LIB. COPY

CONTEXT BASED QUERYING FOR OODBMS

A dissertation submitted to Jawaharlal Nehru University in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE & TECHNOLOGY

By

Akkuluru Venkata Subbaiah

Under the guidance of

Prof. Parimala. N

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI 110067

JULY 2003

जवाहरलाल नेहरू विश्वविद्यालय

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLALNEHRU UNIVERSITY
NEW DELHI – 110067 (INDIA)

CERTIFICATE

This is to certify that the dissertation titled "**Context Based Querying for OODBMS**" which is being submitted by **Mr. Akkuluru Venkata Subbaiah** to the **School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi,** in partial fulfillment of the requirements for the award of **Master of Technology in Computer Science & Technology** is a bonafide work carried out by him under the supervision of **Prof. Parimala .N.** The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.

[illegible] 30-6-03

Prof. K.K. Bharadwaj
Dean, SC & SS
Jawaharlal Nehru University
New Delhi 110067

Parimala N. 30/6/03

Prof. Parimala. N
SC & SS
Jawaharlal Nehru University
New Delhi 110067

ACKNOWLEDGEMENTS

I am thankful to my guide, Prof. Parimala N. for suggesting me to do the work in *Context Based Querying for OODBMS* and being there to direct me all the time.

I would like to thank Prof. K.K. Bharadwaj, Dean, School of Computers and Systems Sciences (SC&SS), JNU for providing excellent lab facilities.

I would also like to thank my classmate Mr. V. Mallikarjuna Chetty and all my batch mates for their cooperation and advice.

A.Venkata Subbaiah

ABSTRACT

Object oriented databases are getting more and more popular day by day. To retrieve the data, object oriented query languages expect the user to formulate the requirements in terms of the query language. In order to lessen the burden of the user we propose a system wherein the user expresses the query without any explicit path expressions. The context for each query is maintained by the system. This context is used to translate the user query to a system query, which references the nested structures explicitly. If multiple system queries can be generated for the user query, then the closeness criterion is used to order these queries. These queries are rephrased using English like syntax. The user selects one or more, which are executed and the result is presented to the user. This query forms the context for subsequent queries.

CONTENTS

CHAPTER 1

INTRODUCTION

Object-oriented databases (OODBs) represent the latest generation of database systems technology. Object-oriented databases (OODBs) evolved from a need to support object-oriented programming and to reap the benefits, such as system maintainability, from applying object orientation to developing complex software systems. OODBs are based on the object model and use the same conceptual models as Object-oriented analysis, object-oriented design and object-oriented programming Languages. OODBs are designed for the purpose of storing and sharing objects; they are a solution for persistent object handling. Persistent data are data that remain after a process is terminated.

Object-oriented database management system adds database functionality to object oriented programming languages. They bring much more than persistent storage of programming language objects. Object -oriented DBMS extend the semantics of the C++, Smalltalk and Java object programming languages to provide full-featured database programming capability, while retaining native language compatibility. A major benefit of this approach is the unification of the application and database development into a seamless data model and language environment. As a result, applications require less code, use more natural data modeling, and code bases are easier to maintain. Object developers can write complete database applications with a modest amount of additional effort.

1.1 Object Oriented Database System:

A database management system consists of a collection of interrelated data and a set of programs to access that data. The collection of data is called *database.* The main aim of the database system is to provide an efficient and convenient access to the data stored.

Object-oriented database systems are new software systems integrating techniques from databases, object-oriented languages, programming environments and user interfaces. Objects in object-oriented programming languages exist only during program execution, but an OODatabase has capabilities for objects to be stored permanently and shared.

An object oriented database system should be

1. A database management system (DBMS), that is, it should have persistence, secondary storage management, concurrency, recovery and an ad hoc query facility.
2. An object -oriented system, that is, it should have complex objects, object identity, encapsulation, types or classes, inheritance, overloading and late binding, extensibility and computational completeness.

1.1.1 Object Oriented Features:

All the above features beginning with the object-oriented system are briefly explained below:

Complex objects can be built from simpler ones by applying constructors to them. An object-oriented database should support sets, tuples and lists. Sets are a natural way of expressing collections from the real world. Tuples are natural way of representing properties of an entity. Lists or arrays capture the order that occurs in the real world.

Object identity is the intrinsic property of an object, which distinguishes it from all other objects. Two different objects with the same values for all their members are not identical. Hence because of object identity, an object has existence, which is independent of its value.

Encapsulation hides the implementation leading to the behavior of the object. An object has an interface and implementation. The interface is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation has a data part, which is the representation or the state of the object, and a procedure part, which describes the implementation of each operation. Encapsulation distinguishes the specification of each operation from its implementation and thus leads to 'logical data independence' i.e. the implementation of an operation can be changed without changing the programs using that operation.

Object ***types*** and ***classes*** both summarise the common features of a set of objects with the same characteristics, but they are employed in different ways. A type corresponds to the notion of an abstract data type. It has two parts, the interface and the implementation. The specification of the class is same as that of a type but it is more of a run-time notion. Since it is used for the creation of objects it can be considered as an object factory.

Inheritance is a relationship among classes having common properties. With inheritance, each class (super or base class) can be specialized into a more specific class (sub or derived class) which inherits the methods and structure of the super class and in addition it contains some other members. Inheritance provides a means of avoiding the explicit and repeated storage of data which can be inferred, and helps code reusability. When two types *t1* and *t2* have features (attributes and operations) in common, those features can be abstracted out into a single subsuming type *T*, with *t1* and *t2* being declared as subtypes of *T* so that they automatically gain the common features without each having to respecify them.

Overloading and ***Late binding*** occur due to Polymorphism. Having two or more types use the same name for related but distinct attributes or operations is called overloading the name. The system can select the appropriate binding for the name at run-time by examining type of the object to which it is being applied (this is called late binding).

Computational completeness means that any computable function can be programmed using the system. Either the database description and manipulation language should itself be able to express any computable function (not the case for the RDBMS query language SQL, for example), or it should provide an application programmer's interface (API) to a standard programming language.

Extensibility means that the system comes with a set of predefined types or classes which can be extended i.e. there is a means to define new types and there is no distinction between system-defined and user-defined types.

1.1.2. Database Features:

All the features of database are briefly explained below:

Persistence is the ability of data to survive the execution of a process and to be eventually reusable in another process. This should happen implicitly - *save* and *load* operations etc. should not be required.

Efficient Secondary Storage Management is *a* classical feature of database system. It should be provided for fast and efficient manipulation of very large databases. Allocation of disk storage and transfer of data to and from main memory should be invisible to the application.

Concurrency means multiple users should be able to work concurrently on a database, viewing and updating shared data without affecting its integrity. The system should lock objects when they are read or written to prevent simultaneous access, and execute atomic operation sequences, which cannot be interrupted leaving the database partially updated.

Recovery means that, in case of hardware or software failure, the system should recover, that is, brought back to some coherent state of the data.

The system should also provide functionality of an ***ad hoc query facility*** to the user. The service consists of allowing the user to ask simple queries to the database simply. The query facility should be

High Level: The query facility should be reasonably declarative concentrating on *what* rather than on *how.*

Efficient: The formulation of queries should lend itself to some form of optimization.

Application Independent: The query facility should work on any possible database.

All the above mentioned topics are described in detail in [1] and [2].

1.2 Query Languages for OODBMS:

Database management systems support many interfaces so that the user can retrieve data from the database. A query language is the most commonly used and easy to use interface. Many query languages have been developed for retrieving data from objects stored in database. OQL and SQL3 are two more popular object oriented query languages. It explains how object oriented query languages, OQL and SQL3, facilitate the retrieval of data using examples from the objects stored in the database.

