types together with the `source` object type. The implementation steps are described below.

1. Designing The Object Type Specification

All the information used by client programs is declared in the object type specification. It declares the data structure and the methods needed to manipulate data. The following statement specifies the `source` object type as `Source_type`:

```
CREATE OR REPLACE TYPE Source_type  AS OBJECT
  (
  sourcePath VARCHAR2(4000),
  data BLOB,
  updateTime DATE,
```

```
STATIC FUNCTION init(name VARCHAR) return Source_type,
STATIC FUNCTION init(content BLOB) return Source_type,

MEMBER FUNCTION getContent return BLOB,
MEMBER FUNCTION getUpdateTime return Date,
MEMBER FUNCTION getSourcePath return VARCHAR2,

MEMBER PROCEDURE setContent (ncontent BLOB),
MEMBER PROCEDURE setUpdateTime(ndate Date),
MEMBER PROCEDURE setSourcePath (npath VARCHAR2)
);
```

The `Source_type` type declared three attributes (`sourcePath, data, updateTime`), two constructor functions, three member functions and three member procedures. The constructor functions, called `init()`, are used to instantiate the source object with a value specifying the source path or a BLOB value. The member functions and member procedures are used as access methods for getting and setting the type attributes respectively.

Similarly, the code fragment for `image` object is given as:

```
CREATE OR REPLACE TYPE Image_type AS OBJECT
  (
   source Source_type,
   width NUMBER,
....
 STATIC FUNCTION Image_init(name VARCHAR2) return Image_type,
 MEMBER FUNCTION getNumber(attr VARCHAR2) return NUMBER,
 MEMBER FUNCTION getString(attrb VARCHAR2) return VARCHAR2,
 MEMBER FUNCTION getSource return Source_type,

 MEMBER PROCEDURE setNumber (nattr VARCHAR2, num NUMBER),
 MEMBER PROCEDURE setString (sattr VARCHAR2, str VARCHAR2),
 MEMBER PROCEDURE  setSource(src VARCHAR2)
 );
```

The `source` attribute of `Image_type` type is of type `Source_type` specified earlier. `Image_init()` is a constructor function for `Image_type` objects. The three member functions are used to return the values of the attribute functions while three member procedures are used to set individual attribute values.

Finally, the `signature` object type, which is used for content-based retrieval in CBIRC, is created as:

```
CREATE OR REPLACE TYPE Signature_type AS OBJECT
  (
    signature BLOB,

  STATIC FUNCTION createSignature(content Source_type) return
Signature_type,
  MEMBER FUNCTION areSimilar (q Signature_type, c Signature_type)
return NUMBER,
  MEMBER FUNCTION findcolourPercentage(q Signature_type, s
VARCHAR2) return NUMBER,
  MEMBER FUNCTION imageContains (q Signature_type,  s VARCHAR2)
return NUMBER
);
```

The signature of the image is stored as a BLOB in the `signature` attribute. `Signature_type` object instances are initialised using the `createSignature()` method, which accepts objects of `Source_type` as parameters. The `areSimilar()` method compares two images for exact similarity, while `imageContains()` method simply determines whether an image contains the colours specified in the `s` parameter. The `findColourPercentage()` returns images with the colour compositions specified in the `s` parameter.

If the `Source_type` object is not declared before the `Signature_type` and `Image_type` are created, declaring these two types will result in an error because they use attributes of type `Source_type`. Nevertheless, these objects can declare and manipulate the object of type `Source_type` without knowing how the `Source_type` represent data or implement its methods. As a consequence, the `Source_type` type methods can be separately changed and implemented without affecting the other object types.

2. Designing The Object Body

The object body implements all the methods defined in the object type specification. Methods in the object specification are declared using the `MEMBER` and `STATIC` keywords. The specification of `signature_type` type above has 4 methods. Notice that the constructor functions are called

`init()`, and do not have the same name as the type name as it is commonly used in class declarations in object-oriented languages. This is because Oracle9i creates a default constructor function that accepts parameters corresponding to the attributes of the each created type, and does not allow user-defined constructor functions to share a name with the type name. An example of a desirable constructor function declaration would be:

```
STATIC FUNCTION Image_type(name VARCHAR2) return Image_type;
```

However, the `STATIC` methods can be invoked on object types, and can thus be used as user-defined constructor functions. `MEMBER` methods on the other hand, can only be invoked on the object instances. In the `Signature_type` object, the function `createSignature()` is used as a constructor function, while the remaining three `MEMBER` functions are used for content-based image retrieval. The code outline for the body of the `signature_type` type is shown below:

```
CREATE OR REPLACE TYPE BODY Signature_type AS
STATIC FUNCTION createSignature(content Source_type) return
Signature_type IS
BEGIN
.....
ncon := IS_COMPUTESIGNATURE(content.getContent());
.....
 END createSignature;

MEMBER FUNCTION areSimilar (q Signature_type, c Signature_type)
return NUMBER IS
.....
   retval := DBMS_LOB.COMPARE(q.signature,c.signature,amt,1,1);
.....
END areSimilar;

MEMBER FUNCTION findcolourPercentage(c Signature_type, s VARCHAR2)
return NUMBER IS
.....
END findcolourPercentage;

MEMBER FUNCTION imageContains (q Signature_type,  s VARCHAR2)
return NUMBER IS
.....
END imageContains;
END;
```

The `createSignature()` method takes an `Image_type` object as a parameter, and extracts the actual image data using the `Image_type.getContent` method. The BLOB value is the passed to a function called `IS_computeSignature`, which extracts the colour information from the image and generates the image signature as a BLOB. The generated signature is then passed as a parameter to construct and return an object of type `Signature_type`.

`AreSimilar()` method takes two `Signature_type` objects and compares them for similarity. It takes `signature` attribute of each `Signature_type` object and uses Oracle's DBMS_LOB package, which is a built-in package called for processing LOBs, to compare the two parameters for similarity. `FindcolourPercentage()` method on the other hand is used to search the database for images that have the same colour compositions as the query image. It takes the signature of the query image and a set of weights specifying the desired colour compositions as arguments, and compares the weights to the signature values to search for the desired combinations.

`ImageContains()` is used to determine whether an image contains specific colours. It takes the signature of the query image as first parameter, and the colours being sought as the second parameter. If the desired colours are present, the function simply returns a Boolean value that indicates whether a certain colour is present or not.

For simplicity, the implementation for `Signature_type` object methods is done Java. Developing Java functions for use in the database involves the following steps:

- **Create the Java Class**

The Java class that contain the functions used in the `Signature_type` is developed as a standalone application external to the database, and has the following code outline:

```
public class ImageSignature
{
public static BLOB computeSignature(BLOB b) throws IOException,
SQLException {}
public static int findCombination(BLOB c, String weights) throws
SQLException, IOException {}
public static int ImageContains(BLOB c, String weights) throws
SQLException, IOException {}
private static BLOB loadData(byte[] s) throws SQLException,
IOException {}}
```

The class is then compiled using the standard java compiler function, `javac`, to create the class file `ImageSignature.class`.

- **Load the Java Class into the database**

Before the Java class can be used in the database, it must first be uploaded into the database schema to make it accessible to the Oracle JVM using the `loadjava` command-line utility. The `loadjava` utility is used to upload Java source, class and resource files into system-generated schema object. The schema object takes a full name of the Java class including the package names. When loaded as source files, `loadjava` can invoke Oracle's JVM compiler to compile the source file, which can be edited and recompiled inside the database schema. When loaded as class files however, modification and recompilations is done outside the database, and the class must be reloaded into the database using the `loadjava` utility to reflect changes in the database. In the following command, `loadjava` loads the file `ImageSignature.class` as a class file into a database called `db`, in the `image_user` schema:

```
loadjava -user image_user/image@db ImageSignature.class
```

- **Publish the Java Class**

The class methods that are directly referenced by Oracle objects are then published in the database data dictionary through SQL call specifications, to make them callable from SQL. Other Java classes that are only referenced by the Java class but do not have methods callable by Oracle objects need not be published. The call specification defines the SQL arguments and return types for each callable Java method. As an example, the `IS_ComputeSignature` function used in `Signature_type` is published from the Java class `ImageSignature` as:

```
CREATE OR REPLACE FUNCTION IS_COMPUTESIGNATURE (name BLOB) RETURN
BLOB AS
LANGUAGE JAVA
NAME 'ImageSignature.computeSignature(oracle.sql.BLOB) return
oracle.sql.BLOB';
```

The functions and specification for the `Image_type` and `Source_type` are also specified in the manner similar to `Signature_type` type. The code outline is shown in Appendix A.1 to A.5.

4.6 Relational model implementation

There are no object types in the relational implementation: all object types are flattened to attributes in tables. One way to implement these three entities is by normalising all their attributes into tables. An `Image_table` table for example, may contain all the attributes of the `Image` entity plus all the attributes of the `Source` entity. Alternatively, each entity can be stored as a table, and linked with foreign key pairs so that the three entities are represented using `Signature_table`, `Source_table` and `Image_Table` as shown in Figure 4 - 4.

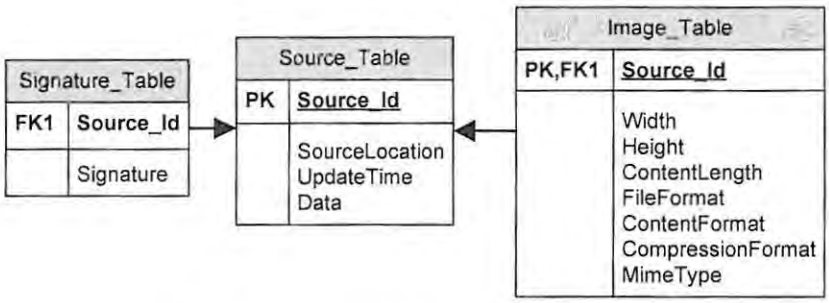

Figure 4 - 4 : Entity representation in the relational model

The appropriate methods are then created and stored as Java classes, which are also loaded into the database as stored procedures, using the same `loadjava` utility. Assuming the same function names as in the cartridge implementation, functions such `createSignature`, `areSimilar` and `findcolourPercentage` are then created from the Java classes using the `CREATE FUNCTION` syntax. These functions now take different parameters since the relational implementation does not allow object types to be defined, and thus object types cannot be passed as parameters. As an example, the new `createSignature` function is created as:

```
CREATE OR REPLACE FUNCTION createSignature (name BLOB) RETURN BLOB
AS
LANGUAGE JAVA
NAME 'ImageSignature.computeSignature(oracle.sql.BLOB) return
oracle.sql.BLOB';
/
```

An example of using this signature requires data to be inserted into the `source_table` first such as:

```
 INSERT INTO Source_table (source_id, source_location, updateTime,
data)
 VALUES  (6011760,'C:\imagedir','03/03/03', EMPTY_BLOB());
/
```

The `data` attribute of the `source_table` is then loaded with data from an operating system file, called a BFILE in Oracle, as:

```
DECLARE
Src  BFILE := BFILENAME('IMAGEDIRECTORY','kauna.gif');
Des BLOB;
Amt INTEGER := 4000;
BEGIN
SELECT data INTO des FROM source_table WHERE id = 6011760 FOR
UPDATE;
DBMS_LOB.OPEN(Src,DBMS_LOB.LOB_READONLY);
DBMS_LOB.LOADFROMFILE(Des,Src,Amt);
DBMS_LOB.CLOSE(Src);
UPDATE source_table SET DATA = DES WHERE id = 6011760;
Commit;
END;
```

A row can then be inserted in the signature table using:

```
 INSERT INTO Signature_table VALUES  (6011760 , EMPTY_BLOB());
```

Finally, the `createSignature` function can then be used to create and update the signature of the image as:

```
DECLARE
   Des BLOB;
   temp BLOB;
BEGIN
SELECT data INTO des FROM source_table WHERE id = 6011760;
SELECT signature INTO temp FROM signature_table WHERE id = 6011760
 FOR UPDATE;
temp := createSignature(des);
```

```
UPDATE signature_table SET signature = temp WHERE id = 6011760;
Commit;
END;
```

Assuming the function `areSimilar` has also been created using the following code outline,

```
CREATE or replace FUNCTION areSimilar (obj BLOB, obj2 BLOB)
RETURN NUMBER AS
.....
END;,
```

two images are compared for similarity using:

```
SELECT c.student_name
FROM student_table c, signature_table s, signature_table t
WHERE  c.photo_id <> t.source_id AND
areSimilar(s.signature,s.signature)=1;
```

A collection of functions and stored procedures can be stored together in Oracle using *packages*. A package treats functions, stored procedures, cursors and the variables they use as a unit. While standalone functions cannot share the same name [RUS02b], packages allow functions to be overloaded. Packages also make the program components more manageable and maintainable because they are organised and privileges can be granted more efficiently. However, packages cannot inherit from each other, thereby limiting their potential usefulness in supporting objects.

4.7 The data cartridge vs. the relational implementation

As seen in the implementation, integrating additional functionality in the database presents a number of implementation choices. The same application can be developed and integrated in multiple ways and still achieve the same functionality. The example in this chapter demonstrated the data cartridge implementation and extension to the relational approach. This provides a basis for comparing the object-relational technology against the required extensibility functionality discussed in chapter 2.

The relational implementation is not difficult to understand; it is straightforward and only requires the usual mapping of entity elements to table attributes. Tables must be in the first normal form, with every attribute limited to a simple atomic data type. This, however, offers limited flexibility to applications that need complex data structures, as data types had to be flattened to fit into tables. The relational approach is thus unsuitable for large applications and applications that require complex object support. The cartridge implementation on the other hand allows multitudes of data types and their methods to be combined to describe complex types. A table that uses these complex types can contain relations that contain nested collections, thereby allowing objects with rich internal structure to be stored in the database.

Tables in the relational implementation are linked by foreign key pairs to maintain relationships among entities. Query statements involving multiple tables also require complex joins to process, and thus becomes difficult to understand and slow to process, because the query planner has to optimise complex joins. This also affects the performance of the query processor because there are too many clauses to check. The use of object types in data cartridges on the other hand eliminates the need for numerous joins and allows fast access paths to be defined for queries with the quantification utilising these data types and their operators.

Another significant difference between the relational and data cartridge implementation is the ability to define object methods in the cartridge implementation. As mentioned in chapter 2, this feature is fundamental to the extensibility of a database system as it allows new functionality to be added to the database. The relational implementation allows methods to be added to the database with stored procedures and functions, which can be grouped together as packages for maintainability and code reuse. In comparison to object methods, packages are stored as static pieces of code, while object types and their methods are stored as templates for individual objects to which instances can be applied. In this regard, packages do not make complex data more manageable, and thus do not offer sufficient support for the retrieval model. While these packages offer some form of encapsulation, they fail to adequately support inheritance, and hence do not promote the necessary productivity offered by the cartridge model. Packages also do not support all kinds of schema elements such as the definition of tables and the use of triggers. It is also extremely difficult to model relationships between different packages.

The data cartridge approach on the other hand, divides the cartridge into a collection of services that carry out different functions, thereby defining the components that can be extended during the data cartridge design. The idea is to allow different services to be added and modified without affecting

other components, and hence allow the database functionality to be extended in a flexible way. As seen in this chapter, it is possible to only extend and use the type service without implementing the other services of the data cartridge. Similarly, it is possible to add new features without affecting the existing functional operations of the type system. However, this approach lies in the assumption that each service is developed independent of other services.

Summarising the findings, it is evident that integrating domain specific applications and the relationships among them cannot be sufficiently captured in the relational model for complex applications. Data cartridges were found to be more suitable for integrating these applications.

4.8 Conclusions

This chapter used an implementation of the colour-based image retrieval technique to demonstrate how Oracle database can be extended with additional retrieval techniques. Prototype implementations of the CBIRC using the cartridge construct and the relational model demonstrated that similar functionality can be achieved with both implementations, and also allowed the appropriateness and implications of using each approach to be evaluated under a common framework. The relational implementation does not require the user to learn concepts about designing data cartridges and thus appears to be simpler to understand and implement, especially for simple applications. The advantage of using data cartridges is that, once the cartridge is constructed, there is little extra complexity in trying to harmonise the workings of the cartridge. The integration of CBIRC has proven that data cartridges are a suitable way for integrating image retrieval techniques in the database.

Chapter 5:A sample data cartridge: indexing and query optimisation

Indexing and query optimisation techniques are some of the important tools available in a DBMS to enhance efficiency. This chapter resumes the integration of CBIRC discussed in the previous chapter to explore the indexing and query optimisation extensibility services of the data cartridge mechanism. It uses the query rewriting approach to demonstrate how user-defined data types and predicates may be supported without modifications to the underlying indexing structures and database engine. The integration revealed that extending new data types with existing indexing methods greatly simplifies the development of both the index and query optimisation techniques, and can even provide good performance. The chapter is then concluded with the implications of using this approach.

5.1 Introduction

Extending the data cartridge with new object types as described in the previous chapter makes it easier to model complex entities and facilitates the reusability of code to make data cartridges easier to understand and maintain. Unfortunately, it does nothing to address how these types should be organised so that queries can be resolved efficiently and relevant portions of data extracted quickly during query processing and resolution. Using that implementation, the functions to compare images for similarity are evaluated row by row, and the query optimiser has to perform a full table scan in the evaluation plan to ensure that all relevant rows are retrieved. This results in poor performance especially if only a small subset of the records is to be retrieved from the database.

As a solution to the situation described above, DBMS provide indexing techniques to aid in evaluating query predicates with index-based lookups. These indexing techniques, can, however, only be used on table columns whose data types and query predicates are understood by the database. Since Oracle provides limited indexing structures, the cartridge developer must define indexing structures for unsupported data types. Different applications have different retrieval requirements and thus require that information be indexed in different ways. It is therefore not possible to define a single technique that is optimal for all applications. For that, user-defined index

structures that encompass unique application requirements are necessary. This chapter looks at the impetus and implications of using user-defined indexes and their query optimisation techniques.

The index at the end of a book is a well-known example used to explain what an index is. Using this index, it is possible to find the pages with the relevant information about a specified concept by looking it up in the index, without searching entire book. Provided the index is clearly defined, this does not necessitate comprehension of the contents of the index by the user, as long as the different index tokens can be identified from each other and a location that maps to place where the required information is stored is provided. Although this relaxed book-indexing scenario does not necessarily apply to complex application scenarios such as image retrieval, it indeed captures the ideal, required situation since image signatures are also stored in a way that cannot be easily comprehended by all index users.

5.2 Objectives

Oracle neither permits indexing of columns containing LOB values, which are used to store large data such as multimedia data, nor does it index attributes of column objects or elements of a collection type. Indexing values derived from user-defined methods and operations is also not supported in Oracle [GIE02a] . This implies that the provided index structures do not suffice for image retrieval scenarios, where a query may require to compare the signatures of images derived from signature computation operations, and may also be stored in BLOB columns. In addition, efficient query execution cannot be achieved because query optimisers are not able to calculate the cost of user-defined query predicates.

Oracle provides generic 'template' indexing and query optimisation interfaces that allow new indexing structures to be defined arbitrarily by the user, and to be associated with user-defined object types. These interfaces enable the query compilers to recognise user-defined query predicates and to know how the index can be searched to fully exploit user-defined indexing structures and thereby facilitate efficient query execution. The aim of this chapter is to explore the implications of extending the indexing and query optimisation structures of a data cartridge, and to highlight the challenges that must be resolved during these extensions.

Designing indexing structures for use in an ORDBMS requires a good understanding of concurrency and recovery protocols, and the actual integration requires the access method to

implement low-level database functionality including the lock manger and buffer page management protocols [KOR99]. Data cartridge developers are often not database server experts, and therefore possess limited knowledge of the internals of the database engine and thus cannot efficiently write these methods. Giving such power to less knowledgeable users is also very risky, as it could lead to malicious code to be run from the database server. As a simplification of this situation, this chapter uses an approach that extends the indexing capabilities of the database through *query rewriting*. *Query rewriting* transforms user-defined data types and query predicates that are not directly supported by the underlying database indexing structures to be rewritten so that their operators can be supported by built-in indexes. This requires mapping user-defined operators to the known database operators to make them recognisable. The chapter describes how the operators of CBIRC may be enabled to use the well-known B-tree indexing structure.

5.3 Building the index

As described in the previous chapter, CBIRC extracts the colour feature from each image and stores it in a form of a vector that captures the content in histogram bins. Although this simplification has greatly reduced the number of dimensions for the signature, it is still not trivial to define the actual index structure in the database, as each index structure must extend auxiliary data structure for the extensible indexing interface to implement the extensible user-defined indexing. The interface allows the user to define the structure of the index and to specify how and where the index data will be stored. It also allows the developer to define how the application manages, retrieves and uses the index data during query processing. To summarise, it defines the following three characteristics:

Content Identifier – to identify individual entries in the index set
Location – to indicate where the information with the specified identity may be located
Storage - to store both the identity and location information of the index

Since query comparisons in CBIRC are performed using image signatures, the index structure is build on the `signature` attribute. According to table 4-3, an image can only contain a total of 13 colours. A simple way of identifying entries in the index is therefore to take each of the 13 colours that appears in the image to be an entry in the index. As such, the index only contains the image identifiers and its set of 13 terms and along with their corresponding colour values, which are stored as histogram bins. The steps to implement the index are given below.

1. Designing the index operators and functions

CBIRC allows users to issue queries for equality, contains and greater than operations. Because of the way that the image signature is stored, these operators cannot be directly applied to the signature data. These operators must therefore be implemented to meet the requirements of the cartridge. An operator consists of the operator name and the functions that implement the operations on the data type. When using the relational implementation, the operator function is the implementation that is used when there is no index defined on the data. The names for the operators and their corresponding functions in CBIRC are shown in table 5 – 1.

Operator Name	Operator Function
Op_ColourIsPresent	Func_IsColourPresent
Op_ColourIsGreaterThanValue	Func_IsColourGreaterThanValue
Op_ImageIsExact	Func_IsImageExact

Table 5 - 1 : Operators and implementing functions

The `Op_ColourIsPresent()` operator is used to check if a specific colour appears in an image, while `Op_ColourIsGreaterThanValue()` operator determines whether the specified colour is greater than a specified value. `Op_ColourIsExact()` operator checks whether two images contain the exact amount of colour compositions. All implementing functions return a numeric value of 1 if the condition is true otherwise they return 0.

The functional implementation for the operator has to be written and compiled using any of the languages allowed in the database. The function must then be registered or published in the database as a user-defined function as explained in section 4.2. As an example, the following shows the code outline for the implementation of the `Func_IsImageExact` function, which takes two `signature_type` objects and compares them for similarity:

```
CREATE or replace FUNCTION FUNC_IsImageExact(obj Signature_type,
obj2 Signature_type)
RETURN NUMBER AS
   amount INTEGER := 130;
   retval NUMBER;
BEGIN
```

```
    /*compare the values*/
    retval :=
    DBMS_LOB.COMPARE(obj.signature,obj2.signature,amount,1,1);
          IF retval = 0 THEN
                RETURN 1;
          ELSE
                RETURN 0;
          END IF;
END;
```

Since the image signature is stored as a BLOB in the `signature` column, the function gets the `signature` column of the `signature_type` type, and uses the `compare()` function of the DBMS_LOB package provided by Oracle to compare the two signatures. The return value is then set to the appropriate Boolean value, because the `compare()` function returns a value of 0 when two LOBs are equal.

Each operator is then be bound to the corresponding function such as:

```
CREATE OPERATOR OP_IsImageExact BINDING(Signature_type,
Signature_type) RETURN NUMBER USING FUNC_IsImageExact;
```

Similarly, the other two operators are bound to their corresponding functions.

2. Implementing index routines

The SQL-based extensible indexing interface provided by Oracle allows domain specific operators and indexing techniques to be integrated with the server. The indexing interface has an `IndexType` component that specifies all the required functionality to manage the user-defined indexing routines such as how the index is stored, how it is maintained and how it is scanned during query operations. A user-defined object that extends the indexing interface must therefore implement the methods of the `ODCIIndex` interface, which contains the code for the `IndexType` component. These methods, which include `IndexCreate`, `IndexInsert`, `IndexDrop`, `IndexStart`, `IndexFetch`, `IndexClose`, `IndexAlter` and `IndexUpdate`, contains methods for defining, manipulating, scanning and exporting the index and its data, and are later automatically invoked by the database when executing the relevant SQL commands involving the `IndexType`. The `IndexType` object CBIRC is implemented by the object `IMGIDXMethods`, and is created as:

```
CREATE OR REPLACE TYPE IMGIDXMethods   AUTHID CURRENT_USER
   AS OBJECT
   (

   curnum NUMBER,

   STATIC FUNCTION ODCIGetInterfaces(ifclist OUT
SYS.ODCIOBJECTLIST)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexCreate (ia    SYS.ODCIINDEXINFO,
                        parms VARCHAR2,
                        env   SYS.ODCIENV)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexDrop (ia  SYS.ODCIINDEXINFO,
                       env SYS.ODCIEnv)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexInsert(ia        SYS.ODCIINDEXINFO,
                        rid       VARCHAR2,
                        newsig IN Signature_type,
                        env       SYS.ODCIEnv)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexUpdate(ia        SYS.ODCIINDEXINFO,
                        rid       VARCHAR2,
                        oldsig IN Signature_type,
                        newsig IN Signature_type,
                        env       SYS.ODCIEnv)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexDelete(ia        SYS.ODCIINDEXINFO,
                        rid       VARCHAR2,
                                          oldsig IN
Signature_type,
                        env       SYS.ODCIEnv)
   RETURN NUMBER,

   STATIC FUNCTION ODCIIndexStart(sctx IN OUT IMGIDXMethods,
                       ia SYS.ODCIINDEXINFO,
                       pi SYS.ODCIPREDINFO,
                       qi SYS.ODCIQUERYINFO,
```

```
                    strt NUMBER,
                    stop NUMBER,
                    querysig     IN IMGIDXMethods,
                    weightstring IN VARCHAR2,
                    env              SYS.ODCIEnv)
  RETURN NUMBER,


  MEMBER FUNCTION ODCIIndexFetch(nrows IN NUMBER,
         rids OUT SYS.ODCIRIDLIST,
         env      SYS.ODCIEnv)
  RETURN NUMBER,

  MEMBER FUNCTION ODCIIndexClose(env SYS.ODCIEnv)
  RETURN NUMBER
  );
```

Since `IMGIDXMethods` is defined as an object type, its member functions must be implemented inside the `CREATE TYPE BODY` statement as shown in section 4.2. The first member function, `ODCIGetInterfaces`, is used to return the list of all the interfaces implemented by the `IMGIDXMethods` type. The implementation of this function is given as:

```
STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
      return number is
  BEGIN
      ifclist :=
sys.ODCIObjectList(sys.ODCIObject('SYS','ODCIINDEX2'));
      return ODCIConst.Success;
  END ODCIGetInterfaces;,
```

where the returned `'SYS','ODCIINDEX2'` specifies the Oracle9i version of the `ODCIINDEX` interfaces. The server invokes this function when an index with the type `IMGIDXMethods` is created or altered using the `CREATE INDEXTYPE` statement.

The second method in the `IMGIDXMethods` object specification is `ODCIINDEXCREATE`, which is called when a `CREATE INDEX` statement is issued to create an index of type `IMGIDXMethods`. The function creates objects to store and generate index data, and to store index data in index data tables or files. Calling this function builds the index for the existing data in the indexed columns when the

table for which the index is created (also called a base table) is not empty, or simply creates an empty table where the index data will be stored when the base table is empty.

The ODCIINDEXCREATE function takes SYS.ODCIINDEXINFO and SYS.ODCIENV object types along with a user-defined VARCHAR2 as parameters. The ODCIINDEXINFO object contains information about the indexed column, such as the index schema and index name, while the ODCIENV object contains the information about the environment handle passed to the routine. The VARCHAR2 parameter specifies the information needed to create the index table, and must be interpreted by the user.

The actual index table need only to consist of the row identifier of the signature in the base table, the colours present in the image and their corresponding colour value. These values are directly read and populated from the signature created using the signature_type above. Recognising that each image contains 13 colours and 13 values at the maximum, an array type used to store 13 numbers is created as:

```
CREATE TYPE COLOURGRID_TYP AS VARRAY(13) OF NUMBER;
```

The code outline for ODCIINDEXCREATE function is then given as:

```
 STATIC FUNCTION ODCIIndexCreate (ia sys.odciindexinfo, parms
VARCHAR2, env sys.ODCIEnv)
RETURN NUMBER
 IS
   ..........
/* Construct the SQL statement for creating the index table*/

   stmt := 'CREATE TABLE ' ||   ia.IndexSchema || '.' ||
ia.IndexName || '_imidx' ||
           '( rid ROWID, colcode COLOURGRID_TYP, colval
COLOURGRID_TYP)';
..........
END;
```

where _imidx is the suffix to the name used for the table holding the index data.

As an example, assuming the colours in table 4-1 are numbered in ascending order from 01 to 13 and the image in Figure 4 - 3 has the number 1 as its row identifier, the signature is represented in

the index table as:

```
(1,colourgrid_typ(06,13,01,09,03,10),colourgrid_typ(19,13,31,13,6,1
8))
```

The ODCIINDEXDROP function drops the tables storing the domain index data when invoked using the DROP INDEX statement. This function takes the ODCIINDEXINFO object containing the information about the indexed column, and the ODCIENV object containing the environment handle passed to the routine as parameters. The code outline for dropping the table with the suffix _imidx is:

```
  STATIC FUNCTION ODCIIndexDrop(ia sys.odciindexinfo, env
sys.ODCIEnv)
    RETURN NUMBER is
   ……..
    / * Construct the SQL statement for dropping the table. */
  stmt := 'drop table ' || ia.IndexSchema || '.' || ia.IndexName
|| '_imidx';
……..
   RETURN ODCICONST.SUCCESS;
  END;
```

If the table was successfully dropped, the function returns ODCICONST.SUCCESS.

The ODCIINDEXINSERT function is used to insert new index data in the index table or file. The function is automatically invoked when a new row is inserted in any table that has the index defined on it. In addition to the information about the index and the indexed columns stored in an object of type SYS.ODCIINDEXINFO, the function also takes the row identifier of the new row in the base table, the new signature to be indexed, and the environment handle passed to the routine as parameters respectively. The code outline for the function is given as:

```
STATIC FUNCTION ODCIIndexInsert(ia  SYS.ODCIINDEXINFO, rid
VARCHAR2,  newsig IN Signature_type,  env  SYS.ODCIEnv)   RETURN
NUMBER IS
……..
     /* Construct the statement. */
```

```
    stmt := ' INSERT INTO ' || ia.IndexSchema || '.' ||
ia.IndexName || '_imidx ' ||
            ' VALUES (:rr, : colcode, :colval)';
    ..........
    RETURN ODCICONST.SUCCESS;
  END ODCIIndexInsert;
```

The ODCIINDEXINSERT function also returns ODCICONST.SUCCESS if the row was successfully inserted in the index table.

The ODCIINDEXUPDATE is invoked when a row in the base table has been updated with new values, to update the index data for the updated row with the new values. In addition to the information about the index and index columns, and the environment handle passed to the routine, the function also contains information about the row identifier of the updated row and the old and new signature values that must be updated. The code outline for this function is given as:

```
STATIC FUNCTION ODCIIndexUpdate(ia  SYS.ODCIINDEXINFO,   rid
VARCHAR2, oldsig IN Signature_type,  newsig IN Signature_type,
env       SYS.ODCIEnv)
RETURN NUMBER IS
....
/*  Delete old entries.  */
    stmt := ' DELETE FROM ' || ia.IndexSchema || '.' ||
ia.IndexName || '_imidx ' ||
            ' WHERE r=:rr';
..............

/* Insert new entries. */

    stmt2 := ' INSERT INTO ' ||   ia.IndexSchema || '.' ||
ia.IndexName || '_pidx' ||
            ' VALUES (:rr, : colcode, :colval)';
 ......

        RETURN ODCICONST.SUCCESS;
  END ODCIIndexInsert;
```

The ODCIIndexUpdate function returns ODCICONST.SUCCESS if the row was successfully updated.