1.2.1 Object Query Language (OQL):

OQL is an object-oriented SQL-like query language. OQL is the query language of the Object Data Management Group (ODMG)-93 standard. It can be used in two different ways either as an embedded function in a programming language or as an ad hoc query language. It has special features for dealing with complex objects and methods in the object-oriented database O2. The examples given below illustrate the queries in OQL for retrieving data.

Consider the following schema

```
    Class Place_to_go
      Type tuple (Name: string,
                  Address: tuple (Street: string,
                                  City: string,
                                  State: string)
                  Details: string,
                    Phone: integer,
                    Things_to_do: set (Thing_to_do))
     Class Thing_to_do
       Type tuple (Name: string,
                   Description: string,
                   Closing_days: string,
                   Fee: integer)
```

Example 1:

Inorder to know what things can one do in Paris for less the fee 50 rupees", the reuired query is

```
select  x.name
from    y in Place_to_go
        x in y.Things_to_do
where   y.address.city = "paris" ans x.fee < 50
```

Example 2:

Similarly in order to know the price of a visit to Tajmahal the query required is

```
Select x.fee
From x in Tajmahal.Things_to_do
Where x.name= "visit"
```

More information about OQL can be obtained from [3] and [4]. The section below gives similar queries in SQL3.

1.2.2 Object Oriented SQL (SQL3):

SQL3 is an extension of SQL, the Structured Query Language, which includes the object-oriented concepts. As in the case of relational databases, the database supporting SQL3 also contains tables. The tuples of these tables are called 'row types' and the columns are the different 'types' that constitute a 'row type'. These 'row types' are similar to complex objects. More information about SQL3 can be obtained from [5] and [6].

Consider the schema containing tables Address with tuples of 'row type' AddressType, MovieStar with tuples of 'row type' StarType. The structure of these RowTypes is given below

```
Create row type AddressType (
        Street  char (50),
        City    char (20));
Create row type StarType (
        Name  char (30),
        Address AddressType);
Create table Address of type AddressType;
Create table MovieStar of type StarType;
```

Example 3:

To find the names and street of those MovieStars who stay in the city "Columbus", the required query is

```
Select MovieStar.name, MovieStar.address.street
From MovieStar
Where MovieStar.address.city = "Columbus";
```

1.3 Problems in Existing Approaches:

The above section (section 1.2) gives an introduction to the two query languages that are most commonly used. From the examples given in that section we can infer that in order to retrieve data even using an ease to use interface such as a query language the user must have knowledge about

1. ***The syntax of the query language:*** If the user doesn't know the query language, he cannot prepare correct queries and the difficult part for a casual user is the

framing of the query involved path expression and also the join information. That is, the complete nested structures have to be expressed while framing the query to retrieve the data needed by him.

2. ***The schema of the database:*** The user must know the different classes that comprise the schema of the database. The user must also know the relationships that exist between classes. For example in order to generate the query given in example 3, (section 1.2.2), the user must know that 'address' in *MovieStar* is an object of type *AddressType.*
3. ***Object Oriented Concepts:*** In order to understand the schema of the database and the relationships that exist between different classes in the database the user must have knowledge about the object-oriented concepts.

The main aim of this implementation is to provide the user with the data needed by him without expecting any such prior knowledge from the user.

1.4 The Proposed Approach:

In our approach the user will be provided with Graphical User Interface (GUI) for building the query. This interface allows the user to select the data needed by him and impose conditions, if any, on the data selected by him.

Every query is interpreted in its context. If the user chooses to execute the query in a new context, then the user query is translated into one or more system queries, which reference the nested structures explicitly. In general, there can be multiple queries that can be generated for the user query. The 'closeness' criterion is used to order the generated queries with the ones more closes higher in the order.

If the user chooses to execute the query in the old context, then the context left behind by the previous queries is taken into account to translate the user query into system query. If more than one such query can be generated, then these are, as before, ordered according to the 'closeness' criterion.

In either case the system queries are rephrased using English like syntax. The rephrased queries are presented to the user. The user selects one or more, which are executed and the result is presented to the user.

This interface hides the complexity that exists in retrieving data from different kinds of constructs that are supported by the object oriented database systems. This interface is not restricted either to a particular application or a database. This is open for all object oriented databases i.e. the same interface can be used to retrieve data from different object oriented databases. This interface is designed in Java and oracle is used as the backend to store data and respond to the queries sent.

CHAPTER 2

QUERY RESOLUTION

In this chapter we discuss about how the user request is transformed into system queries. Here the user will be provided with graphical user interface for building the query. In this system the user need not be aware of what information that exists in each database. Even user need not to specify explicitly the databases while submitting query. The Query interface allows the user to select the data needed by him and impose any conditions, if any, on the data selected by him. It also provides the interface to display the results, that is, the data retrieved by the user query. After submission of query, every query is interpreted in its context.

In this chapter and in the chapters to come objects of the following schema assumed to exist in the database.

```
create type address_type as object
( street    varchar2(10),
  city      varchar2(15),
  state     varchar2(10),
  pincode number(7));

create type details_type as object
( condate      date,
  closingday varchar2(10),
  entryfee     number(4),
  architect     person_type);

create type person_type as object
( firstname  varchar2(10),
  lastname   varchar2(10),
  age        number(3),
  nationality varchar2(10),
  address     address_type);
create type phone_varray as varray(5) of number(10);

create type monument_type as object
( name     varchar2(20),
  address  address_type,
  phone    phone_varray,
  details  details_type);

/* Object table to be used for REF from Tour table */
create table Monuments of monument_type;

create type hotel_type as object
 ( name varchar2(15),
   address address_type,
   stars number(3),
   details varchar2(60));

create table Tour
( city     varchar2(15),
  details  varchar2(60),
  what_to_see REF monument_type,
  where_to_stay varchar2(15)
  foreign key (where_to_stay) references HotelChain );

/* Nested table to be used with HotelChain */
create type Hotel_table as table of hotel_type;

create table HotelChain
( name       varchar2(15) primary key,
  headoffice varchar2(15),
  Hotels     Hotel_table)
```

```
nested table Hotels store as Hotel_table_tab;

create table Archeology
( name        varchar2(15),
  description varchar2(20),
  address     address_type);
```

In the above schema, the term 'REF' indicates that a member of an object points to some other object. For example, 'what_to_see', the member of object of 'Tour' type points to an object of type 'monument_type'.

A 'nested table' means collection of variable number of objects or members of a type. 'Hotels' in 'HotelChain' is a collection of objects that is nested table. A 'varray' is a collection of strings. 'Phone' in 'Monument_type' is a collection of strings.

Let the user query be

```
Select name, street
Where city contains 'New Delhi'
```

The system doesn't let the user know the structure of the data stored. When the user indicates the completion of selection to the system, it begins processing the user request by finding out classes, resolving the paths, preparing queries, handling conditions and resolving complex structures. All these stages are explained in detail below.

2.1 Finding Out Classes:

The schema of the database is maintained in a separate data structure. This can be easily updated to accommodate changes to the database schema. This structure can be as complex as B-Trees and Balanced trees to facilitate efficient searching in the case of large databases. In this implementation, schema is stored in a file and a linear search method is used.

The database schema is searched to find out the classes in which the selected attributes are members. This search yields a set of classes for each of the selected attributes such that each attribute is a member of each class in the set corresponding to that attribute. If a1, a2 and a3 are the attributes selected by the user then s1, s2 and s3 are the sets of classes such that a1 is a member of each class (or one of its component) in s1 and so on.

For the above user query, the attribute 'name' is mentioned in 'Monuments', 'Archeology', and 'HotelChain' say set s1, the attribute 'street' is mentioned in 'Monuments', 'Archeology', and 'HotelChain' say set s2 and the attribute 'city' is mentioned in 'Monuments', 'Archeology', 'HotelChain' and 'Tour', say set s3.