The ODCIINDEXDELETE function deletes the index data from the index table or file when a row is deleted from the base table. The function requires the base table information, and the row identifier for the deleted row, the value for the deleted row and the environment handle passed to the routine. The code outline for the function is given as:

```
STATIC FUNCTION ODCIIndexDelete(ia SYS.ODCIINDEXINFO, rid
VARCHAR2,  oldsig IN Signature_type,  env SYS.ODCIEnv) RETURN
NUMBER IS
.....
   -- Construct the statement.
     stmt := ' DELETE FROM ' || ia.IndexSchema || '.' ||
ia.IndexName || '_imidx ' ||  ' WHERE r=:rr';

  .....
   RETURN ODCICONST.SUCCESS;
   END ODCIIndexDelete;
```

The ODCIINDEXSTART function is used scan the index for all the rows that satisfy operator functions defined in the previous step. The code outline for this function is given as:

```
STATIC FUNCTION ODCIIndexStart(sctx IN OUT IMGIDXMethods, ia
SYS.ODCIINDEXINFO,  pi SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO, strt
NUMBER,  stop NUMBER,  weightstring IN VARCHAR2,  env
SYS.ODCIEnv)
RETURN NUMBER IS
.....
  END;
```

The ODCIINDEXSTART function is invoked when a query involving the index operator that can be executed using the index is issued. As an example, a query that asks for all images that contain the colour white causes the function to scan the second column of the index table to see if it contains the colour code 01 and return its value. The first argument in the function is a scan context, which is used to maintain context between different query calls. The second argument contains the index information, while the third argument contains information about the operator predicate. The qi parameter contains information about the query. The fifth and sixth arguments specify the start and

stop values of the upper and lower bounds on the operator return values respectively, and thus always have the same type as the return type of the operator. `weightstring` specifies the colour specified in the query that require the operator invocation. The last argument is for the environment handle passed to the routine.

The next function is `ODCIIndexFetch`, which is continually invoked until all the rows that satisfy the operator predicate have been retrieved. The last function, `ODCIIndexClose`, is used to round up the processing of the index scan operations, and is usually invoked at the end of a processing function. The code for this function is given as:

```
MEMBER FUNCTION ODCIIndexClose(env SYS.ODCIEnv)
RETURN NUMBER IS
 cnum INTEGER;
BEGIN
  cnum := self.curnum;
  dbms_sql.close_cursor(cnum);
  RETURN ODCICONST.SUCCESS;
END;
```

Other `indextype` functions not implemented above are shown in table 5-2.

Method	Description
ODCIIndexAlter	Modifies, rebuilds, renames and reorganises the index
ODCIIndexExchangePartition	Converts between partitioned and non-partitioned index
ODCIIndexMergePartition	Merges data from merged partitions into a single table
ODCIIndexSplitPartition	Splits data from split partitions into a different tables
ODCIIndexTruncate	Truncates the index

Table 5 - 2 : Other INDEXTYPE methods

5.4 Query optimisation

There are often many ways to process an SQL statement. The goal of a query optimiser is to find the most efficient way to execute a given query. This requires the optimiser to be able to enumerate all the possible plans for evaluating query expressions and to estimate the cost of each enumerated plan so that the plan with the least estimated cost can be chosen for efficiency. In order for the query processor to provide this requirement therefore, it must know about all the possible query operations and how to use them appropriately. Allowing new indexing and query processing

techniques to be defined for user-defined data types increases the possible query operations and thus more evaluation options are introduced, thereby widening the choices available to the query optimiser. Finding a good plan for evaluating a query therefore poses a significant challenge to the cartridge developer. This section discusses issues relevant to exposing user-defined indexes to the optimiser and method selection optimisation as explained in [LUS02].

To optimise a query, the Oracle query optimiser considers many factors related to the objects referenced and the conditions specified in the query. Queries can then be either optimised for best throughput, whereby the path with the least amount of resources necessary to process all rows accessed by the statement is chosen, or for best response time, which chooses the path that uses the least amount of resources to process the first row required by the SQL statement. User-defined query optimisation techniques extend the Cost-Based Optimiser (CBO) using the DBMS_STATS package and the ANALYSE command provided by Oracle.

The DBMS_STATS package is used to collect and invoke standard statistics on the metadata information stored in the system data dictionary. This dictionary, which contains information about each relation, index and view in the database, normally also contains statistics about relations and indexes. The extensible optimiser can thus use this information to create and store statistics collections, selectivity and cost functions for user-defined indexes and columns. This involves defining representations for the statistics and their maintenance and implementing the functionality for the selectivity and cost functions and, depending on the required functionality, implementing some or all of the methods from the Oracle Data Cartridge Interface Statistics (ODCIStats) interface. In CBIRC, the extensible optimiser is implemented using the following steps:

1. Creating the statistics table

The statistics about the colour distributions in individual image signatures in CBIRC are collected and stored in a table `ColorStatsTable`. The table contains the following columns:

- The table and column for which the statistics are collected
- The colour for which statistics are collected
- The maximum value for the each colour over all images
- The total number of images containing each specific colour

For simplicity, the statistics and optimiser routines described in this section are collected for user-defined statistics on an index only, and does not describe the statistics collected on individual tables.

2. Creating the optimiser methods

The optimiser methods extend the `ODCIStats` interface to specify the methods that the extensible optimiser uses to efficiently execute queries. The database server automatically calls these methods when an implementation of the interface involving a CBIRC data type is registered with the server. The methods are created using an object type called `ColorStatistics`, which will be called when an `ANALYSE` command is issued used to collect and delete statistics or to evaluate the best execution plan for a query. This object type is defined below.

```
CREATE OR REPLACE TYPE ColorStatistics AUTHID CURRENT_USER
   AS OBJECT
   (

     STATIC FUNCTION ODCIGetInterfaces(ifclist OUT
SYS.ODCIOBJECTLIST)
     RETURN NUMBER,

     STATIC FUNCTION ODCIStatsCollect(ia      SYS.ODCIIndexInfo,
options SYS.ODCIStatsOptions,  stats   OUT RAW,  env
SYS.ODCIEnv)
     RETURN NUMBER,

     STATIC FUNCTION ODCIStatsDelete(ia       SYS.ODCIIndexInfo,
stats    OUT RAW,  env      SYS.ODCIEnv)
     RETURN NUMBER,

     STATIC FUNCTION ODCIStatsSelectivity(pred  SYS.ODCIPredInfo,
sel OUT NUMBER, args SYS.ODCIARGDESCLIST, strt NUMBER,  stop
NUMBER, sigcol     IMGIDXMethods, qsig IMGIDXMethods, weighstring
VARCHAR2  env        SYS.ODCIEnv)
     RETURN NUMBER,

     PRAGMA RESTRICT_REFERENCES(ODCIStatsSelectivity, WNDS, WNPS)

);
```

The corresponding implementation of the object body is done using the `CREATE TYPE BODY` syntax. The first function `ODCIGetInterfaces` is implemented as:

```
STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
   RETURN NUMBER IS
BEGIN
   ifclist :=
sys.ODCIObjectList(sys.ODCIObject('SYS','ODCISTATS2'));
   RETURN ODCIConst.Success;
END ODCIGetInterfaces;
```

The server to determine the type of interfaces implemented for user-defined statistics type by invoking this function. The returned parameter `'SYS','ODCISTATS2'` specifies the Oracle9i version of the `ODCIStats` interfaces.

The second function collects statistics for indexes of type `IMGIDXMethods`. This function analyses the domain index by simply analysing the table that implements the index. This function is implemented as:

```
STATIC FUNCTION ODCIStatsCollect (ia sys.ODCIIndexInfo, options
sys.ODCIStatsOptions, rawstats OUT RAW, env sys.ODCIEnv)
RETURN NUMBER IS
.....
   stmt := 'ANALYSE TABLE '  || ia.IndexSchema || '.' ||
ia.IndexName || '_imidx'
        || ' COMPUTE STATISTICS';
/* execute the statement */
.....
END;
```

The function is used to collect information on user-defined statistics on an index. The first parameter specifies information for the index for which information is being collected, while the options parameter lists the options passed to the `ANALYSE` statement. The returned parameter `STATISTICS`, returns the collected user-defined parameters.

The next function deletes the statistics for the domain index by deleting the statistics table implementing the index.

```
STATIC FUNCTION ODCIStatsDelete(ia sys.ODCIIndexInfo, statistics
OUT RAW, env sys.ODCIEnv)
RETURN NUMBER IS
    .....
   stmt := 'ANALYSE TABLE '  || ia.IndexSchema || '.' ||
ia.IndexName || '_imidx'
        || ' DELETE STATISTICS';
.....
  END;
```

The first parameter specifies information for the index for which information is being collected, while the `statistics` parameter contains the aggregate statistics for the index. The `env` parameter in this case is used to obtain information about the number of times the server has called the delete statistics function.

The last function in the type is used to estimate the selectivity of implemented operators and functions. If a query asks, for example, for all images that are mostly blue, the selectivity of the predicate function estimates the fraction of rows of the table that satisfies this predicate, and returning it as a percentage of rows. In our case, the selectivity may be computed when the predicate is of the form:

```
Op(<signature>,<colourcode>, <colourvalue>)
<operator><comparevalue>,
```

Where the function `Op` returns a 0, 1, or NULL. If the predicate does not meet the required form, the optimiser makes a guess or simply returns an error. The code outline for the selectivity function is:

```
STATIC FUNCTION ODCIStatsSelectivity(pred  SYS.ODCIPredInfo, sel
OUT NUMBER, args SYS.ODCIARGDESCLIST, strt NUMBER,  stop NUMBER,
sigcol     IMGIDXMethods, qsig       IMGIDXMethods, wstr
VARCHAR2, thresh     NUMBER,  env        SYS.ODCIEnv)
RETURN NUMBER IS
BEGIN
```

```
IF (args(1).ArgType != ODCIConst.ArgLit AND
        args(1).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
END IF;

/* compute the stop value */
    IF (args(2).ArgType != ODCIConst.ArgLit AND
        args(2).ArgType != ODCIConst.ArgNull) THEN
      RETURN ODCIConst.Error;
    END IF;
.....
/* determine selectivity for the three conditions, >=0, > 1, NULL)
*/
.....
    RETURN ODCIConst.Success;
  END;
```

The parameter `pred` specifies the parameter for which the selectivity is being computed, while `sel` returns the computed selectivity as a percentage value. `args` parameter contains the list of the arguments with which the function, the type method, or the operator was called. `start` and `stop` specifies the lower and upper bounds of the function respectively, while `sigcol` specifies the signature for which the function is called. `wstr` specifies weight string, which includes the number and position of the colour value we are looking for. `env` contains the parameters about the environment in which the routine is executing.

4. Associating methods with database objects

The user-defined statistics methods have to be associated with appropriate database objects for the optimiser to use them. The following statement associates the statistics with the types, index types and functions defined in the CBIRC cartridge:

```
ASSOCIATE STATISTICS WITH TYPES signature_type USING
ColorStatistics;
ASSOCIATE STATISTICS WITH INDEXTYPES IMGIDXMethods USING
ColorStatistics ;

ASSOCIATE STATISTICS WITH FUNCTIONS
Op_ColourIsPresent
```

```
Op_ColourIsExact
Op_ColourIsGreaterThanValue
Op_ImageIsExact
USING ColorStatistics;
```

5.5 Testing the data cartridge

The sub-components of the CBIRC must be assembled into the server before they can be used for image retrieval. This involves loading the Java source and class files in the Oracle JVM as described in chapter 2, publishing and making the classes accessible from SQL and defining SQL scripts for creating all object types and tables. Provided the sub-components are installed in the server, the cartridge can be used implement the Students application described in chapter 3 and depicted in Figure 3 - 1 in the following way:

- **Create the table for storing the images.**

```
CREATE TABLE students
 (
StudentNumber  NUMBER     PRIMARY KEY   NOT NULL,
StudentName           VARCHAR(50)              NOT NULL,
Course                VARCHAR(50)              NOT NULL,
StudentPhoto          IMAGE_TYPE         NOT NULL,
PhotoSignature        SIGNATURE_TYPE);
```

- **Insert a record into the table.**

```
INSERT INTO STUDENTS (StudentNumber, StudentName, Course,
StudentPhoto, PhotoSignature)
VALUES
(6011760,'Kandeshi','Doctor',Image_type.init(Source_type.init('/ima
gedirectory/kandeshi.gif')),
Signature_type.init(Source_type.init('/imagedirectory/kandeshi.gif'
)));
```

- **Create an index on the signature to speed up retrieval.**

```
CREATE INDEX PHOTOINDEX ON STUDENTS (PHOTOSIGNATURE) INDEXTYPE IS
IMGIDXMethods parameters('test');
```

- **Search the database for images similar to the image inserted in step 2**

```
SELECT c.studentname
FROM students c, students q
 WHERE
  q.studentnumber = 6011760
 AND
   ARESIMILAR ( c.photosignature, q.photosignature);
```

5.6 Outlook on the design options

The Extensible Indexing framework allows the cartridge developer to define the structure and methods of the index used by the server to index user-defined data types. Each implemented index type must define how and where the index is stored, how the index is maintained during index update operations and how the index is searched during query processing operations. As such the content and structure of the index is controlled by the user-defined extensions, giving power and flexibility to the cartridge developer.

The extensible indexing framework does not restrict the location of the index storage, allowing the actual index data to be stored inside the database in form of database tables or other user-defined structures, or outside the database in the form of files. This, however, revealed different implications for concurrency and recovery. While the data stored inside the database is protected by concurrency mechanisms of the database, concurrency and consistency is not guaranteed for index data stored outside the database. As an example, the index file was deliberately locked for updating by an instance of the database that uses the CBIRC cartridge. Accessing this file led to a deadlock because it was not possible to control the locking and unlocking of index file from the database.

Perhaps the main advantage of using the extensible indexing framework in a data cartridge is that, once the index type is registered with the database, the maintenance and access of the custom index structure is hidden from the user and thus the user does not have to worry about the maintenance issues. The database knows about the existence of the index and can thus use it to manage all the index related functionality. The index is completely integrated with the data, in such a way that

application can define routines that manage and manipulate index data in the same way as built-in indexes to evaluate SQL queries. The database automatically invokes user-defined functions to build and maintain the index functionality using the cartridge functionality. This way, all update on the index table is automatically reflected on the index data, thereby guaranteeing that the index is always consistent with the table data, and thus maintaining the integrity and correctness of the database.

The extensible optimiser also has interfaces to enable them to recognise user-defined indexing. This enables the optimiser to approximate the selectivity and cost of executing domain indexes.

5.7 Conclusions

Chapter 4 and 5 demonstrated most of the extensibility interfaces that can be implemented to extend the capabilities of the database. It must, however, be noted that there is neither a minimum nor a maximum number of components that should be implemented for the component to be considered a data cartridge. A data cartridge can, for example, extend only the type system and the indexing capabilities without the query optimisation features, etc. However, it is clear that Oracle's intent is for the provided extensibility interfaces described above to serve as a guide for extending the capabilities of the database.

Broadly speaking, data cartridges provide a sufficient platform to completely integrate new retrieval functionality within the database server. The data cartridge approach also incurs many object-oriented paradigm benefits including application-based control and ability to modify data cartridge components independent of the database. Once the data cartridge components are installed, users define and manipulate cartridge components through SQL, while at the same time accessing standard Oracle features. The only evident shortcoming is that the extensibility interfaces provided by the Oracle do not reduce the complexity of developing data cartridge components. Rather, they seem to be focused on guiding the database engine in accessing the implementations provided by the cartridge developers.

Chapter 6:A data cartridge for face recognition

An important goal of extensibility is to improve the modelling of real-world entities in the database using object technology. This chapter uses a face recognition application to explore the implications of using an object-oriented approach in Oracle9i. The recognition application is first developed as a standalone application in Java and then integrated as a data cartridge. The integration revealed that most key object-oriented concepts are unusable for objects types that contain LOB data types. It was also found that not all Java APIs required for LOB and image processing are supported in the database server and the user must avoid classes that use GUI objects if successful integration is to be achieved.

6.1 Introduction

In an object-oriented environment, complex applications are better modelled as objects. It is no surprise then that one of the motivations of extensibility in ORDBMS is the modelling of complex data using the object-relational technology. As explained in chapter 2, however, not all ORDBMS support the object-oriented capabilities to the same degree. Some of the approaches on how to model objects in an object-relational environment are proposed in [SAH87].

In this chapter, the support for object-orientation in Oracle is explored using a face recognition application developed in Java. Due to the growing interest in biometric authentication, face recognition has become a widely researched topic today and is widely used in numerous applications such as airport and banking buildings for security purposes [SUB98]. Faces are complex and cannot be easily described using simple shapes or patterns. As such, similarity measures do not always perform accurate matching.