2.2 Resolving the Paths:

In the previous stage, a set of classes is identified for each attribute. Let C be an element in the set of classes for an attribute A. A may be a member of C or a member of a component class of C or a member of a component class of component class of C and so on. In this context, path of the attribute A describes the location of A in C. Recognizing this path is necessary to generate the query to retrieve the attribute A. This process can be termed as "Path resolution".

Let o1 be an object of 'Monument', o2 be an object of 'Archeology', o3 be an object 'HotelChain' and o4 be an object of 'Tour'. The paths generated for 'name' are:

```
o1.name
o2.name
o3.name
o3.Hotels.name
```

The paths generated for 'street' are:

o1.address.street

o1.details.architect.address.street

o2.address.street

o3.address.street

The paths generated for 'city' are:

o1.address.city

o1.details.architect.address.city

o2.address.city

o3.address.city

o4.city

In this stage we have to develop two lists on scanning the schema. One list contains the resolved paths for the selected attributes and the other list contains the corresponding class for each of the paths. These lists are later used to generate queries.

2.3 Preparing Queries:

Using the lists generated in the previous stage queries are formed in this stage. As in SQL, the query language supported by oracle to retrieve data from the objects contains a "Select" clause, a "From" clause and a "Where" clause. Each of these clauses is stored in a separate ordered list. The use of three ordered lists makes the resolution of complex structures such as nested tables and references is easier. Hence if n queries are generated, then each query is split into 3 parts. Query no i, i<=n, is represented by the i'th element in each of these lists. For the given user query a few generated queries are

1. Select o1.name , o1.address.street from Monuments o1
2. Select o1.name, o1.details.architect.address.street from Monuments o1
3. Select o2.name, o2.address.street from Archeology o2.

2.4 Handling Conditions:

In this stage the condition imposed by the user is incorporated into the queries generated in the previous stage. For the user query given above the following queries will be generated.

1. Select o0.name, o0.address.street from Monuments o0 where o0.address.city like '%New Delhi%'.
2. Select o0.name, o0.details.architect.address.street from Monuments o0 where o0.address.city like '%New Delhi%'.
3. Select o0.name, o0.details.architect.address.street from Monuments o0 where o0.details.architect.address.city like '%New Delhi%'.
4. Select o1.name, o1.address.street from Archeology o1 where o1.address.city like '%New Delhi%'.

2.5 Resolving Complex Structures:

The queries generated in the earlier stages are not able to retrieve data from complex structures such as nested tables, references, etc. While generating the queries, we doesn't check whether the attribute being retrieved is a nested table or a reference or any other complex structure. In this stage, we parses the queries generated and updates them, if required, thus making them capable of handling complex structures.

Let us consider the query given below

Select o0.city, o0.what_to_see from Tour o0.

The member 'what_to_see' of 'Tour' refers to a 'monument_type' object. By using the list that contains the attributes which are references, identifies that 'what_to_see' refers to a 'Tour' object and updates the query. The modified query is given below:

Select o0.city, b0.name, b0.address.street, b0.address.state,
b0.address.city, b0.address.pincode as what_to_see from Tour o0,
Monuments b0 where ref(b0) = o0.Monuemnts.

Let us consider the following queries

Select o0.name from HotelChain o0.

Select o1.address.street from Hotels o1.

Here we observes that 'Hotels' is a nested table within HotelChain and modifies the above to generate the queries given below:

Select o0.name from HotelChaino0.

Select o1.address.street from HotelChain a, table (a.Hotels) o1.

This ensures the execution of second query whenever the first query is executed, by maintaining an ordered list. With this stage the mapping of user requests into queries is completed and these queries are used in the later stages to retrieve data.

2.6 Query Resolution Order:

In this stage the queries formed in previous stages are ordered using the closeness criterion. As mentioned earlier, there can be multiple queries that can be generated for the user query. We believe, that the users prefer certain resolutions over others. If all the names are resolved to a single structure at the same level of a schema, then that resolution is the most preferred one. If that is not possible, then the next 'close' resolution is the one where all the names are in the same structure but at different levels. Within these, the resolution with lesser level difference is to be preferred over that resolution where the level difference is more. If it is not possible to resolve all the names within a single structure, then the next preference is where two or more structures have to be joined. Within these, the query resolution with lesser number of joins is more 'close' than the one with more joins. This desirability is expressed using three principles - density, level and relationship. While doing so, we distinguish between simple attributes and structured attributes. Simple attributes are those which are of basic type. The rest are structured attributes. The three principles are explained below.

Density:

The '*density*' principle states that the most 'close' resolution for a selected field is that structure in the resolution which has been referred to maximum number of times. As an example, consider the query where the user selects

city, name, phone, details

Let us assume that 'name' and 'phone' have been resolved to 'Monuments' and city has been resolved to 'Tour'. The field 'details' can be resolved to either 'Monuments' or 'Tour'. Tour has been referred to once in this query resolution whereas Monuments has been referred to twice. Therefore, resolution of details to Monuments is closer than resolution to Tour.

Level:

The '*level*' principle states that if a selected field has multiple definitions then the resolution of a name to a structure in the query which gives rise to minimum difference in levels is more close than that structure where the level difference increases. As an example, consider the query where the user selects

name, street

Let name be resolved as Monuments.name. street can be resolved as either Monuments.address.street or Monuments.details.architect.address.street. The level difference in the former between name and street is 1 whereas in the latter is 3. Therefore, the resolution where the level difference is 1 is more close than the other.

Relation:

The '*relationship*' principle states that for resolving any name in the user query, minimum number of joins between classes is more close than the resolution where the number of joins are more. The join is performed only in those cases where a foreign key is defined.

In order to incorporate 'closeness', we define three terms - Weighted count, Height, Distance. Weighted count, WC, is a value associated with every query and it incorporates the 'density' principle. Height incorporates the 'level' principle and Distance incorporates the 'relationship' principle. These terms are explained below.

The user query can have names which refer to simple attributes or structured attributes whose types can be REF, Varray, Nested table or OBJECT. The name can also refer to a table. When a corresponding query is generated, all the names in the user query which are attributes are prefixed with the names of structures to which they belong. Prefixing the name of the enclosing structure rule is repeatedly applied till a table name is encountered. That is, the complete path for accessing an attribute is identified.

Weighted count:

In order to find the weighted count, we use a 'reference count' associated with each unique table. The reference count gives the number of times a table S has been referred to in the generated query. Weighted Count (WC) is computed as follows:

Let the user query be of the form

SELECT name1, name2 .... namei

WHERE namei+1 = ' ' AND namen = ' '

Here, the number of names in the user query is n. Let the structures which are referenced in the query resolution as detailed above be S1, S2 ... Sm. Let their reference counts be C1, C2 ... Cm ordered in the descending order of reference counts. Then,

Weighted Count WC $= \sum_{i=1}^{m}(n-i+1)Ci$

WC gives a measure of the density.

Height:

In order to find the height of a query we have to construct a forest of trees for each of the generated query. The procedure to construct a forest of trees is

1. Start with the first name in the user query. If the name in a user query is the name of a structure S, then there are two possibilities. The first is where it refers to a table name. In this case create a node S and this shall be the root of the tree.
2. If it is an attribute which is a simple type then create a node S for the structure S to which it belongs and make the attribute a child of S. If it is the name of a structured attribute, then the structure can be a row, REF, a nested table etc. Refer to this as S1. Create a node S for the enclosing structure S and add a node S1 and make it a child of S.
3. In both the cases above if S is itself embedded, then recursively create a node as detailed above for the enclosing structure. The process stops when we reach a structure which is not embedded in any other structure.
4. Repeat the above process for each name in the user query.

For example, if the generated query is

```
select Tour.what_to_see.name, Tour.what_to_see.address
from Tour
where Tour.city like '%New Delhi%'
```

Then the height for name is 3, for address is 3 and for city is 2.