The face recognition cartridge is a more specialised and complex cartridge in comparison to the CBIRC cartridge developed in the previous two chapters. This is because face recognition systems work by firstly detecting faces in an image. As such, the system must know what a face looks like and learn to recognise it among other faces. This involves *training* the recognition algorithm to recognise known faces, which requires the signature of the faces to be regenerated every time a new face is added to the database. The face recognition cartridge thus uses high-level information to

create the image signatures as compared to the low-level colour comparisons described in CBIRC. In this thesis, the face recognition algorithm is firstly implemented as a standalone application. It is then adapted and mapped to a database application, which requires a different integration mechanism than the one used in CBIRC.

6.2 Objectives

The main reason to write the data cartridge described in this chapter was to explore how the behaviour and structure of existing classes can be mapped to maintain complex relationships among objects in the database. Mapping existing objects to databases can be problematic, because the object structure, classes and behaviours are usually written in an object-oriented language such as Java or C++. Databases such as Oracle, on the other hand, do not support all the functional components of these languages and thus require a mapping layer to store Java and C++ objects in the database. This has often resulted in what is well-known as the *impedance mismatch* [OLL98]. In this chapter therefore, the mechanisms that can be used to map existing Java objects to Oracle object types are explored. Although the chapter reports on the experience of mapping Java objects, the focus of the chapter is on the capabilities and limitations of the actual database engine in supporting these objects, rather than the particulars of the Java language. The tradeoffs involved in extending an ORDBMS using Java may be found in [GMS98].

As discussed in [BUR01], the object-oriented approach in Oracle presents a number of database design and implementation options. According to [SUK99], the principal challenges and opportunities with object-oriented approach in ORDBMS include (i) handling domain specific data, (ii) handling object behaviour, (iii) handling object references (iv) handling collections of objects and (v) mapping and handling objects on the client side. Rather than attempting to present a state-of-art face recognition algorithm or introduce new face recognition techniques, this chapter discusses the object-relational features pertaining to these key challenges and opportunities.

6.3 The face recognition application

This section describes the implementation of the face recognition algorithm, discussing the steps in the recognition process and explaining how the algorithm is trained to classify and recognise faces from a group of images.

6.3.1 Content representation

Each image is searched for "face-like" structures, because the only relevant object in an image in this cartridge is a face. Like the images described in CBIRC, each image is stored in a physical image file consisting of a header and an image data matrix. Images are thus still represented using the `Source_type` and `Image_type` types described in the previous chapter, although the signature type changes. The signature in this cartridge is aimed at the regions of the image where face-like structures are found.

6.3.2 Signature representation

Assuming that each image in a training set of 2-dimensional n x n pixels images contains a face, and the training set is exposed to the face detection algorithm, each image will occupy n^2 points in the 2-dimensional space. Assuming also that faces occupy almost the same size and position in each image, faces will most likely not occupy the whole n^2 space and they will be closely distributed in this 2-dimensional space because they are similar in structure, making them easier to detect. Similarly, images not containing faces should occupy different areas of in this dimensional space. Faces can thus be described using a lower dimensional subspace.

The above idea is proposed in [TUP91], which presents an approach to train the algorithms to recognise faces by projecting all face images to a dimensional feature space to construct the average face of the entire image database. Using the same algorithm in this cartridge, the position of each image in the space is described by finding the Euclidean distance between the points in the average face and its Eigenvectors. The Principal Component Analysis (PCA), is then used to calculate the Eigenvectors that best describe the distribution of faces in the face space, so that each unique image occupies different points in the image space. The resulting feature space, which constitutes the signatures of the faces, is stored along an image identifier, such as a row identifier, that associates signature components to database image instances.

6.3.3 The recognition process

Recognition of faces is performed by comparing the query face to a set of images in the signature. For each query image a set of eigenvectors is computed and is projected to the signature of the training set. Moreover, it is possible to determine whether this image consists of a face since the algorithm knows what a face looks like. It is also possible to determine whether a query face is similar to a comparison face by specifying a threshold value that assumes that a query image is similar to the comparison face if the value resulting from their comparison is below the specified threshold. Assuming then that an image is a face, the distance between the query image and all other images in the signature is calculated by projecting it onto individual entries in the signature. Since similar images are closer to each other in the signature, the face is identified by finding the nearest known face from the signature. However, even tiny changes in the size of the image, the size of the face in an image or head orientation can cause the location of the image in the signature to change dramatically, resulting in false matches. The steps that summarise the recognition process as summarised in Figure 6 - 1 are given below.

1. *Initialisation,* where the entire image database set is exposed to the EigenFace extraction algorithm to compute the average face.

2. Each new image is projected to the average face and its set of weights and EigenFaces are determined and stored in a feature space.

3. When a new image is submitted to the training set, the points in the image are compared to check if it is close to the average face.

4. If it is close to the average space, it is most likely an image of a face. Thus, determine whether a similar face is already in the database or not.

5. If a similar image is already in the database, make another reference to where it may be found. Otherwise, add it as a new face.

6. If an image that is not close to a feature space is repeatedly found in the database, accept it as a face and learn to recognise it.

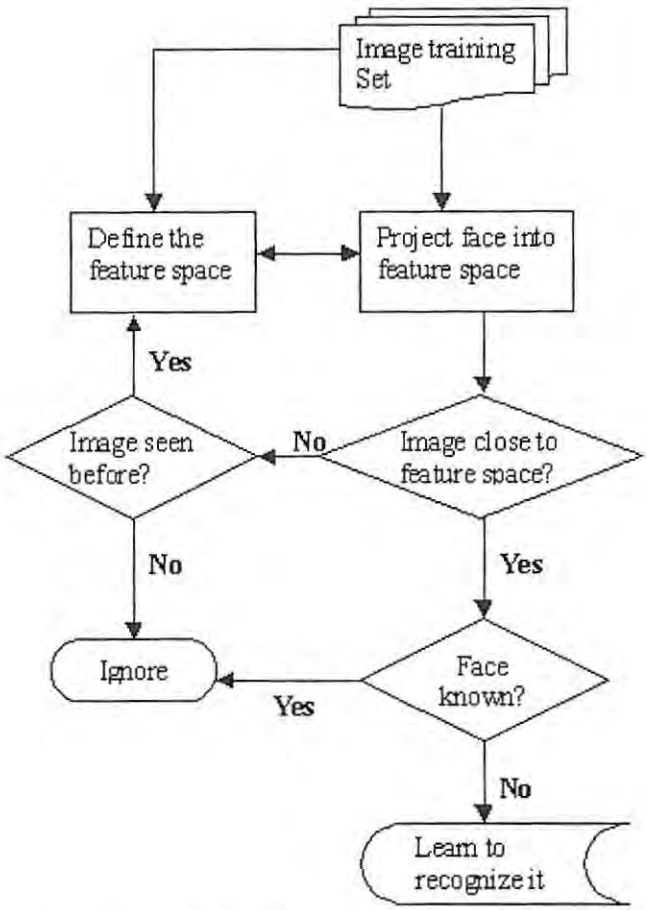

Figure 6 - 1 : The face recognition process

6.4 *Implementing the data cartridge*

As already explained in the previous section, the data cartridge was implemented by firstly developing a standalone Java application that consists of the classes shown in table 6-1, and was independent of the database. The signature of the training set is computed by the class `FaceSpaceComputation`, and stored in a standard operating system file.

Class Name	Description
FaceSpaceComputation	Computes the face space
FaceSpace	Signature with average space and Eigen faces of all images
FaceSpaceCreator	Used for matching images

Table 6 - 1 : Java classes used to the face recognition data cartridge

The two basic entries of the face recognition cartridge are the image itself (`source_image`) and the image signature (`signature_type`). `Source_image` can contain images of different types and thus

has a subtype called `Image_type`, which is used to classify the images that are interpretable by the cartridge. An image of type `image_type` can have two types of images: an image containing a face, or an image containing an unknown object. Since the face recognition cartridge only recognises images with faces, the signature is only created for images that contain faces. The entity and relationships of the face recognition data cartridge entities is shown in Figure 6 - 2.

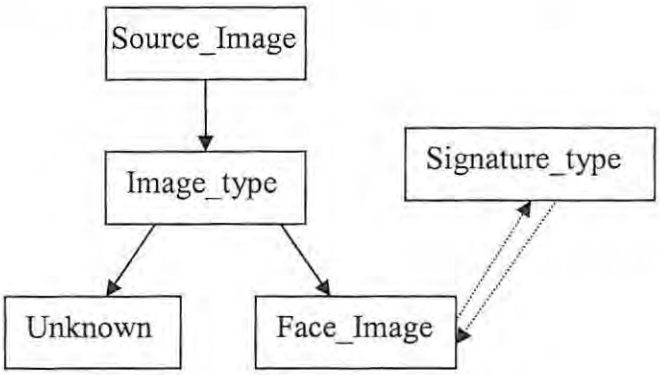

Figure 6 - 2 : Graphical representation of face recognition entities and relationships

The second part of implementing the face recognition data cartridge was to create object types and their methods. This involved using the infrastructure to support the use of object data that matches the data model used in object-oriented applications provided by Oracle. Part of this is *Oracle SQLJ*, which is used in this cartridge to create object types and their methods. (*SQLJ* is a standard that "*allows SQL code to be embedded into Java code in a way that is compatible with the Java design philosophy*" [WRI02]). Together with JDBC, Oracle SQLJ provides sets of interfaces to access SQL data from Java classes and persist Java objects in the database, with JDBC interfaces allowing SQL types to be mapped to Java classes while Oracle SQLJ allows Java types to be mapped and used inside the database. This allows for a real object-oriented architecture that treats each entity as a single atomic unit in the database.

SQLJ classes are mapped to the database by extending the `ORAData`, `SQLData` or `CustomDatum` interfaces provided by Oracle. These interfaces allow the SQLJ class elements to be accessible from the server using standard SQL. The attributes of each SQLJ class are treated like column attributes of standard object types and the corresponding class methods are treated like SQL object methods. In the following code segments, the SQLJ classes that implement the required face recognition functionality are presented. All the presented classes implement the `oracle.sql.SQLData` interface. The first code segment outlines the implementation of the `Source_Image` object.

```
public class Source_Image implements SQLData
{
   protected String sourcePath;
   protected BLOB data;
   protected Date updateTime;
   protected String sql_type;

   public void readSQL(SQLInput stream, String stype) throws
SQLException{}
   public void writeSQL(SQLOutput stream) throws SQLException{}
    ..................
public static Source_Image init() throws SQLException{}
   public static Source_Image init(String s) throws IOException,
   public String getSourcePath() throws IOException{}
}
```

To demonstrate how inheritance is supported in Oracle, the `Image_type` object is implemented so that it extends the `Source_image` type. The code outline for the `Image_type` is shown below:

```
public class Image_Type extends Source_Image
{
   protected int width;
   protected int height;
   protected String fileFormat;
   protected String contentFormat;
   protected String compression;
   protected long length;
   protected String sqltype;
   protected SQLData data;

public Image_Type(){}
   public Image_Type (String s) throws IOException, ImgException,
SQLException{}
   ..................
}
```

The `face_type` and `unknown` type inherits from `Image_Type`. The implementation for the `face_type` is shown below. Since images with "unknown" states, (that is, they do not contain faces) are not represented in the signature, the implementation of the `unknown` entity is ignored in the rest of this chapter.

```
public class Face_Type extends Image_Type
{
public BLOB face;
public Face_Type() throws IOException, SQLException{}
.....
}
```

The signature for the set of database images is then created using the `Signature_File` class, which has the following code outline:

```
public class Signature_File implements SQLData
{
public BLOB signature;
public static void init(Resultset rs) throws IOException,
SQLException{}
}
```

All the classes described above are first compiled using a SQLJ compiler. They are then loaded into the database using the `loadjava` command-line utility, together with the classes shown in table 5-1. However, the classes shown in table 5-1 are only used as a reference by the `Signature_File` to generate the face space, and are thus, never published in the database. The three SQLJ classes shown above are published to allow persistent storage of objects in the database using the `CREATE TYPE` statement. Since Oracle SQLJ allows a one-to-one mapping of the Java classes to SQL object types, where each attribute is mapped to the corresponding SQL types, the `source_file` object type is represented to the database as:

```
CREATE OR REPLACE TYPE Source_type
AS OBJECT
  EXTERNAL NAME 'Source_Image'
  LANGUAGE JAVA USING SQLData
  (
  sourcePath VARCHAR2(4000) EXTERNAL NAME 'sourcePath',
  data BLOB EXTERNAL NAME 'data',
  updateTime DATE EXTERNAL NAME 'updateTime',

  STATIC FUNCTION init return Source_Image_typ
        external name 'init() return Source_Image',
```

```
  STATIC FUNCTION init(name VARCHAR) return Source_Image_typ
        external name 'init(java.lang.String) return Source_Image',
  .....
) NOT FINAL;
```

The EXTERNAL NAME clause identifies the Java class whose attributes and methods are mapped to a corresponding type in the server and while the USING clause specifies the interface that describes how the type is represented in the server.

As seen in the last line of the type declaration, the Source_type is declared as NOT FINAL. NOT FINAL keyword in Oracle indicates that the type can have subtypes. All user-defined types are declared FINAL by default, and can thus not have subtypes. Methods can also be declared as FINAL to prevent the subtype from altering them or NOT FINAL to allow the subtype to override them. Another important keyword pertaining to inheritance that can be used with the object type is NOT INSTANTIABLE, which means instances of the object type cannot be created. This allows correct representation of abstract classes in the server.

To map the SQLJ class Image_type in the database, the CREATE TYPE statement must indicate that the type is a subtype by specifying the supertype using the UNDER parameter. The code outline for the Image_type is shown below:

```
CREATE OR REPLACE TYPE Image_type
UNDER Source_type
  EXTERNAL NAME 'Type_Image'
  LANGUAGE JAVA USING SQLData
  (
   WIDTH NUMBER EXTERNAL NAME 'width',
   LENGTH NUMBER EXTERNAL NAME 'length',
   .....
  STATIC FUNCTION init return Type_Image_typ
        external name 'init() return Type_Image',
        .....
);
```

The Image_type inherits all the attributes and methods of source_type, and any changes made to source_type are reflected in Image_type as well. Once objects of type Image_type have been

created, changes to source_type are restricted since the type contains subtypes that are dependent on it. Apart from the overridden methods, Image_type and all its subsequent subtypes will, in addition to new attribute and methods that the individual subtype introduce, accumulate all the attributes and methods in Source_type. As an example the Face_type can be represented as:

```
CREATE OR REPLACE TYPE Face_type
UNDER Image_type
  EXTERNAL NAME 'Face_type'
  LANGUAGE JAVA USING SQLData
  (
   face BLOB EXTERNAL NAME 'face',
   .....

  STATIC FUNCTION init return Face_type
        external name 'init() return Face_type',

        .....
);
```

In addition to the attributes of Image_type, Face_type also inherits all the attributes and methods of Source_type, thereby allowing type hierarchy to be maintained among supertypes and subtypes. This facilitates code reuse because the code does not have to be rewritten to make it available in different object types in the type hierarchy. However, Image_type is declared as a final type, meaning that subtypes of Image_type cannot be created. The Face_type can thus be simply created using SQL as:

```
CREATE OR REPLACE TYPE Face_type AS OBJECT
(
  face BLOB,
  STATIC FUNCTION INIT(content Image_type) return Face_type
);
```

Note that an SQL type such as Face_type described above, cannot inherit directly from the Image_type type, unless it is first declared as a SQLJ type that uses SQLData. The ideal situation would have been to allow the SQL type Face_type to be created as:

```
CREATE OR REPLACE TYPE Face_type
UNDER Image_type
(
  face BLOB,
  STATIC FUNCTION INIT(content Image_type) return Face_type

);
```

However, SQLJ object types can only have SQLJ types as their supertype or subtypes. This implies that inheritance is only allowed among object types with similar representation in the database, so that SQLJ object types only inherit from SQLJ objects and SQL object types only inherit from other SQL types.

6.5 Observations

This section presents some of the implications observed when mapping Java classes to Oracle object types.

6.5.1 Using the cartridge object types

Like classes in an object-oriented environment, type declarations do not allocate storage to objects. The class must be mapped into a table to store object instances. One way to achieve this is by using object tables. An object table is a special kind of table in which each row represents an object. As an example, an object table `images` that stores instances of `Image_type` is defined as:

```
CREATE TABLE images OF Image_type;
```

The `images` table is treated as a single column table where each row represents an object of type `Image_type`, allowing object-oriented operations to be directly applied to the table. The table contains a combination of the attributes of `Image_type` and `Source_type` and can be initialised with an instance such as:

```
INSERT INTO images VALUES
('c:\ear',empty_blob(),null,10,20,30,'gif','gif','rle8');
```

However, when a table consists of a hierarchy of subtypes, the list of attributes becomes very long. A better choice would seem to enable the table object to be instantiated using a constructor function such as:

```
INSERT INTO images VALUES (Image_type.init());
```

This, however, was not possible with all the types defined in this cartridge. Although the constructor function returns an object of type `Image_type`, the attributes of images are stored as a list of all the attributes of `Image_type`, rather than as a single, atomic attribute called `Image_type`, causing the constructor function to return inconsistent data types when executed.

Another notable issue with an object table such as `images` is that an object table cannot be instantiated with `NULL` values. This is because each object or row is referenced as an object, which, as in standard object-oriented programming, must have an identity and a state. Oracle automatically assigns an object identifier value (OID) to each individual row in an object table, which allows each row object to be referred from other objects or relational tables. Assigning `NULL` values to objects invalidates their state in an object table.

The `images` object table can also be treated as a multi-column table where each column is an attribute of the object type `Image_type`, allowing relational operations to be performed on the table such as:

```
SELECT width FROM images;
```

6.5.2 Storing object collections

The face recognition cartridge treats a set of face objects as a collection, which is stored in the database as a set of images. Oracle allows sets of types to be stored as Collections, which can either be nested tables or Variable Arrays (VARRAYS). Collections define a unit that contains an indefinite number of elements, all belonging to the same data type. The following example creates a set of face objects based on the `Image_type` type:

```
CREATE or replace type face_array as VARRAY(10) of image_type;
```

The above statement declared an array collection of face images. As one would expect, an array stores an ordered set of data elements, where each element has an index number corresponding to the position of the element in the array. (The number 10 in the brackets indicates the maximum size of the array.) Since the array is declared as a type, it does not allocate storage space, and must be used as an attribute of an object type or a data type of a table to allocate storage. The following statements uses `face_array` as an attribute of an object type called `face_object`.

```
CREATE OR REPLACE TYPE face_object as OBJECT (face face_array);
```

A preferable solution would be for Oracle to allow arrays to be used as a data type of a table as in the following statement:

```
CREATE TABLE face_tables(id number,face face_array);
```

However, the above statement only works when the array contains simple data types such as numbers and strings. It does not work with `face_array` because `face_array` is a VARRAY of type `Image_type`, which contains an embedded LOB.

Another collection type that can be used is the nested table collection, which contains an unordered set of data elements. Nested tables can contain other nested tables to create a hierarchy of tables within other tables. The following statement declares a nested table type, which nests tables of face images.

```
CREATE or replace type face_set as TABLE OF image_type;
```

The statement defines a type, and thus, does not allocate storage space. The nested table type can again be used as an attribute of another object type or an attribute of a column in a table. Again, it is not possible to store nested tables with columns containing embedded LOBs.

Because of the shortcomings with storing collections of columns containing embedded LOBs, collections were not used to store the training set of the image recognition cartridge. Instead, each image of the set is stored as an individual object in a standard table, while the signature, which is a computed from the entire training set, is stored as a BLOB in the `signature_type` object.

6.5.3 Storing object references

As mentioned earlier, Oracle assigns an OID to every object stored in an object table to allow objects stored in rows to be referred to from the relational tables and from other objects. Specifically, the *REF* keyword is used to represent such reference objects, which in practice is simply a logical pointer to a row object. References are also used to model associations among objects, eliminating the need to use foreign keys and providing the means to navigate between objects. An example of how the REF function works is shown below:

```
select REF(P)
from Images P
where width = 7;
```

In the above example, the REF function takes "P" as input, which is an alias to the object table, to return the actual OID for the row object selected. The query returned the following results:

```
REF(P)
--------------------------------------------------------------------
00002802095C3AD6AD3B3C4CD7B080A49C97BACD9835619CD6E25C49F8AC8879791
B3DA8F0040F0590002
```

This value by itself is not useful, as it does not give the actual value of the row object. A dereferencing function, called *DEREF,* is provided to return the actual object instance corresponding to the REF. As an example, the DEREF function may be used as:

```
select DEREF(REF(P))
from Images P
```

References can either be scoped or dangled, with scoped references constrained to references to specified object tables while dangling references can refer to objects that do not exist. Despite the many potential benefits of the referencing and dereferencing functions, these functions can also not be applied to object tables that use columns with embedded LOBs, making them unusable in the face recognition data cartridge.

6.5.4 Handling domain specific data

It was not possible to create object types on the fly within Oracle; they had to be created and stored in the database using languages such as PL/SQL, Java, and SQLJ before they could be used and shared by other programs. Once object types were defined, they are used as row objects in object tables, or as column objects, which are attributes of a table.

As seen in the implementation, the object-oriented view of Oracle eliminates the impedance mismatch between the applications and the database, allowing corresponding attributes and methods of standalone applications to be effectively mapped and used inside the database. The data structure, along with its corresponding functions, can be encapsulated in a Java class, and an interface that presents the mapping between the actual Java class and the object type in the database can be implemented using SQLJ. Like classes in Java, however, object types do not allocate storage to object instances, and an object table or object type that uses the user-defined object must be defined to allocate object storage.

For each user-defined object, Oracle implicitly creates an object type, which can be used to create instances of that object type. Although user-defined constructor functions are also allowed, these could only be used with column objects or row objects whose types were implemented using PL/SQL. Row objects whose types were implemented using SQLJ required the specification of a value for all the corresponding attributes of the object type.

As expected, it was only possible to update object types that did not have dependents in the database. Object dependents that are referenced in any schema objects such as tables, views and other objects could not be updated, making it difficult for types to evolve once they were defined and used. In such a case, the user could either define new objects using older objects, or first delete all the object dependencies before updating the type.

Oracle provides `MAP` and `ORDER` methods that allow objects of the same type to be compared. (An object type can only have one comparison function at a time.) `MAP` methods are applied to each object individually, and their results are then compared. `ORDER` method, on the other hand, compares two objects simultaneously and returns the results of the comparison. `MAP` functions can, however, only be used with functions that return built-in data types, making them more efficient for operations such as comparing numerical values. `ORDER` functions, on the other hand, can also be used for functions that return complex objects. Since Oracle recommends that each object type

should have either a MAP or ORDER method defined (otherwise a field-by-field comparison of attribute values is done when object comparisons are performed, which do not always imply similarity), the function used to compare faces for similarity in this data cartridge was implemented as an ORDER method, which provides a more natural way of comparing objects.

6.5.5 Handling inheritance

As seen in the implementation, object types could be declared as NOT FINAL to enable them to be used as supertype for other object types. Although this gave power and flexibility to object types, it also introduced a higher level of complexity and dependency. A typical example is on the depth of the type hierarchy, where a subtype could be defined from a supertype either directly or through multiple levels of other subtypes. As an example, `Face_type` was created under `Image_type`, which was also created under `Source_type`. Although this is an advantage from the inheritance point of view, it could prove difficult to evolve any of the supertypes in the type hierarchy, especially if any of the subtypes has table dependencies. Oracle, however, provides the option of declaring types as NOT INSTANTIABLE, which implies that they cannot be instantiated. To minimise the difficulty of evolving types, therefore, supertypes that are most likely to change and do not require direct table dependencies, could be declared as NOT INSTANTIABLE.

It was also noted that SQLJ object types that inherit from other objects are stored as unpacked objects, where each of the attributes from all the supertypes is mapped to a column in the table containing them. Consequently, it was not possible to instantiate a subtype with the constructor to the supertype, as can be done with a call to `super()` in Java. Other observed characteristics of handling inheritance in Oracle include the supports for only single inheritance, so that each subtype can have exactly one supertype, although, a supertype can have more than one subtype. This was, however, expected since Java also only supports single inheritance. Like Java, Oracle also supports dynamic polymorphism, which allows the appropriate version of the method to be executed based on the type of the object.

6.5.6 Handling collections of objects

As already mentioned, Oracle supports two collection types: VARRAYS and `nested tables`. VARRAYS store a limited, ordered set of elements, where each element has a position that uniquely

identifies it in the array. `Nested tables` on the other hand are not ordered, and allow tables to be embedded inside other tables. `VARRAYS` and `nested tables` could, however, not be effectively used in this data cartridge because they do not support columns containing LOB objects. It was, however, noted that indexes could not be created on `VARRAYS`. `Nested tables`, on the other hand, allow indexes to be created, and can thus be used with more complex collections of data than `VARRAYS`.

6.5.7 Handling object references

As previously mentioned, objects are stored as row objects or column objects in Oracle. In addition to columns corresponding to all the attributes of the object, row objects also store an additional column that stores a unique, system-generated object identifier. Column objects on the other hand are stored as scalar values, and thus do not have object identifiers. Once objects were stored as column objects, they could only be accessed by selecting the columns they occupy. Object stored as row objects, on the other hand, could be referenced using their object identifiers, via the discussed REF function.

As noted in the implementation, however, the REF function also could not be used with object types containing LOB columns. Usually, the REF function is used as a substitute for primary key – foreign key relationships, eliminating the need to perform the complex joins used in the relational model. References also simplify object querying, because they support navigational access rather than associative access. (Associative access is used with foreign keys, where the entries in one table must be looked up from another table, while navigational access using multiple level of pointers to retrieve complex objects.) An example of a preferable way where the REF function could be used in the face recognition data cartridge would have been:

```
select REF(P)
from Images P
where width = 7;
```

According to [SUK99], the "natural way to model relationships between objects" provided by navigational access does always not guarantee optimal solutions if referenced objects reside in the same table. Hence, associative access was used to model relationships, since object references could also not be effectively used with objects containing LOB columns.

6.5.8 Handling objects on the application side

As discussed in the integration process, oracle provides the JDBC and SQLJ interfaces to provide a mapping of objects between the server and the standalone application. In addition, data in relational tables can also be retrieved and updated as if it were stored as objects using object views, providing a seamless interface for fetching of objects into a client side cache. Using JDBC and SQLJ, it was possible to switch between the server and client side code without affecting the consistency of the data, thus providing flexibility and different object schema representation using both the relational and object view. The cartridge also demonstrated how JDBC and SQLJ facilitate a seamless integration of already developed object-oriented applications into the database in section 6.4.

6.5.9 LOB management

Oracle provides support for four different kinds of LOBs: (i) BLOBs, which are unstructured binary data, (ii) CLOBs, which are single byte large character data, (iii) NCLOBs, which are multiple byte large character data and (iv) BFILEs, which are large binary files stored outside the database. Apart from the BFILEs, the actual LOB data is stored inside the database table space (also called inline storage), and thus participates in the transaction model of the database. This means that data stored inline is protected by many of the database's features such as concurrency and recovery. BFILE data on the other hand, does not participate in the transaction model, and any support for data integrity and recovery must be provided by the application or underlying operating system. In this data cartridge, actual face images were stored as BLOBs, while the signature was stored as a BFILE.

The maximum length restriction for all column data and buffer size when processing SQL queries is 4KB. Only face images with a size less than or equal to 4GB could be stored as internal LOB (LOB stored inside the database) because of the maximum size restriction stipulated by Oracle. Images of greater size could have been stored outside the database as BFILEs, or they could also be compressed so that they fit into 4GB, or be broken into chunks of less than 4GB that are stored as separate columns or separate rows. Storing images this way, however, could further complicate the development of the data cartridge. As a result, the data cartridge only used images of a size less than 4GB.

6.6 Conclusions

This chapter demonstrated how existing Java classes might be integrated as a data cartridge in Oracle, exploring how some of the important object-oriented concepts such as inheritance and encapsulation can be realised in an object-relational environment. Although Oracle does not support the object-oriented paradigm completely, it provides a reasonable level of support for it. Developing the face recognition cartridge required a proficient understanding of the different extension possibilities. The code was initially developed for a standalone application, and its integration into the database required writing code that is harder to understand and maintain. Since most of the important object-relational features could not be applied to LOBs, integrating this data cartridge emphasised the need to extend the support of object-relational concepts to LOB data, otherwise Oracle's object-relational engine are still viewed as having the same limitations as the relational engine when dealing with LOB data.

Chapter 7:An extensible data cartridge: integrating GiST

This chapter exits the image retrieval domain used to integrate data cartridges in the previous two chapters, to demonstrate the integration of generalised search trees (GiST) into Oracle, exploring extensible indexing interfaces in greater detail. The GiST is implemented as an extensible data cartridge that provides low-level indexing functionality to other data cartridges, simplifying their development. The integration revealed that although the indexing extensibility mechanisms of Oracle provides a set of interfaces that guide the data cartridge developer in extending the database, it does not shield the developer from the implementation of low-level details, and thus does not reduce the implementation effort of user-defined functionality.

7.1 Introduction

The previous chapters described the implementation of data cartridges that solve domain specific problems in the domain of image retrieval. An alternative to this approach is to implement one "supercartridge" with an extensible architecture, which can be extended by "subcartridges" to specialise the functionality to the required domains. This not only reduces the development time and makes the cartridge components more manageable and consistent, but also facilitates code sharing and, generally, makes the software more reliable. This way, the development of custom data types can be delegated to domain experts of the data type while leaving the complicated data cartridge features to database developers.

This chapter moves out of the image retrieval domain to explore the extensibility of indexing techniques in Oracle in greater detail. The implementation is based on the Generalised Search Trees (GiST), a template indexing structure originally proposed in [HKP95] and adapted for better performance and concurrency and recovery mechanisms in [KOR00]. GiST has been described as one of the most extensible indexing structures available today and has been used to enhance the indexing capabilities of multimedia applications such as MPEG-7 Multimedia

Data Cartridge [KOS02], in image indexing and retrieval applications such as BLOBWorld [CTB+99], and for investigating issues regarding concurrency and recovery protocols in Informix Server in [KOR99].

7.2 Overview of the GiST architecture

The GiST framework provides template algorithms for building balanced index trees such as the B-tree and the R-tree. Internal nodes and leaves have an index entry (*k,ptr*), where *k* is a predicate used as a search key and *ptr* is a pointer to another node for internal nodes and a row identifier for leaves. The predicate key may, in principle, take any arbitrary predicates, and needs not be ordered. This implies that GiST is not limited to any data type, making the framework highly flexible and extensible in the kind of data it can index. Predicate keys are categorised into different sub-categories using some user-defined characteristics. The categorisations are then used to form a tree hierarchy, where each node stored under a hierarchy belongs to a certain category. Queries issued against the GiST are conducted by searching the index using the categorisations, which makes GiST extensible in the kinds of query it can handle.

GiST provides all the required abstract methods for the search functionality in a single data structure. When extending the GiST, the user only specifies the properties and characteristics of the specific tree that distinguish it from other trees. These properties, which are a set of methods for specific functions such as insertion, deletion and searching of data in a tree, are summarised in table 7-1.

Method	Description
Consistent	Determines whether there is a possibility that data stored below a given node may match the predicate.
Union	Computes the union of a set of predicates
Compress	Compresses the representation of the predicate
Decompress	Decompresses the compressed representation of the predicate
Penalty	Computes the penalty for inserting a new key into a specific node
PickSplit	Split the page when there is an overflow

Table 7 - 1 : Summary of the GiST interface methods

The generic algorithms provided by GiST and the methods from table 6-1 that must be implemented by the user to realise them are discussed below:

Search

Search transverses the tree to find entries that satisfy a specified search predicate. For unordered nodes, the search recursively descends all the paths in the tree that contain a key consistent with the predicate search key. For ordered domains, the algorithm returns the minimum tuple in the tree that satisfy the predicate. Assuming the nodes are sorted in ascending order from left to right, this is achieved by descending the left most branch of the tree for the branch whose entries are consistent with the predicate search key, and returning the first key in its leaf node. The interpretation and evaluation of this qualification for data entries in the tree is implemented by the user-defined `consistent()` function, which takes a query predicate and a page entry as arguments and return true if the entry matches the predicate.

Insert

Insert is used to add new entries in the tree in such a way that the tree still remains balanced after the insertion. Given an entry *(k,ptr),* and a level *l*, the algorithm finds a new single leaf where to insert the entry and returns a new GiST resulting from inserting the entry at level *l*. GiSTs allow overlapping search predicates, for which there might be more than one leaf at level *l* where the new entry can be inserted. Before inserting the entry therefore, the user-defined `penalty()` function, which takes the new key and page entry as arguments and returns a penalty that reflects how much the tree needs to be expanded to accommodate the new entry, might be computed. Since there is more than one leaf where the new entry might be inserted, the path with the least penalty is chosen until the right place to insert the entry is found. However, each leaf contains a limited number of elements, and inserting a new entry may result in an overflow. In this case, case the leaf must be split using the user-defined `PickSplit()` method. If the parent nodes do not cater for the new key, the tree must be expanded from the root so that the new key is put at the right place. The user-defined `union()` function is then used to combine the new entry with the old entries in the tree so that the tree can be updated.