For each tree the depth of the tree is the height of the tree. Now, the height for a generated query is the difference between the heights of the trees and is given as

Height = Maximum height - Minimum height

Height represents the 'level' principle.

Distance:

The distance for a generated query is equal to the number of joins. Distance incorporates 'relationship' principle.

By using the principles of density, level and relationship, the queries are ordered in terms of 'closeness'. The ones which are more 'close' will be higher in the order. The resolutions which have minimum number of relationships (joins) are higher in the order of 'closeness'. Within resolutions which have the same number of joins, the lesser the difference in levels, more higher are these resolutions in the order of 'closeness'. Among those at the same level, the ones in which density is maximized are more close than others.

In terms of Weighted Count, Height and Distance, this can be translated as follows:

1 First order the generated queries according to Distance starting with minimum Distance.
2 Within queries having equal Distance, order the queries according to Height starting with minimum Height.
3 within queries with equal Height, order the queries in the descending order of Weighted Count.

In the above ordering, the query with minimal Distance and Height, and maximum WC is the most 'close' query.

Generated Query	Distance	Height	Weighted Count
Select o1.name, o1.address.street From Monuments o1 Where o1.address.city like '%New Delhi %'	0	1	8
Select o1.name, o1.details.architect.address.strret From Monuments o1 Where o1.address.city like '%New Delhi%'	0	3	6
Select o1.name, o1.details.architect.address.strret From Monuments o1 Where o1.details.architect.address.city like '%New Delhi %'	0	3	8
Select o2.name, o2.address.strret From Archeology o2 Where o2.address.city like '%New Delhi%'	0	1	8
Select o3.name, b0.address.strret From HotelChain o3, table(o3.Hotels) b0 Where b0.address.city like '%New Delhi %'	0	2	6
select o4.what_to_see.name, a.what_to_see..address from Tour a where 04.city like '%New Delhi %'	0	1	8

Fig 2.1

Here figure 2.1. gives the Weighted Count, Height and Distance for the queries generated in section 2.4. The values of density, height and relationship are used to order the queries in terms of 'closeness'.

2.7 Query Rephrasing:

In this stage we rephrase the generated SQL queries into simple English queries. The manner in which this is done is explained below.

The rephrased query is constructed in Englishquery which is initialized to "You are requesting". Thereafter, the procedure given below is executed.

For each name in the user query for which a tree is built as given by the procedure in section 2.6 do the following

```
    Start with the leaf of the tree.
    While (root not reached)
        Add name to Englishquery.
        If the leaf node is simple attribute, then concatenate 'of the' followed
        by the name of the attribute to Englishquery.
        If it is a table name then concatenate 'all the fields of' followed by the
        name of the table to Englishquery.
        If the name is part of the condition, concatenate appropriate
        comparison operator. Follow this with the value itself.
        Pick up the parent node.
    End while
```

Concatenate ' and ' to Englishquery if a name is still remaining in the select clause or in the conditional part and ' where ' if the condition is to be taken up for the first time.

Let us consider the query

```
select o1.name, o1. address.strret
from Monuments o1
where o1.address.city like '%New Delhi%'
```

Then this query is rephrased as

'You are fetching the name of the Monuments and the street of the address of the Monuments where the city of the address of the Monuments is New Delhi'.

If another generated query is

```
select o3.name, b0.address.street
from HotelChain o3, table(o3.Hotels) b0
where b0.address.city like '%New Delhi%'
```

The rephrased query is

'You are fetching the name of the HotelChain and the street of the address of the Hotels of the HotelChain where the city of the address of the Hotels of the HotelChain contains New Delhi'.

This process is executed when the user chooses to execute the query in new context.

2.8 Context:

When a user chooses a query, then the user may desire that the system maintain the structures chosen by him and use this information for reordering the generated queries of the subsequent query. If the user chooses to execute the query in old context, then the context left behind by the previous queries is taken into account to translate the user query into system query. We will order the queries using two principles - 'commonality' and 'closeness'. For a new query, the 'commonality' factor between each generated query and the previous query is computed. These are then ordered according to the commonality value with the ones with larger value higher in the order.

For example, if a query selects

name, street

And of all the generated queries

Select name, address.street from Monuments

is picked up, then a subsequent query which selects 'details' is likely to be the details of monuments rather than the details of Hotels. If the old context is to be used, then the user clicks on the old context from context list of the GUI interface of section 3.1.

Every query is executed in a context. The first query is executed in the database context which consists of all the tables with all the objects. When each generated query is executed then the context is updated which forms the context for the subsequent query. When a context is to be updated two cases arise. When a structure is referred to in the select clause or in the condition, it is added to the context. The structure may have different kinds of attributes. The types of the attributes can be

a) Simple
b) Structured
c) Varray
d) REF to a structure
e) Nested table

In the case of (a), (b) and (c) all the fields of the selected attributes are added to the context and marked as Direct. For example, if address which is of address_type is chosen then all the fields in the adrress_type are part of the context. In the case of (d) and (e), the referenced object/nested table is added to the context but marked as Indirect. The structure it self may be embedded in some other structure. The embedded structure is also added to the context but is flagged as Indirect. The rule is applied recursively till no more structures can be added.

Consider, now, the tuples that form part of the context. Two cases arise

a) There is no condition. In this case all the tuples of the direct structure and all the tuples of the indirect structure form part of the context.
b) A condition exists. The selected tuples of the direct structure form part of the context.

If the structure contains other structures then all the object/tuples of the indirect structure referenced from the direct structure are in context. If it is contained in some other structure which is an indirect structure, then all the tuples of the indirect structure which contain the selected tuples of the direct structure are in context.

Let the set of structures selected in a query Qi be Si = DS ∪ IDS, where DS = {DS1, DS2 ...DSn} and IDS ={IDS1, IDS2 ...IDSm}. In the subsequent user query, let the queries that are generated be Qj1, Qj2 ... Qjp. Let the structures referenced in these queries be Sj1 = {DSj11, DSj12 ... DSj1n}∪{IDSj11, IDSj12 ... IDSj1m}, Sj21, Sj22 ... Sj2p and so on. Then, commonality, C [Qi Qj1], between query Qi and Qj1 is

$$C\,[Qi\ Qj1] = 2 * (DSi \cap DSj1) + (IDSi \cap IDSj1).$$

The generated queries are ordered using the commonality with the one having higher commonality higher in the order. The principle of commonality has to be juxtaposed with the closeness criterion defined in section 2.6.

Let the queries that are generated be Qj1, Qj2 ... Qjp. Let the commonality computed as given above be C [Qi Qj1], C [Qi Qj2], ... C [QiQjp], respectively. The following situations arise: All queries have distinct commonalties. Reorder these according to the commonality factor. The query with maximum commonality now is the first in the list. Some queries have identical commonality. Order these according to the algorithm of section 2.6. That is, the WC, height and relationship are the criterion for ordering these queries with identical commonality.

For example if the user selects 'details' from the select list and chooses to execute the query in old context the queries generated are shown in figure 2.2 below.

	Commonality	Distance	Height	Weighted Count
Select b0.details From HotelChain o2, table(o2.Hotels) b0	0	0	0	1
Select o1.details From Monument o1	2	0	0	1

Fig 2.2

Both the queries have the same 'closeness'. However, since the second query has commonality 2, it will be the first in the order of the generated queries.

2.9. Displaying Results:

This stage involves two stages namely retrieving data and displaying data. These two stages are explained below.

Retrieving Data:

In this stage, the queries formed in the previous stages are used to retrieve data from the database. First we will establish the connection with the database and stores the retrieved data. Initially, the interface requests the database server for connection. Once the connection is established the queries formed are used to retrieve the data from the database. The retrieved data is stored in a data structure. In this implementation the data retrieved is stored in an ordered list (array).

Displaying Data:

In this stage, the data retrieved is displayed in a user-friendly format. Let us consider the query

```
Select o1.name, o1.address.street from Monuments o1
Where o1.address.city like '%New Delhi%'.
```

By scanning this query, we will prepare an ordered list of strings whose contents are given below:

Monuments

address (2-5)

name, street

This list determines the way in which the data displayed should be labeled. Each element in the list represents a row on the screen. The first row contains the label 'Monuments', the second row contains 'address' from column 2 to column 5 and the third row contains the attributes whose values are being displayed. This format resembles the actual structure of the data in the database.

The queries containing Sets and references are also dealt in a similar fashion. Chapter 3 pictorially depicts the way the data is displayed for different types of queries.

CHAPTER 3

GRAPHICAL USER INTERFACE

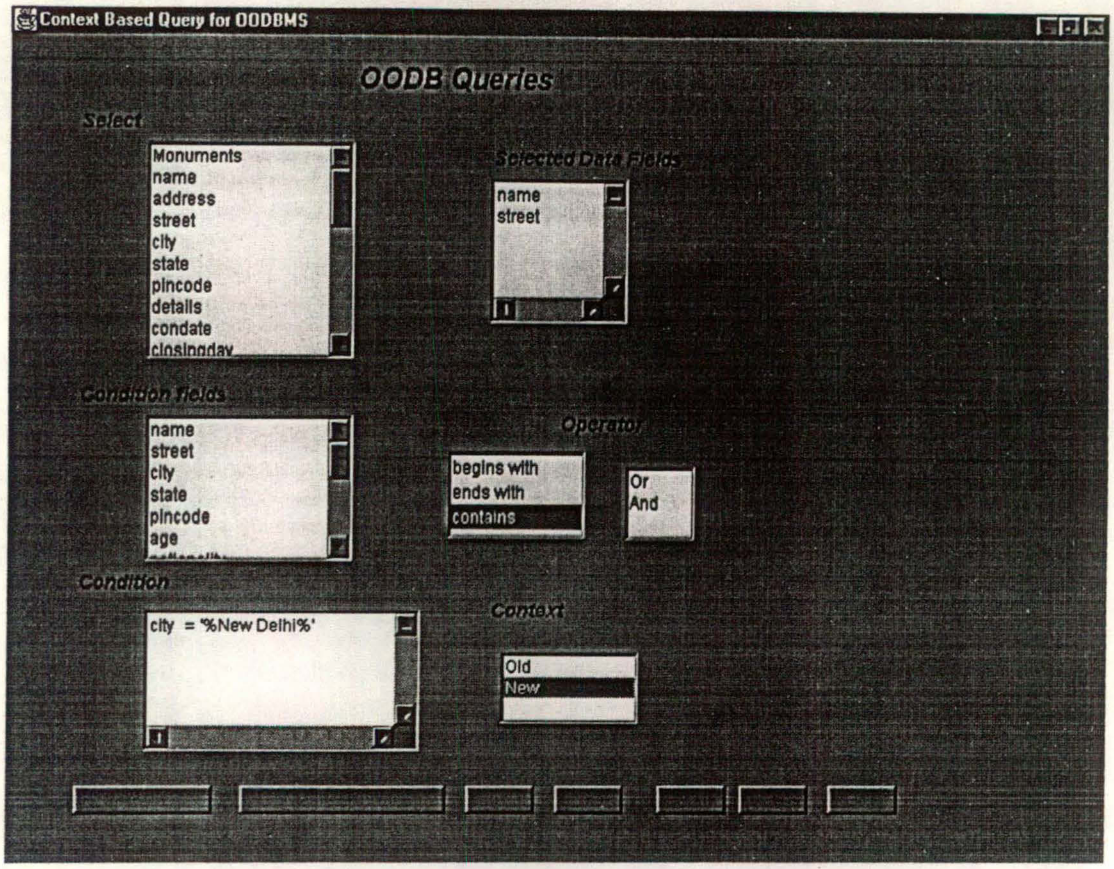
In this chapter we discuss about GUI Query Interface design. The Query interface allows the user to select the data needed by him and impose any conditions, if any, on the data selected by him. It also provides the interface to display the results, that is, the data retrieved by the user query.

3.1 The Query Interface:

The system provides the user a graphical user-friendly interface for building the query. This interface allows the user to retrieve the data that is needed by him. This chapter explains the way the user interacts with the interface. It also explains how the retrieved data is displayed to the user.

In the figure 3.1, the *Select* box is a list, which displays the schema of the database without its structure. Here the users need not to remember all the attributes of the schema. The user will not know that 'architect' is an object and the address 'address' of the 'architect' is another object within that 'architect' object. The members with same names are not repeated in the select box to avoid confusion for the user. For example, though 'name' is the name of a member in 'monument_type' and 'hotel_type', it is listed only once. No distinction is made either for references or nested tables. Hence the user will not know that 'phone' is a varray and 'what_to_see' refers to some other object.

The *Condition fields* box is a list that displays the conditional attributes used to impose a condition on the data he wants to retrieve. To build the condition, operator box provides the relational operators like less than (<), greater than (>), less than or equal to (< ==), greater than or equal to (>==), and logical operators like AND and OR operators.



Selected Data Fields: name, street

Condition: city = '%New Delhi%'

Fig 3.1

The *context* box is a list that is used to choose the context of the query, i.e. old context or new context. The *selected data fields* box is a text area used to display the selected data fields. The *condition* box is a text area used to display the condition.

At the bottom of the screen the user will be provided several buttons. The purpose of each button is explained below.

Clear condition: This buttons helps the user to clear the condition specified.

Clear query: The user can use this button to clear the query i.e. selected data fields.

Exit: This button helps the user to exit from the system.

Save: This button will be used to save the selected data fields and the condition specified by the user that is the query, in a local system .

H*elp*: This button helps the user to understand the controls of the screen.

Submit: This button is used to submit the query for further processing.

Open: This is a button. The saved query with condition specification store in file is opened. The data is fetch from the file and is displayed in attribute textbox and condition textbox. data is fetch from the file and is displayed in attribute textbox and condition textbox.

When the user clicks on 'Submit' button after specifying the condition the system starts processing the user requests. As mentioned before the system maps the user requests into one or more system queries, which reference the nested structures explicitly. The system uses these queries to retrieve data from the database.

The remaining part of this section, through some examples, illustrates how the system displays data retrieved when the user requests include complex structures such as sets, nested objects etc.

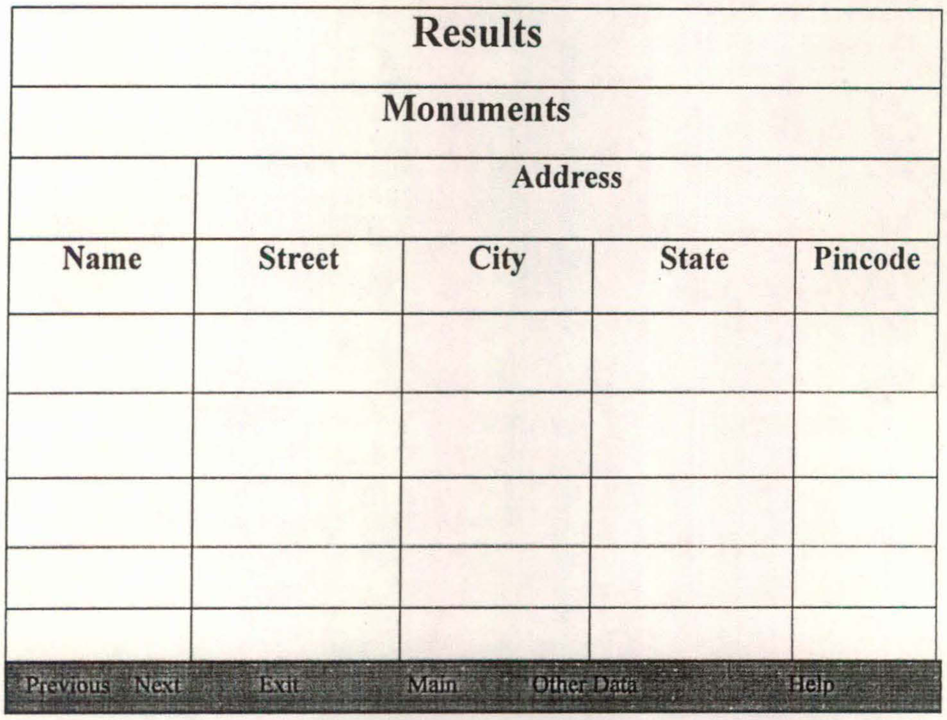
3.2 Path Resolution:

Let us consider the case when the user selects 'name' and 'address'. The database used in this implementation is Oracle. The queries to retrieve 'name' and 'address' in the query language supported by Oracle are given below.