Delete

Delete removes a key from the GiST in such a way that the balance of the tree is maintained. This function works by descending the tree to locate the leaf whose key is specified in the delete argument and removes it from the tree. Deleting an entry can result in underflows, so the tree must be adjusted accordingly using the user-defined `union()` function to keep the balance of the tree. This may result in the original tree being shrunken.

7.3 Design and implementation of the INDEXTYPE

The algorithms for implementing the GiST interface are given in both [HKP95] and [KOR00], and the standard GiST code developed in C++ may be found at [BEC00]. This code has been used in [DOK02] and [KOR99] to extend the Oracle and Informix databases respectively with additional indexing techniques. Both [DOK02] and [KOR99] divided the functionality of the index in three components: the `GiSTCore`, the index extension and the user-defined data type. In [DOK02], the `GiSTCore` component is stored as an external database module that requires the database to be configured to recognise and use it.

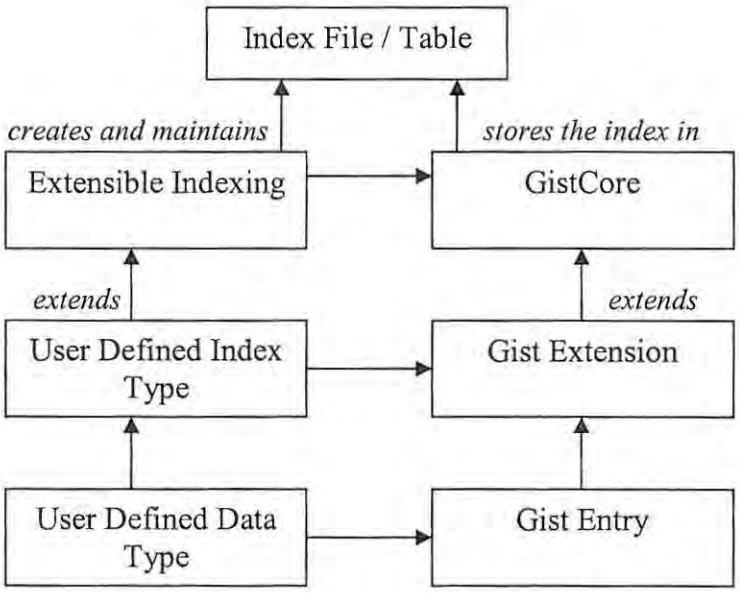

Figure 7 - 1 The GiST Implementation Approach

The code for `GiSTCore` in this thesis was developed in Java so that all the code is run from inside the database, as opposed to the external procedure approach used in [DOK02]. The implementation approach used in this thesis is shown in Figure 7 - 1.

The approach represented in Figure 7 - 1 consists of two major parts: the database dependent part and the database independent part. The database dependent part is on the left side of the figure, and is constituted by the User-Defined Data Types, the User-Defined Index Types and the Extensible Indexing Interface. The database independent part is on the right side of the figure, and is constituted by the classes that can run independently from the database. These classes include the `GiSTCore`, the `GistExtensions` and the `GistEntry`. The implementations of the constituents of the two parts are discussed in more detail in the next sections.

7.3.1 Implementing the database independent components

The database independent components are the generic interfaces that must be extended by the user to implement the functionality of GiST in the database. These interfaces implement the GiST interface methods shown in table 7-1. The implementation consists of nine Java classes, as shown in table 7-2. In addition, the table also shows the `PageFile` class, which is used to layout the index pages in the index file. The index pages contain data for which the index type issues queries to read, write and update.

Class	Description
GistConstants	Stores the library constants such as number of entries in the GiST
GistEntry	Defines the predicate keys and their pointers
GistNode	Defining GiST nodes
GistPenalty	Defines the `penalty()` of implementing methods
GistPredicate	Defines the `consistent()` method
GistKey	Defines the predicate key of the gist
Gist	Creates a Gist tree
GistListNode	Contains a list of nodes in the gist
GistList	Maintains the position (such as front and real) of the Gist tree
PageFile	Implements the storage manager for a tree

Table 7 - 2 : Classes used to implement the GiST Core

The classes in table 7-2 are developed outside the database and loaded into the database using the `loadjava` utility. Since these classes are abstract classes that need to be implemented, they are loaded as Java source classes to make them editable from the Oracle Enterprise

Manager Console. The main class for the index type is `Gist.java`, and contains methods to insert, search, delete and create the index file. The code outline for `Gist.java` is shown in appendix C.1.

`Gist.java` creates and stores the index tree in a Random Access File outside the database or in a database table. The implementation must thus include operations to issue commands to write, scan and delete data stored outside the database when a statement that references the GiST is issued. It must also incorporate the operations to create tables and files to store index data. Upon invocation, the function parameters and the description of the index are used to either build a new index or create indexed data in the new column.

7.3.2 R-tree : Extending the database independent component

The sample tree that extends GiST in this example is a balanced R-tree that uses an *n*-dimensional space represented as a square. As [HKP95] argues however, R-trees provides only the `Contains` key predicate, and do not allow the specification of the `PickSplit` and `Penalty` algorithms introduced by the GiST. R trees were also found to lack optimisation mechanisms for linearly ordered domains, providing limited extensibility in terms of the features offered by the GiST architecture. Because of their simplicity, however, they are used to demonstrate the extensions and usage of the GiST architecture in this example.

The keys in the R-tree are a set of integers, representing the lower left and upper right corners respectively. As an example, a key given as (x_1, y_1, x_2, y_2) represents (x_1, y_1) and (x_2, y_2) on the Cartesian plane. For simplicity, the key (x_1, y_1, x_2, y_2) is also used to represent the predicate Contains((x_1, y_1, x_2, y_2),q), where q is used as a free variable which represents the search key. The tree is used to search nodes that intersect with a given rectangle, and all rectangles that contain a given point. As such, the query predicates supported in this class are:

- Contains (key, q)
- Overlaps (key, q)
- Equals (key, q)

Initially, the Java classes that extend the GiST interface methods shown in table 7-1 are implemented. As explained in [HKP95], the implementations of the above operators is done as follows:

- **Contains** $((x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)}), (x_{ul(2)}, y_{ul(2)}, x_{lr(2)}, y_{lr(2)}))$ only returns true when $(x_{lr(1)} \geq x_{lr(2)}) \Lambda (x_{ul(1)} \geq x_{ul(2)}) \Lambda (y_{lr(1)} \geq y_{lr(2)}) \Lambda (y_{ul(1)} \geq y_{ul(2)})$
- **Overlaps** $((x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)}), (x_{ul(2)}, y_{ul(2)}, x_{lr(2)}, y_{lr(2)}))$ only returns true when $(x_{ul(1)} \geq x_{lr(2)}) \Lambda (x_{ul(2)} \geq x_{lr(1)}) \Lambda (y_{lr(1)} \geq y_{ul(2)}) \Lambda (y_{lr(2)} \geq y_{ul(1)})$
- **Equals** $((x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)}), (x_{ul(2)}, y_{ul(2)}, x_{lr(2)}, y_{lr(2)}))$ only returns true when $(x_{ul(1)} = x_{ul(2)}) \Lambda (y_{ul(1)} = y_{ul(2)}) \Lambda (x_{lr(1)} = x_{lr(2)}) \Lambda (y_{lr(1)} = y_{lr(2)})$

As mentioned earlier, R-trees do not allow the specification of the `Penalty` and `PickSplit` methods. `Compress` and `Decompress` methods are also optional implementations. So, of the six GiST methods shown in table 7-1, only two were necessary. These were then implemented as follows:

- **Consistent** (E,q). For each entry E = (p,ptr), where p `Contains` $(x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)})$ and q either `Contains`, `Overlaps or Equals` $(x_{ul(2)}, y_{ul(2)}, x_{lr(2)}, y_{lr(2)})$, return true only when q `Overlaps` $((x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)}), (x_{ul(2)}, y_{ul(2)}, x_{lr(2)}, y_{lr(2)}))$. This means that an entry E is *consistent* with q if there is no doubt that the data stored below the pointer ptr may match q.
- **Union** $(Eq = ((x_{ul(1)}, y_{ul(1)}, x_{lr(1)}, y_{lr(1)}), ptr), \ldots, En(x_{ul(n)}, y_{ul(n)}, x_{lr(n)}, y_{lr(n)}))$, return $(Min(x_{ul(1)}, \ldots, x_{ul(n)}), Max(y_{ul(1)}, \ldots, y_{ul(n)}), Max(x_{lr(1)}, \ldots, x_{lr(n)}), Min(x_{lr(1)}, \ldots, x_{lr(n)}))$

The implementation of the R-tree is summarised in the classes shown in table 7-3. Note that this table also contains a class for the `RtreeEntry`, which defines the interface used to enter new keys in the GiST tree. The `RtreeEntry`, corresponds to the GiST Entry in Figure 7 - 1.

Class	Description	Methods Implemented
Rtree	Main class for the Rtree functionality	
RtreePageFile	Creates persistent storage for the index	Extends page file
RtreeFile	Keeps a record of used nodes	Union
Rtreeentry	Defines the Interface for the entry	Insert
Rtreepredicate	Defines query predicates	Consistency

Table 7 - 3 : Classes for R-tree implementation

Since the user-defined object types that use the cartridge described in this chapter are already defined in chapter 4, the next section proceeds to the implementation of the index type.

7.3.3 Implementing the database dependent components

As explained in chapter 5, the data cartridge mechanism provides a set of APIs that are used in the extensibility framework to build different data cartridge components. One of these APIs is the extensible indexing API, which provides a set of interfaces that must be implemented to support user-defined indexing techniques in the database. Using this framework, different indexing components can be developed independently of each other (for example, one can create operators without creating the index type), which are later combined to enable the database server to invoke them when executing SQL commands involving user-defined operations. As seen in the implementation of the previous data cartridges, however, the implementation is started afresh every time a new technique is required, making it complex and time consuming because there are so many interfaces to implement and internal details to take into account.. Using a generic indexing framework, such as GiST, has the potential of specialising the data cartridge APIs to provide extensible indexing and operator interfaces, that at the same time also allow different combinations of data types and operators to use different indexing techniques. This section describes how the database independent components were integrated with Oracle's extensible indexing interface to provide the functionality for calling the appropriate generic index methods in the database.

The first step in integrating the database independent components in the database is to define the index functions that use these components. Oracle provides a package called Oracle Data Cartridge Interface package, also known as the oracle.ODCI.* package, which is a Java-based interface to allow domain specific operators and indexing schemes to be defined and integrated into the server from standalone Java applications. This package has been used to enhance the indexing capabilities of Oracle in [1] [DOK02], which, as mentioned earlier, implements the actual indexing functionality as an external procedure. Oracle9i, however, did not allow the user-defined Java classes to extend the oracle.ODCI.* package. Attempting to extend the package gave the following error:

[1] The author uses Oracle8i version

```
xxx : The method java.util.Dictionary getTypeMap() declared in
nested class xxx._Ctx cannot override the method of the same
signature declared in class
sqlj.runtime.ref.ConnectionContextImpl.  They must have the
same return type.
```

where xxx is the name of the class in the ODCI package. The error suggests that there are some unresolved errors in the Oracle9i ODCI package, which could therefore not be used in this case.

As a result, the index extensions were done using the SQL-based interface demonstrated in chapter 5. Initially, PL/SQL functions were created and bound to the Java functions in the `Gist.java` file, to create an interface between the functions and the server. As an example, the function below is used to create and bind the function for inserting an entry in the index:

```
CREATE OR REPLACE FUNCTION GistCreate (minentries IN
BINARY_INTEGER, maxentries IN BINARY_INTEGER) RETURN
BINARY_INTEGER AS
LANGUAGE JAVA
NAME 'Gist.Insert(GistEntry, java.lang.int) return
java.lang.int';
```

The corresponding indexing operation for the above functions is:

```
   STATIC FUNCTION ODCIIndexCreate (ia    SYS.ODCIINDEXINFO,
                       parms VARCHAR2,
                       env   SYS.ODCIENV)
   RETURN NUMBER,

   stmt VARCHAR2(1000);
        crs NUMBER;
   temp NUMBER;
   min NUMBER := 1;
   max NUMBER := 10;

   BEGIN
   -- Construct the SQL statement for creating the table
   stmt := 'CREATE TABLE ' ||ia.IndexSchema || '.' ||
ia.IndexName || '_pidx' ||'( r ROWID, val GiSTEntry, pos
NUMBER)';
```

```
   -- Dump the SQL statement.
   sys.ODCIIndexInfoDump(ia);

   -- Execute the statement.
   crs := dbms_sql.open_cursor;
   dbms_sql.parse(crs, stmt, dbms_sql.native);
   junk := dbms_sql.execute(crs);
   dbms_sql.close_cursor(crs)
.....
    -- Now create the GiST File
temp = GistCreate(min, max);
.....
   RETURN ODCICONST.SUCCESS;
  END;
```

And the same is done for all the ODCI Interface methods explained in chapter 6.

7.4 Conclusion

Although Oracle provides interfaces to allow the user to control and define the content of the index, it does not simplify the index development process. Oracle provides the Oracle Database Indexing Interface (ODCI) and numerous examples in Oracle8i documentation that shows how to extend the interface using Java. However, it was impossible to extend the ODCI interface in Oracle9i using a Java application. As a result, it was also not possible to build a general index inside the database that does not require the user to extend the ODCI interface. To build a general index, therefore, the indexing techniques were divided into database-independent and database-dependent components. This, in a way, removes the flexibility of developing data cartridges because it forces the developer to follow the cartridge development process discussed in chapter 4 (see Figure 4 - 1).

Building a general index could also increase the flexibility of the data types that can be indexed using the same access method. The separation demonstrated in this data cartridge did not completely eliminate the development of user-defined extensions, because the extension still has to extend the numerous ODCIIndex interface methods. Nevertheless, it facilitated code reuse since the already developed GiST library was used to integrate the ODCIIndex

functionality. The preferable solution would have been for the index type to accept supertypes to accommodate instances of the data types.

This chapter demonstrated the integration of the extensible GiST data cartridge, which was tightly integrated with the database engine to provide a general, easy to use and performance enhanced framework inside the database. GiST was implemented using a database-independent and a database-dependent component, allowing developers to extend it to support different search trees both inside and outside the database. GiST allowed the developers to concentrate on the semantics of user-defined predicates and functionality of the actual tree, without being concerned with low-level details. This demonstrated that it is possible to reduce the implementation effort of user-defined functionality by building extensible data cartridges.

Chapter 8:Summary and conclusions

This chapter summarises the findings in developing sample data cartridges in Oracle9i and discusses the experience gained from it.

8.1 Motivation for the thesis

A major benefit of the object-relational technology is the ability to add user-defined functionality to the database. Using this capability, the database can be enhanced with functionality that was previously unavailable in the database to support a broader class of application requirements. In the Oracle database, this possibility is offered by the data cartridge mechanism, which allows user-defined data types and their indexing and query optimisation to be integrated into the Oracle database model. The data cartridge mechanism, however, presents a number of choices with implications that are not so obvious. There was therefore a need to explore the key features of the data cartridge mechanism, to analyse the implications of using them, and to discuss and suggest ways to improve the usage of these features if efficient data cartridges are to be built.

Although there are numerous methods that could be used to respond to the above need, the advantages and issues that come with the data cartridge mechanisms were explored by actually integrating data cartridges into the database. The integration pointed out features that might not be so obvious, and demonstrated how the capabilities of the underlying technology may be properly used.

8.2 Aim of the thesis

The aim of the thesis was to explore the mechanisms that can be used to integrate additional functionality in Oracle DBMS using the data cartridge mechanism. The thesis used a colour-based image retrieval technique, a face recognition algorithm and a GiST extensible architecture as three practical scenarios to achieve this aim. Image retrieval was used as the main domain for integration, but the principles and findings of this integration are applicable to the integration of any application that stores data as BLOBs and uses Oracle data cartridge.

8.3 General observations

The scenarios listed above allowed the exploration of the data cartridge services provided by Oracle to extend the database in four key areas: the type system, the server execution environment, the indexing structures and the query optimisation techniques. This section summarises the observations made when building data cartridges that extend these four areas.