1. Select o0.name, o0.address.street, o0.address.city, o0.address.state, o0.address.pincode from Monuments o0
2. Select o0.name, o0.details.architect.address.street, o0.details.architect.address.city, o0.details.architect.address.state, o0.details.architect.address.pincode from Monuments o0
3. Select o1.name, o1.address.street, o1.address.city, o1.address.state, o1.address.pincode from Archeology o1.

From the above queries it can be seen how the nesting of objects is made transparent to the user. The user doesn't have to mention the path 'o0.details.architect.address.street' for retrieving the 'street' where the architect of the monument lives. Just a click on the 'street' will serve the purpose for the user. The user even need not know about the existence of 'monument' objects in the database. Thus the schema of the existing database is made transparent to the user.

The result for the query number. 1 given above is displayed in the following manner.



Results				
Monuments				
	Address			
Name	Street	City	State	Pincode

Fig 3.2

The format, in which the data is displayed, represents the structure of the data. In figure 3.2 the label "Monuments" indicates that 'name' and 'address' are members of 'Monuments' object. Similarly the label "address" indicates that 'street', 'city', 'state' and 'pincode' are members of 'address' object.

At the bottom of the screen there are six buttons. The user uses these buttons to interact with the system. The purpose of each of these buttons is explained below:

Previous & Next: If the user query retrieve a large amount of data, it will not be possible to display the entire data at a time. These buttons help the user to go through the retrieved data.

Exit: The user can use this button to exit the application.

Main: On clicking this button the user will encounter the initial screen depicted in Fig.1 so that he can form the query again.

Other Data: This button will be used to display data retrieved by the execution of the next query in the set of queries generated.

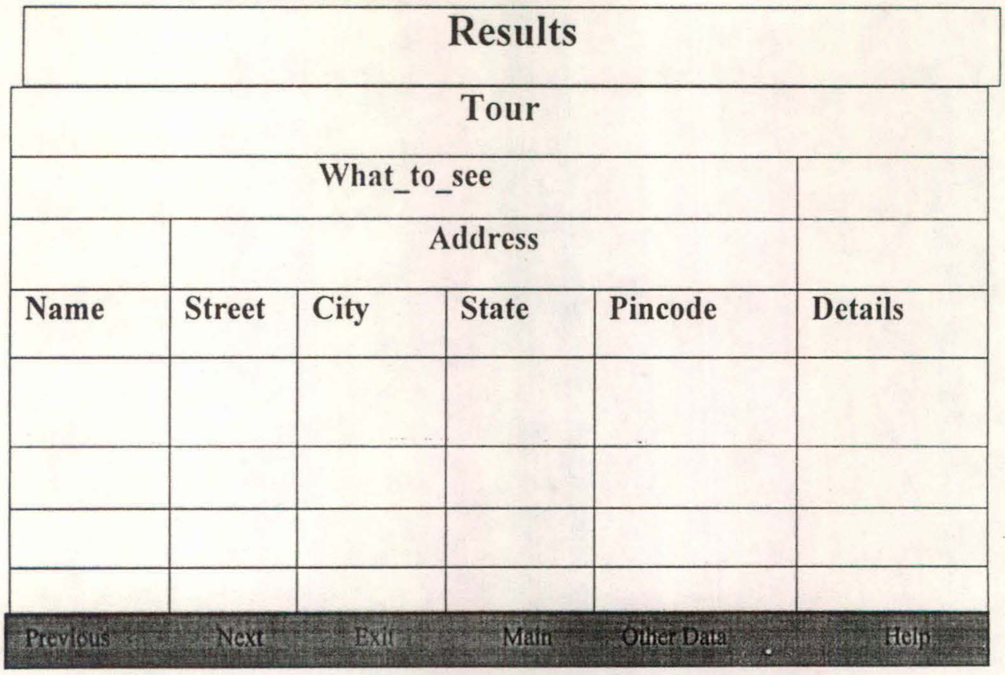
Help: This button helps the user understand the format in which the data is displayed.

3.3 Resolving References:

Let us consider the case when the user selects 'what_to_see' and 'details' in 'Tour'. The member 'what_to_see' of 'Tour' refers to a 'Monument' object. The query generated in this case are given below:

```
Select b0.name, b0.address.street, b0.address.city, b0.address.state,
b0.address.pincode, o0.details from Tour o0, Monuments b0 where ref (b0) =
o0.what_to_see.
```

The data retrieved by this query is displayed in figure 3.3.



Results					
Tour					
What_to_see					
	Address				
Name	Street	City	State	Pincode	Details

Fig 3.3

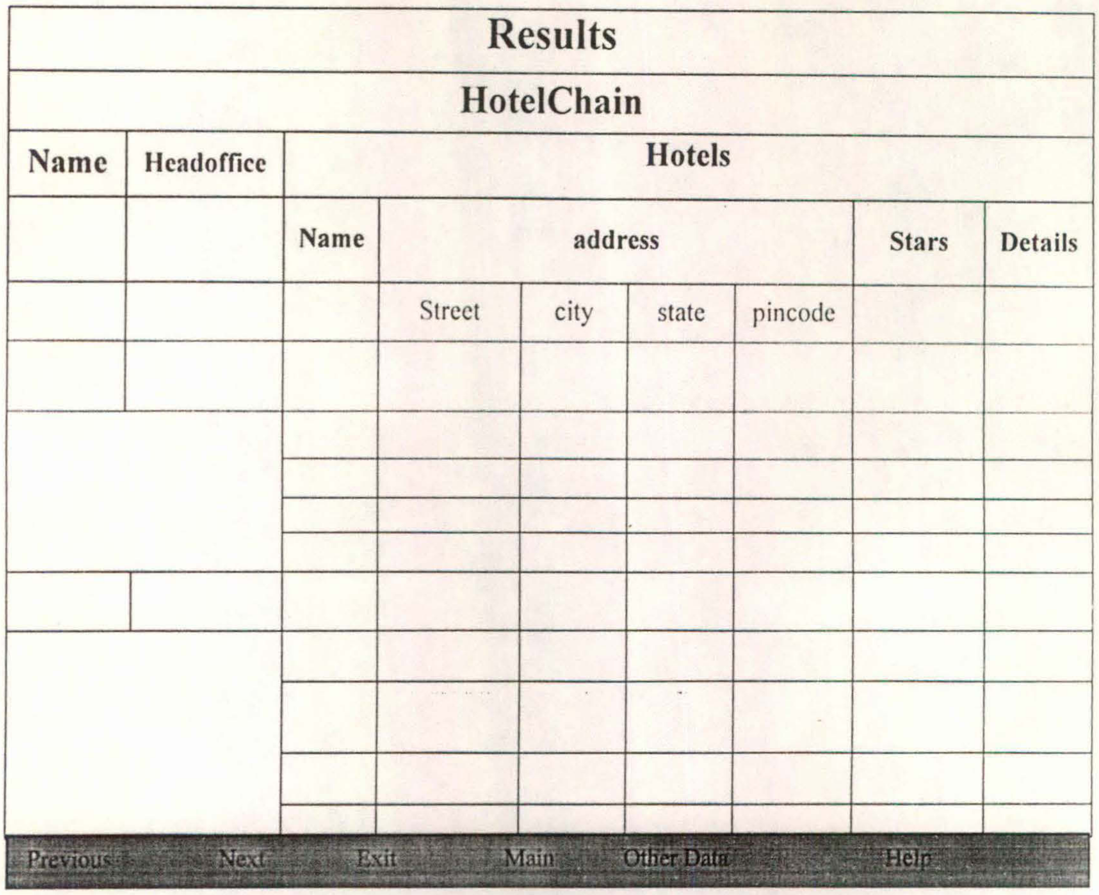
In the above figure 3.3 the structure of the data describes 'name' and 'address' are the members of the 'Monument' object to which the member 'What_to_see' of 'Tour' points.