8.3.1 User-defined data types

User-defined data types (UDTs) can be created from built-in or previously defined user-defined data types. It became apparent that object types cannot be created on the fly – they had to be integrated in the database server before they could be used in applications. Object types in Oracle are defined using PL/SQL, Java, or using C, where they are integrated as opaque types.

Unlike the extensible indexing and query optimisation services of the data cartridge mechanism, the type system does not have an SQL-based interface that facilitates the integration of UDTs. Although Oracle provides the interfaces such as `ORADATA` and `SQLDATA` to map Java attributes to their corresponding SQL attributes, there is no extensible framework that facilitates the integration of UDTs. An interface could usefully guide the implementation and integration of UDTs, especially those that are developed in external languages such as Java.

8.3.2 User-defined methods

The methods of the data cartridge can be written in PL/SQL, C, C++ or Java. PL/SQL and Java methods have the advantage of running on any operating system, enabling only one version to be released for different platforms. C and C++ code, on the other hand, requires different compilations on different operating systems such as Windows and Linux, and therefore requires extra configuration on different machines.

8.3.3 Server execution environment

PL/SQL and Java run from inside the database because there is a PL/SQL engine and an Oracle JVM integrated with it. Running inside the system, these languages automatically detect defective code such as buffer overflows or assignment of values to the NULL pointers, and in the worst case catch the exception or exit with an error code, without causing any damage to the system. C and C++ code, on the other hand, is run outside the database as a set of external procedures, called from inside the database and dispatched and executed in an external address space. A main drawback of this approach is that external routines do not have the DBMS's support for concurrency and recovery. Still, if the code is defective, it does not crash the database.

8.3.4 Extensible indexing

The extensible indexing extension of the data cartridge mechanism provides a set of interfaces for implementing a new index structure. Although the developer writes the actual code and defines how and where the code is stored, the actual indexing routines are automatically managed by the database that calls appropriate interfaces during query processing. These interfaces provide abstract functions for operations such as inserting an entry in the index, starting a scan of an index, getting the next entry, updating or deleting an entry and closing the index after a scan operation. The cartridge implementations in this thesis demonstrated three alternatives to extending the indexing capabilities of the database.

- CBIRC extended the underlying database indexing capabilities to user-defined data types by defining operators that were previously not directly supported by the database. This allowed the data cartridge to apply existing indexing methods such as B-tree to new data types, which already have the advantage of a tight integration with the database, and do not require the definition of low level functionality in the database engine, such as lock manager for locking index objects. Since CBIRC mapped existing indexing structures to the required structures, the performance of the index was not affected, due to the tighter integration of the indexing structured with the database. The index was also much easier to write and implement because it did not require additional modification to the underlying database engine.

- The face recognition cartridge required advanced database indexing and therefore could not be implemented using the indexing techniques offered by the database. This cartridge requires the index to be reconstructed every time a new face is added to the database, and uses more complex algorithms to create the signature. The cartridge required multiple search ranges, which are not based upon relational operators. Also, the index in the face recognition cartridge was stored outside the database in an operating system file, necessitating separate concurrency and recovery procedures.

- The GiST data cartridge is extensible in nature. In this thesis, the basic functionality of the GiST was separated into a database-dependent and a database-independent component. The database-independent component had to be implemented first. This component contains the basic functionality for calling the appropriate generic index methods inside the database, and thus provided a framework out of which the ODCIIndex interfaces used in the database were be built. The integration of GiST demonstrated the storage of the index information using a file-based tree index structure, which is accessed and managed by a table stored inside the database. The GiST architecture could not work inside the database, however, without being extended with user-defined data types and index type.

8.4 Other findings from integrating data cartridges

Simplicity of development

Conceptually, users are able to develop data cartridge components independently and integrate them, once completed, into the database. As seen in the implementations, however, different data cartridge components are much easier to develop and integrate if the type system is integrated first, as very little can be done if the data types to be used in the data cartridge are not defined. Already developed code such as Java classes that do not require attribute-to-attribute matching between the Java classes and SQL object types defined can be easily integrated in the database, but mapping such classes to equivalent SQL data types is more complex.

Database stability and developers' productivity

Apart from increasing the complexity and richness of domain specific applications that can be managed from the database, the support for objects is provided without compromising the scalability and robustness of the relational database engine. The data cartridge mechanism also allows the sharing of logic among database and client applications, thereby reducing the time needed to design and deploy applications in the database, while reducing the number of errors. Objects also accrue the benefits of caching because after the objects are accessed for the first time, they are kept in memory, allowing significantly better performance inside the database.

Ease of use

As demonstrated in chapter 5 and 7, the presence of the data cartridge mechanism and of the ODCI API does not in itself simplify the actual development of the data cartridges. However, once the data cartridges are integrated, they are treated in the same way as built-in database components. As seen in the implementations, however, some of the extensibility features such as Collections and REF functions could not be used with object types containing BLOB columns, which imply that the current database features do not sufficiently support complex data.

Main challenges for data cartridge development

The main challenge for the data cartridge developer is the learning curve required to understand the cartridge development and integration to enable already developed code to fit into the constraints of the data cartridge mechanism. Like other database extensions such as DataBlades in Informix and Extenders in DB2, maximum benefits of the data cartridge technology can only be realised if the appropriate data model and design choices are made during the data cartridge integration. Provided developers understand the practical implications of specific design choices, they can develop data cartridges and benefit from the productivity gain and ease of maintenance offered by them.

8.5 Conclusions

This thesis developed and integrated a colour-based image retrieval cartridge, a face recognition cartridge, and an extensible indexing framework to investigate the requirements and implementation aspects of the data cartridge mechanism of Oracle9i. The investigation was partially motivated by the continual improvement of retrieval techniques in the content-based image retrieval field, and by the appearance of new ORDBMS features that allow databases to be extended with new user-defined data types and their methods. Although Oracle ORDBMS offers many options for extending the database, the data cartridge mechanism was found to be most suitable to integrate complex and fully functional components in the database. The data cartridge mechanism allows the complete package of user-defined types, operators, indexing and query optimisation to be integrated into the database and to be transparently used by the end user.

Although Oracle provides a guide to developing data cartridges, it does not explain how much extensibility is safe, and does not prevent developers from implementing the entire data cartridge functionality, including low-level functionality, which may require a thorough knowledge of database internals. The use of extensible architectures such as the GiST allows knowledgeable data cartridge developers to develop low-level functions that require knowledge of database internals, while novice programmers can use the provided database interfaces for additional functionality. Even extensible data cartridges, however, do not provide all the required support for all possible data cartridges, since data cartridge requirements are arbitrarily defined by users. Building an extensible data cartridge relies on making good choices for the basic functions; otherwise extensible data cartridges may restrict the flexibility of subcartridges to some extent, and may not even effectively accommodate all the required functionality.

As noted in chapter 5 and 7, the mechanisms provided by the data cartridge simply define the sequence of calls that the database must perform during query processing, thereby guiding the database engine in accessing the implementations provided by the cartridge developers. Although data cartridges facilitate installation and de-installation of database schema elements, and guide the database engine in executing calls, implementing them is nevertheless a complex and time-consuming task. Indeed, if data cartridges that achieve user-defined functionality comparable to the provided database functionality are to be built, the provided interfaces must be improved to better guide the user in building data cartridges.

8.6 Future work

Possible future directions of work are the following:

- Many attractive features provided by Oracle, such as VARRAYS and REFERENCES, can only be used with simple, structured objects that are stored in the database. Considering the advantages and potential of these features, applications such as the face recognition cartridge described in this thesis could probably benefit more from the data cartridge mechanism if these features were extended to support large objects (LOBs). There is therefore a need to analyse carefully the shortcomings of these features, and to identify and develop similar techniques that can be used with large, unstructured data objects such as BLOBs.

- What features of the data cartridge should be evaluated to establish sufficiently the extensibility of Oracle database? It would seem necessary to establish what features of an ORDBMS are worth evaluating to determine the usefulness and capability of different integration mechanisms and individual features in different ORDBMS. A benchmark can then be designed to evaluate these features in different databases.

- Oracle provides other object-relational features such as object cache manager and object type translators, which were not investigated in this thesis. Future work could explore some of these features to produce a complete analysis of the implications of using object-relational features in Oracle, and to demonstrate how they can be used with the data cartridge mechanism to extend the database.

- Different ORDBMS have extensibility components with different strengths and weaknesses. Limited, if any, research has been undertaken to compare these components in different databases. The data cartridges presented in this thesis could be tested with different ORDBMS to be able to compare and contrast them to Oracle databases using a common framework.

Appendix A: Outline of CBIRC code

A1. Source_Image class code

```
public class Source_Image implements SQLData
{
   protected String sourcePath;
   protected BLOB data;
   protected Date updateTime;
   protected String sql_type;
   public Source_Image() throws SQLException {}
   public Source_Image(String s) throws IOException,
SQLException{}
   public Source_Image(BLOB cont) throws IOException{}
   public Source_Image(String s, BLOB cont) throws
IOException{}
   public Source_Image(String s, BLOB cont, Date d) throws
IOException{}
   public BLOB loadData(String s) throws SQLException,
IOException{}
   public String getSQLTypeName() throws SQLException{}
   public void readSQL(SQLInput stream, String stype) throws
SQLException{}
   public void writeSQL(SQLOutput stream) throws SQLException{}
   public static Source_Image init() throws SQLException{}
   public static Source_Image init(String s) throws
IOException, SQLException{}
   public static Source_Image init(BLOB b) throws IOException{}
   public static Source_Image init(String s, BLOB b) throws
IOException{}
   public String getSourcePath() throws IOException{}
   public BLOB getContent() throws IOException{}
   public Date getUpdateTime() {}
}
```

A2. Image_type class code

```
public class ImageFile
{
   public static int getWidth(String s) throws IOException {}
```

```
  public static int getHeight(String s) throws IOException {}
  public static long getLength(String s) throws IOException {}
  public static String getFormat(String s) throws IOException
{}
  public static String getContentFormat(String s) throws
IOException{}
  public static String getCompression(String s) throws
IOException {}
  public static BLOB loadData(String s) throws SQLException,
IOException{}
  public static String getPath(String s) throws IOException {}
  public static Date getUpdateTime(String s) throws
IOException {}
}
```

A3. Signature_type class code

```
public class ImageSignature
{
public static BLOB computeSignature(BLOB b) throws
IOException, SQLException {}
public static int findCombination(BLOB c, String weights)
throws SQLException, IOException {}
public static int ImageContains(BLOB c, String weights) throws
SQLException, IOException {}
private static BLOB loadData(byte[] s) throws SQLException,
IOException {}
}
```

A4. Image_type Body Code

```
CREATE OR REPLACE TYPE BODY Image_type AS
  STATIC FUNCTION init(name VARCHAR2) return Image_type IS
   BEGIN
         DECLARE
               I IMAGE_TYPE;
               T SOURCE_TYPE;
               b BLOB;
               d DATE;
```

```
            N VARCHAR2(4000);
            w NUMBER;
            h NUMBER;
            l NUMBER;
            ff VARCHAR2(4000);
            cf VARCHAR2(4000);
            fc VARCHAR2(4000);

        BEGIN
            b := IF_GETSOURCEFROMFILE(name);
            d := IF_GETDATE(name);
            N := IF_GETSOURCEPATH(name);
            T := SOURCE_TYPE(N,b,d);
            w := IF_GETWIDTH(name);
            h := IF_GETHEIGHT(name);
            l := IF_GETLENGTH(name);
            ff := IF_GETFORMAT(name);
            cf := IF_GETCOMPRESSION(name);
            fc := IF_GETFILEFORMAT(name);

            I := IMAGE_TYPE(T,w,h,l,ff,cf,fc);

            RETURN I;
        END;
  END Image_init;


MEMBER FUNCTION getNumber(attr VARCHAR2) return NUMBER IS
  BEGIN
    IF (attr = 'WIDTH') THEN
     RETURN SELF.WIDTH;
    END IF;

    IF (attr = 'HEIGHT') THEN
     RETURN SELF.HEIGHT;
    END IF;

    IF (attr = 'LENGTH') THEN
     RETURN SELF.LENGTH;
    END IF;
END getNumber;
```

```
MEMBER FUNCTION getString(attrb VARCHAR2) return VARCHAR2 IS
  BEGIN
   IF (attrb = 'FILEFORMAT') THEN
    RETURN SELF.FILEFORMAT;
   END IF;

   IF (attrb = 'CONTENTFORMAT') THEN
    RETURN SELF.CONTENTFORMAT;
   END IF;

   IF (attrb = 'COMPRESSIONFORMAT') THEN
    RETURN SELF.COMPRESSIONFORMAT;
   END IF;
 END GETSTRING;


 MEMBER FUNCTION getContent return BLOB IS
      BEGIN
           RETURN SOURCE.DATA;
 END GETCONTENT;


 MEMBER PROCEDURE setNumber (nattr VARCHAR2, num NUMBER) IS
 BEGIN
   IF (nattr = 'WIDTH') THEN
      WIDTH := nattr;
   END IF;

   IF (nattr = 'HEIGHT') THEN
      HEIGHT := nattr;
   END IF;

   IF (nattr = 'LENGTH') THEN
      LENGTH := nattr;
   END IF;
 END SETNUMBER;


 MEMBER PROCEDURE setString (sattr VARCHAR2, str VARCHAR2) IS
 BEGIN
 IF (sattr = 'FILEFORMAT') THEN
      FILEFORMAT := str;
   END IF;
```

```
    IF (sattr = 'CONTENTFORMAT') THEN
        CONTENTFORMAT := str;
    END IF;

    IF (sattr = 'COMPRESSIONFORMAT') THEN
        COMPRESSIONFORMAT := str;
    END IF;


  END SETSTRING;


  MEMBER PROCEDURE  setSource(src VARCHAR2) IS
  BEGIN
  DECLARE
        T SOURCE_TYPE;
        b BLOB;
        d DATE;
        N VARCHAR2(4000);

  BEGIN
        b := IF_GETSOURCEFROMFILE(src);
        d := IF_GETDATE(src);
        N := IF_GETSOURCEPATH(src);
        T := SOURCE_TYPE(N,b,d);
  END;
  END SETSOURCE;
 END;
 /
 SHOW ERRORS;
```

A5. Sample Functions for Image_type

A5. 1. IF_ GETCOMPRESSION

```
CREATE OR REPLACE  FUNCTION "IMAGE_USER"."IF_GETCOMPRESSION"
    (name VARCHAR) RETURN VARCHAR2 AS
LANGUAGE JAVA
```

```
NAME 'ImageFile.getCompression(java.lang.String) return
java.lang.String';
```

A5. 2. IF_ GETSOURCEPATH

```
CREATE OR REPLACE  FUNCTION "IMAGE_USER"."IF_GETSOURCEPATH"
    (name VARCHAR2) RETURN VARCHAR2 AS
LANGUAGE JAVA
NAME 'ImageFile.getPath(java.lang.String) return
java.lang.String';
```

A5. 3. IF_GETSOURCEFROMFILE

```
CREATE OR REPLACE  FUNCTION
"IMAGE_USER"."IF_GETSOURCEFROMFILE"
    (name VARCHAR2) RETURN BLOB AS
LANGUAGE JAVA
NAME 'ImageFile.loadData(java.lang.String) return
oracle.sql.BLOB';
```

Appendix B: Outline of the face recognition cartridge code

B1. Source_type code outline

```
public class Source_Image implements SQLData
{
   public Source_Image() throws SQLException{}
   public Source_Image(String s) throws IOException,
SQLException{}
   public Source_Image(BLOB cont) throws IOException{}
   public Source_Image(String s, BLOB cont) throws
IOException{}
   public Source_Image(String s, BLOB cont, Date d) throws
IOException{}
   public BLOB loadData(String s) throws SQLException,
IOException{}
   public String getSQLTypeName() throws SQLException{}
   public void readSQL(SQLInput stream, String stype) throws
SQLException{}
   public void writeSQL(SQLOutput stream) throws SQLException{}

   public static Source_Image init() throws SQLException{}
   public static Source_Image init(String s) throws
IOException, SQLException{}
   public static Source_Image init(BLOB b) throws IOException{}

   public static Source_Image init(String s, BLOB b) throws
IOException{}
   public String toString(){}
   public String getSourcePath() throws IOException{}
   public BLOB getContent() throws IOException {}
   public Date getUpdateTime() {}
}
```