3.4 Handling Nested Tables:

Let us consider the case where the user selects the members 'name', 'headoffice', and 'Hotels' of 'HotelChain'. The member 'Hotels' in 'HotelChain' is a set of objects (considered as nested table). The queries generated to retrieve the data are given below:

1. Select o0.name, o0.headoffice from HotelChain o0.
2. Select o1.name, o1.address.street, o1.address.city, o1.address.state, o1.address.pincode, o1.stars, o1.details from HotelChain a, table (a.Hotels) o1.

The data retrieved by these two queries is displayed in the following manner.



Results								
HotelChain								
Name	Headoffice	Hotels						
		Name	address				Stars	Details
			Street	city	state	pincode		

Fig 3.4

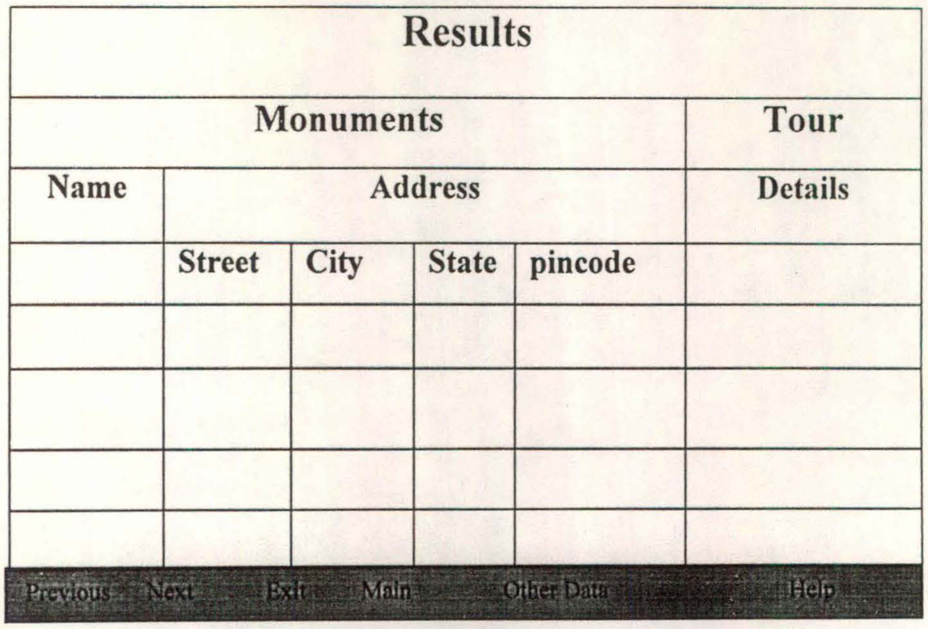
In the above figure 3.4, ‘name’, 'headoffice' and ‘Hotels’ are the members of ‘Hotels’. Each HotelChain has a number of Hotels.

3.5 Handling Relationships:

Let ‘name’, ‘address’ and ‘details’ be the attributes selected by the user. ‘name’ and ‘address’ are members of ‘Monuments’ and ‘details’ is the member of ‘Tour’. The system explores the relationship between these two objects by generating the following query

```
select o0.name, o0.address.street, o0.address.city, o0.address.state,
o0.address.pincode, o1.details from Monuments o0, Tour o1
where o0.address.citry=o1.city.
```

The data retrieved will be displayed in the following manner:



Results					
Monuments					Tour
Name	Address				Details
	Street	City	State	pincode	

Fig 3.5

As in previous cases, the correspondence between data requested and data retrieved is apparent. In figure 3.5 'name' and 'address' are members of 'Monuments' and 'details' is the member of 'Tour'. This is similar to 'join' in relational databases.

3.6. Transparency:

The transparency achieved through this interface is described below:

1. The user doesn't have to know about the schema of the database.
2. In object oriented databases objects may contain other objects. To retrieve data from these component objects the user should have knowledge about their container objects. This interface doesn't expect the user to have such knowledge and provides him with the required data.
3. The complex structures such as nested tables, varrays etc. and references to other objects are made transparent to the user.
4. The user need not have knowledge even about the object-oriented concepts to retrieve data from the database using this interface.

CHAPTER 4

DESIGN

In this chapter we discuss the Overall System Architecture and design of the project using structural charts.

4.1 Overview of System Architecture:

The system is based on a two-tier architecture. In this architecture there will be one or more clients and a server, which responds to the queries of those clients.

The figure 4.1 shows the architecture of the system.

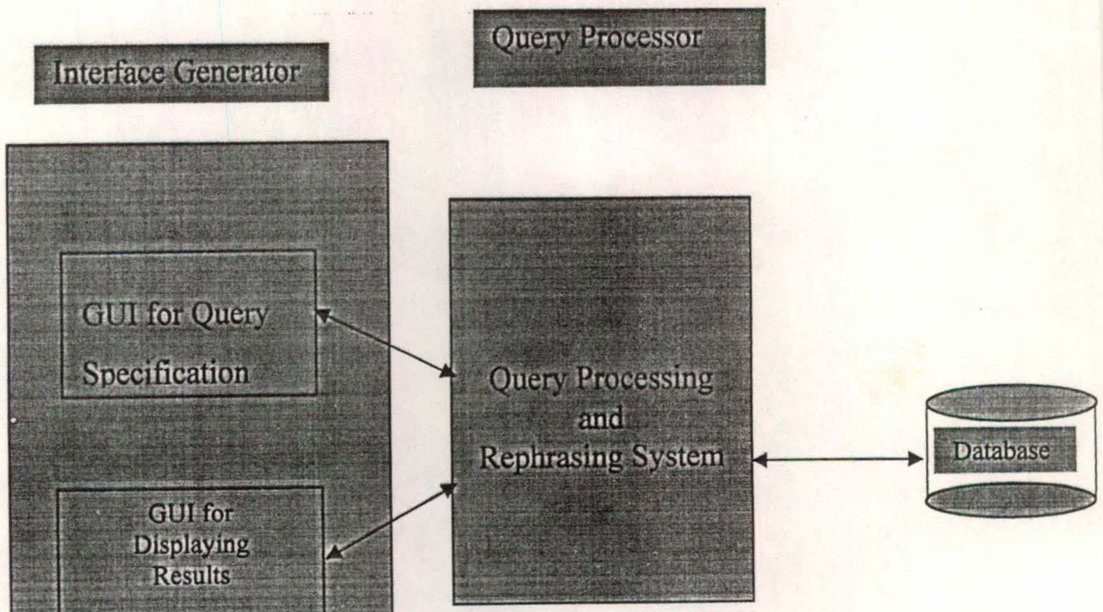

Interface Generator
Query Processor
GUI for Query Specification
GUI for Displaying Results
Query Processing and Rephrasing System
Database

Fig 4.1

In the system designed, a front-end application acts as client. It generates queries, which correspond to the user requests and send them to the back-end database server. The database server sends the retrieved data to the client, which then displays it to the user. *Concurrency,* one of the features of object oriented database systems, enable many users (clients) to simultaneously access the database.