B2. Image_type code outline

```
public class Image_Type extends Source_Image
{
```

```
  public Image_Type() throws SQLException{}
  public Image_Type (String s, String attrs) throws
IOException, SQLException{}
  public Image_Type (BLOB blobs, int attrs) throws
IOException,   SQLException{}
  public void readSQL(SQLInput stream, String typeName) throws
SQLException{}
  public void writeSQL(SQLOutput stream) throws SQLException{}
  public static Source_Image init() throws SQLException{}
  public static Source_Image init(String f, String s) throws
SQLException, IOException {}
  public static Source_Image init(BLOB bm, int attr) throws
SQLException, IOException {}
  public String toString(){}
  public String getSQLTypeName() throws SQLException{}
}
```

B3. FacespaceCreator code outline

```
public class FacespaceCreator
{
  public Face getMatch(Face f) throws IOException {}
  // Used to construct the face-spaces from the given faces.

  public void readFaceSpaces(Face[] n) throws IOException,
ClassNotFoundException {}
  // Submit the training set and construct a face-space
object.

  private FaceSpace submitSet(Face[] faces) throws
IOException, ClassNotFoundException {}
  //Saves the face-space object in <code>f</code> file.

  private void constructIndex(File f, FaceSpace bundle) throws
FileNotFoundException, IOException{}
  //Read from the index object.

  private FaceSpace scanIndex(File f) throws
FileNotFoundException, IOException, ClassNotFoundException {}
  //Construct the face-spaces from a given set
```

```
 private FaceSpace createFaceSpace(Face[] faces) throws
IllegalArgumentException, IOException {}

public double[] readImage(Face f) throws Exception {}
```

B4. FaceSpace code outline

```
  public class FaceSpace implements Serializable, Comparable
{

  public FaceSpace(double[] avgF, double wk[][], double[][]
eigV) {}

  // Submit an query against the face-space.
  public void submitFace(byte[] face) {}
  public void submitFace(int[] face) {}
  public void submitFace(double[] face) {}
  // Clear the submitted image from the face-space object.
  public void clearFace() {}
 // The distance of how far away the submitted image is in
this face-space object.
  public double distance() {}
  //Compare this face-space bundle to another
  public int compareTo(Object o) {}
  //compute face
  private void compute() {}
  static double max(double[] a) {}
  static double sum(double[] a) {}
  static void divide(double[] v, double b) {}
 }
```

B5. FaceSpace code outline

```
//Computes an "face space" used for face recognition.
public class FaceSpaceComputation {

public static FaceSpace submit(double[][] face_v, int width,
int height) {}
}
```

B6. Face code outline

```
public class Face implements ORAData, ORADataFactory
{
  public static final String _SQL_NAME = "Face";
  public static final int _SQL_TYPECODE =
OracleTypes.JAVA_STRUCT;

  public void setConnectionContext(DefaultContext ctx) throws
SQLException{}
  public DefaultContext getConnectionContext() throws
SQLException{};
  public Connection getConnection() throws SQLException {}
  public static ORADataFactory getORADataFactory(){}
  protected Face(boolean init){}
  public Face() {}
  public Face(Connection c) { }
  public Face(BLOB content) throws SQLException, IOException
{}
  public Datum toDatum(Connection c) throws SQLException {}
  public ORAData create(Datum d, int sqlType) throws
SQLException {}
  public void setFrom(Face o) throws SQLException {}
  protected void setContextFrom(Face o) throws SQLException {}
  protected void setContextFrom(Face o) throws SQLException {}
  protected void setValueFrom(Face o) {}
  protected ORAData create(Face o, Datum d, int sqlType)
throws SQLException {}
  public BLOB getFaceContent() throws SQLException {}
  public void setFaceContent(BLOB content) throws SQLException
{}
 }
```

Appendix C: Outline of GiST code

C1. GiST code outline

```
abstract public class GiST implements GiSTconstants
{
  public GiST () {}
  public GiST(boolean _isOrdered) {}
  public GiST(int minEntries, int maxEntries) {}
  public GiST(boolean _isOrdered, int minEntries, int
maxEntries) {}
  public abstract GiSTnode CreateNode();
  public GiSTnode getNode (GiSTentry E, GiSTnode N) {}
  public GiSTentry findMin (GiSTnode N, GiSTpredicate q) {}
  public GiSTnode nextOnLevel (int level, GiSTnode N) {}
  public GiSTnode prevOnLevel (int level, GiSTnode N) {}
  public GiSTentry next (GiSTpredicate q, GiSTentry E) {}
  public GiSTlist search (GiSTpredicate q) {}
  public void delete(GiSTentry E) throws NullPointerException
{}
  public void condenseTree (GiSTnode L) {}
  public GiSTnode getRoot () {}
  public boolean getIsOrdered() {}
  public int insert (GiSTentry E, int level) throws
NullPointerException {}
  public void adjustKeys (GiSTnode N) {}
  public GiSTnode chooseSubtree(GiSTnode R, GiSTentry E, int
level) {}
  public GiSTentry searchMinPenalty (GiSTnode R, GiSTentry E)
{}
  public void split(GiSTnode N, GiSTentry E) {}
}
```

Appendix D: External code installation instructions

D1. Setting up the Oracle9i environment for external routines

1. An external procedure needs a listener process that connects and calls out to the external routine. To configure the listener process, modify the file "`tnsnames.ora`"

```
EXTPROC_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = IPC)(Key = PLSExtProc))
    )
    (CONNECT_DATA =
      (SID = proc_PLSExtProc)
    )
  )
```

Please note that the entry "`EXTPROC_CONNECTION_DATA`" should not be changed, but should remain as entered in the "`tnsnames.ora`" file. The specified key, in this case, "`PLSExtProc`" and SID, in this case, "`proc_PLSExtProc`" can be changed, but must correspond to the names given in the configuration of "`listener.ora`" as shown in number 2 below.

2. Configure `listener.ora` to add an entry to the external procedure listener.

```
SID_LIST_ext_proc_PLSExtProc =
  (SID_LIST =
    (SID_DESC =
      (PROGRAM = ExtProc)
      (SID_NAME = proc_PLSExtProc)
      (ORACLE_HOME = /ORA/ORA90/bin)
    )
  )

ext_proc_PLSExtProc =
```

```
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = IPC)(KEY = PLSExtProc))
)
```

The entry for program must be "extproc", and as already mentioned, SID_NAME must match the SID entry in the tnsnames.ora file. ORACLE_HOME must be set to the directory name where Oracle is installed, and the file that contains the external routine must, in this case, reside in the /bin directory of ORACLE_HOME.

References:

[AKJ02] S. Antini, R. Kasturi, and R. Jain. *A survey on the use of Pattern Recognition methods for abstraction, indexing and retrieval of images and video.* The Journal of Pattern Recognition Society, Vol. 35, pp. 945 – 965, 2002.

[BUR01] Burleson, Donald (2001). *The object/relational features of Oracle.* TechRepublic, Inc. http://www.dba-oracle.com/art_oracle_obj.htm

[CAR87] M.J. Carey(ed.). *Special Issue on Extensible Database Systems.* IEEE Data Engineering Bulletin (10:2), 1987

[CHIT01] V. Chitkara. *Colour-based image retrieval using compact binary signatures.* Master's thesis, Department of Computing Science, University of Alberta, 2001.

[CPZ97] P. Ciaccia, M. Partella, and P. Zezulla. *M-tree: An efficient access method for similarity search in metric spaces.* In the Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece, pp. 426 - 435, 1997.

[CTB+99] C. Carson, M. Thomas, S. Belongie, J.M. Hellerstein, and J. Malik. *Blobworld: a system for region-based image indexing and retrieval.* In Proceedings of the 3rd International Conference on Visual Information Systems, Amsterdam, pp. 509-516, 1999.

[CWN+97] M.J.Carey, D.J.DeWitt, J.F.Naughton, M.Asgarian, P.Brown, J.E. Gehrke and D.N.Shah. *The BUCKY Object-Relational Benchmark.* In Proceedings of the ACM-SIGMOD International Conference on the Management of Data, Tuscon, Arizona, May 1997.

[DOK02] M. Doeller, H. Kosch. *Enhancement of Oracle's Indexing Capabilities through GiST-implemented Access Methods.* Technical Report: Institute of Information Technology, University Klagenfurt. April 2002.

[FS95] M. Flicker, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J.Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. *Query by image and video content: the QBIC system*. IEEE Computer, Volume 28. pp, 23-32, 1995.

[FUT99] Y. Fu, and J. C. Teng. *Improving high-dimensional Indexing with heuristics for Content-based image retrieval*. In International Workshop on Integrated Spatial Databases, Maine, USA, pp. 249 – 267, June 1999.

[GAG98] V. Gaede, and O. Guenther. *Multidimensional access methods*. Computing Surveys, 30(2), pp.170 – 231. 1998.

[GIE02a] W.Gietz. *Oracle9i Data Cartridge Developer's Guide.* Release 2 (9.2), Oracle Corporation Documentation Part No. A96595-01, March 2002.

[GMS98] M. Godfrey, T. Mayr, P. Seshadri, and T. V. Eicken. *Secure and Portable Database Extensibility*. In Proceedings of the ACM SIGMOD Conference on Management of Data, Seattle, USA, pp.390—401,June 1998.

[HKP95] J.M. Hellerstein, J.F. Naughton and A. Pfeffer. *Generalized Search Trees for Database Systems*. Proceedings in the 21st International Conference on Very Large Databases (VLDB) Conference, Zürich, September 1995, pages 562-573.

[INF97] INFORMIX, *Extending Informix Universal Server: Data Types,* Informix Software Inc. Version 9.1. Part No. 000-3765A, March 1997.

[INF98] INFORMIX, *Extending Informix Universal Server: User-Defined Routines,* Informix Software Inc. Version 9.1, 1998.

[INF99] INFORMIX, *Excalibur Image DataBlade Module User's Guide.* Informix Corporation. Part No: 000 –5356. Version 1.2, March 1999.

[INF00] INFORMIX – *Image Foundation DataBlade Module User's Guide.* Informix Corporation. Part No: 000 –6902. Version 2.00, December 2000.

[INF01] INFORMIX - *DataBlade Development Overview,* International Business Machines Corporation Documentation Part No. 000-5403A, August 2001.

[KAB97] W. Klas., K. Abarer. *Multimedia and its impact on Database System Architectures*. In P.M.G. Apers, Multimedia Data and Perspective, Springer, London, 1997

[KAS97] N. Katayama, and S. Satoh. *The SR tree: An index structure for high dimensional nearest neighbor queries.* In the Proceedings of the ACM SIGMOND International Conference on Management of Data, Arizona, pp.365 – 380, May 1997.

[KOR99] M. Kornacker. *High-Performance Extensible Indexing*. Proceedings in the 25th Very Large Databases (VLDB) Conference, Edinburgh, Scotland, pp. 699-708, 1999.

[KOR00] M. Kornacker. *Access Methods for Next-Generation Database Systems*. PhD. Thesis, University of California, Berkeley, 2000.

[KOS02] H. Kosch. *MPEG-7 and Multimedia Database Systems*. ACM Sigmond Records, 31(2). June 2002. pp.34-39.

[LKK00] S.H. Lee, S.J. Kim, and W. Kim. *The BORD benchmark for Object-Relational Databases.* Proceedings in the 11th International Workshop on Database and Expert Systems Applications, Greenwich, London, United Kingdom 2000.

[LUS02] L. Luscher, *Oracle9*i *Database Performance Tuning Guide and Reference*, Release 2 (9.2) Oracle Corporation Documentation Part No. A96533-01, March 2002.

[MKL97] B. M. Mehtre, M. S. Kankanhalli, and W. F. Lee. *Shape Measures for Content Based Image Retrieval: A Comparison*, Information Processing and Management 33(3), pp: 319—337, 1997.

[OLL98] J. Olsson, A. Lassen, *Experiences from Object-Relational Programming in Oracle8*. Centre of Object Technology. 1998.

[OST95] V. Ogle and M. Stonebraker. *Chabot: retrieval from a relational database of images*, IEEE Computing 28(9). pp 40-48. 1995.

[TUP91] M.A. Turk and A.P. Pentland. *Face Recognition using eigenfaces*. Proceedings of the IEEE Computer Society, Conference on Computer Vision and Pattern Recognition, Hawaii, pp. 586 – 591. 1991

[POM99] K. Porkaew, M. Ortega and S. Mehrotra. *Query reformulation for Content-based multimedia retrieval in MARS*. IEEE International Conference on Multimedia Computing System. Vol. 2. pp. 509 – 516. 1999.

[RAG00] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, McGraw-Hill, 2000.

[RCH99] Y. Rui, S-F. Chang, and T.S. Huang. *Image Retrieval: Current Techniques, Promising Directions and Open Issues*. Journal of Visual Communication and Image Representation, Volume 10, pp. 39-62, 1999.

[RUS02a] J. Russell, *Oracle9i Application Developer's Guide - Fundamentals*, Release 2 (9.2), Oracle Corporation Documentation Part No. A96590-01, March 2002.

[RUS02b] J. Russell, *Oracle9i Application Developer's Guide - Fundamentals*, Release 2 (9.2), Oracle Corporation Documentation Part No. A96590-01, March 2002.

[SAC+96] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta. *A standard default colour space for the Internet – sRGB*. Version 1.10, November 5, 1996. http://www.w3.org/Graphics/Color/sRGB.html

[SAH87] M. Stonebraker, J. Anton, H. Hanson. *Extending a database system with stored procedures*. ACM Transactions on Database Systems (TODS). Volume 12 , Issue 3 (September 1987). Pages: 350 - 376.

[SAJ98] S. Santini and R. Jain. *Beyond Query by Example*. ACM International Conference on Multimedia. pp 345 –350.1998.

[SAJ99] S. Santini and R. Jain. *Integrated Browsing and Querying for Image Databases,* IEEE Multimedia Magazine, 1999

[SMC96] J. Smith and S.F. Chang, *VisualSEEK: a fully automated content-based image query system*. In Proceedings of the 4th ACM Multimedia Conference, Boston, MA, pp.87-98, 1996

[STA03] T. Stakemire. *Design of a Performance Evaluation Tool for Multimedia Databases with Special Reference to Oracle*. Masters Thesis. Computer Science Department. Rhodes University. 2003.

[STO86] M. Stonebraker, *Inclusion of new types in relational database systems*. In Proceedings of the International Conference on Data Engineering, Los Angeles, CA, pp.262 – 269, February 1986.

[STO96] Michael Stonebraker. *Object-Relational Database Systems: The Next Wave*. Morgan Kaufmann, San Francisco, 1996.

[SUB98] V.Subrahmanian. *Principles of Multimedia Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, California, 1998.

[SUK99] M. Subramanian, V. Krishnamurthy, *Performance Challenges in Object-Relational DBMSs*. IEEE Data Engineering Bulletin. Volume 22(2). Pages 27-31. 1999.

[SWB91] M. J. Swain, and H. D. Ballard. *Colour Indexing*. In International Journal on Computer Vision, pp.11-32, 1991.

[TMY78] H. Tamura, S. Mori, and T. Yamawaki. *Texture features corresponding to Visual Perception*. IEEE Transactions on Systems, Man and Cybernetics 8(6), pp 460 – 473,1978.

[WAR01] R. Ward, *Oracle9i interMedia User's Guide and Reference*, Release 9.0.1, Oracle Corporation Documentation Part No. A88786-01, June 2001.

[WEK99] U.Westermann and W. Klas. *Architecture of a DataBlade Module for the Integrated Management of Multimedia Assets*. Proceedings of the 1st International workshop on Multimedia Intelligent Storage and Retrieval Management, Orlando, Florida, October 1999.

[WHJ96] D. A. White and R. Jain, *Similarity indexing with the SS-tree*. In proceedings of the 12th International Conference on Data Engineering, Louisiana, pp.512 – 523, 1996.

[WNM+95] J. Wu, A. Narasimhalu, B. Mehrtre, C. Lam and J. Gao. *CORE: A content-based retrieval engine for multimedia information systems*. ACM Multimedia Systems, Vol. 3(1).1,pp. 25 – 41,1995.

[WRI02] B. Wright, *SQLJ Developer's Guide and Reference,* Release 2 (9.2), Oracle Corporation Documentation Part No. A96655-01, March 2002.

[WYA97] J. Wang, Y. Yang, and R. Acharya. *Color clustering techniques for color-content-based image retrieval from image database.* In Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMSC), Canada, pp. 442 – 449,1997.