The design details are explained in the following sections.

GUI for Query Specification is a user friendly GUI is provided for the user to develop the query for the system. Here the users need not to remember the schema of the database. The user can form complex queries using logical connective like AND and OR. Condition using arithmetic comparisons greater than (>), less than (<), equals to (=) can also be performed. The submitted query would be given to the query processing and rephrasing system to get the required data.

In **Query Processing and Rephrasing System** query given by the user through GUI will be interpreted in its context. It the user chooses to execute the query in a new context, then the user query is translated into one or more system queries, which references the nested structures explicitly. In general there can be multiple queries that can be generated for the user query. The closeness criterion is used to order the generated queries with the one more closer higher in the order.

If the user chooses to execute the query in old context, then the context left behind by the previous queries is taken into account to translate the user query into system query. If more than one such query can be generated, then these are, as before ordered according to the closeness criterion.

In either case the system queries are rephrased using English like syntax. The rephrased queries are presented to the user. The user selects one or more for execution. The results are displayed to the user.

In **Displaying Results,** results obtained from the query processing system are presented to the user, which represents the structure of the data.

4.2 The Design:

This interface is implemented in Java. Oracle is used as the backend. There are two main modules in this implementation namely "Project" module and "Database connection". The detailed design of this system is given in this section.

"Project" module controls the interaction between the user and the interface. The interface allows the user to perform some actions such as selecting the data needed by him and imposing conditions, if any, on the data selected by him. The system interprets the user query into one or more queries, which reference the nested structures explicitly. In general there can be multiple queries generated for the user query. The closeness criterion is used to order the generated queries with the ones more closer higher in the order. These queries are rephrased using English like syntax. The rephrased queries are presented to the user. The user selects one or more, which handed over to "Database connection" module for further processing.

"Database connection" module establishes a connection with the database. It then sends the queries generated by project to retrieve data from the database. It also presents the data retrieved to the user in an orderly and desirable manner.

The figure 4.2 is the structure chart depicting the overall system design. The module "Main" coordinates the execution of "Project" module and "Database connection" module.

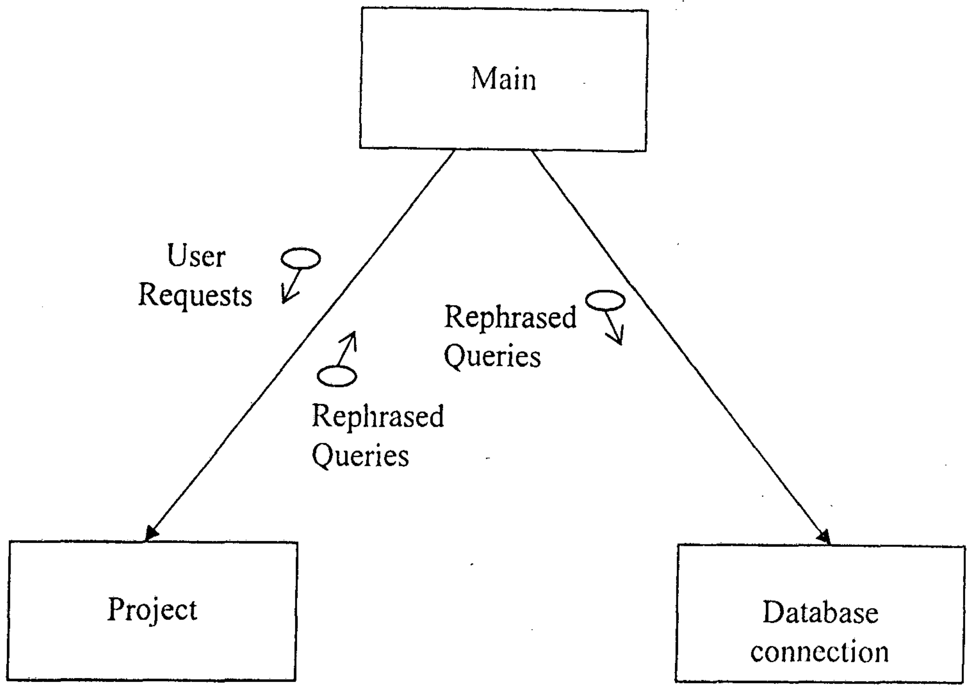


Fig 4.2

The detailed design of each of these modules given through structure charts is shown below.

4.3 Structure Chart for Query Processing:

The "Project" module prepares the interface so that the user can interact with it and loads the schema on to the interface in a user-friendly format. The figure 4.3 represents the structure chart for Query processing that consists of three sub modules in it namely, "GUI for Building Query", "Query Resolution" and "Query Rephrasing".

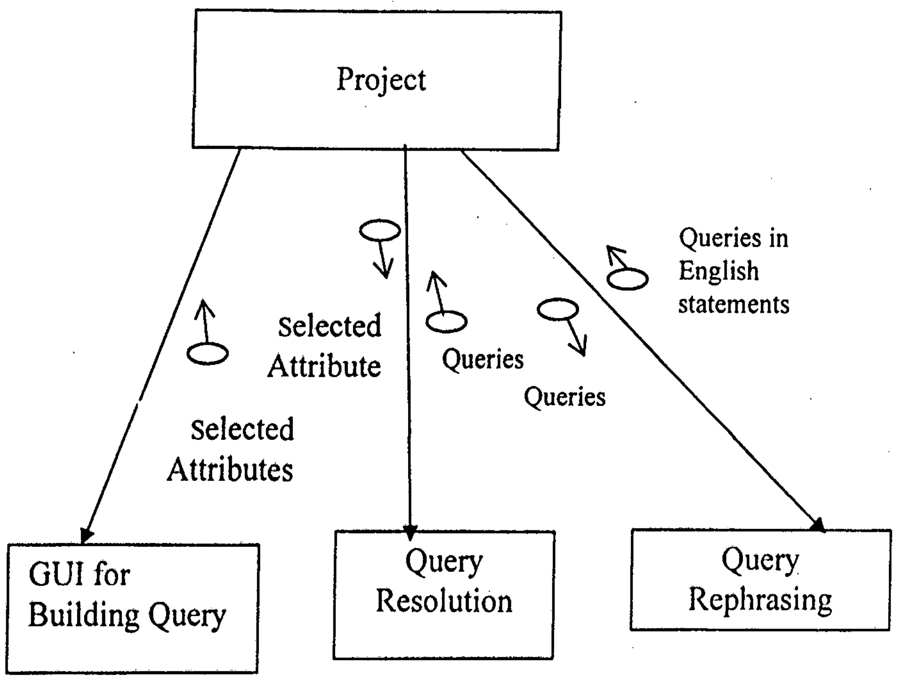


Fig 4.3

The module "GUI for Building Query" gives the user-friendly interface to select the required information from the select list and condition specification from the condition attribute list.

The module "Query Resolution" finds the classes of the selected attributes, resolves their paths, prepares the queries, handles the complex structures and order the queries using the closeness criterion. The module "Query Rephrasing" accepts the ordered queries and rephrased using English like syntax.

4.4 Structure Chart for Query Resolution:

The figure 4.4 represents the structure chart for Query Resolution, which consists of five sub modules namely "Finding Classes", "Resolving Paths", "Preparing Queries", "Handling Complex Structures", and "Ordering Queries". These are discussed in detail below.

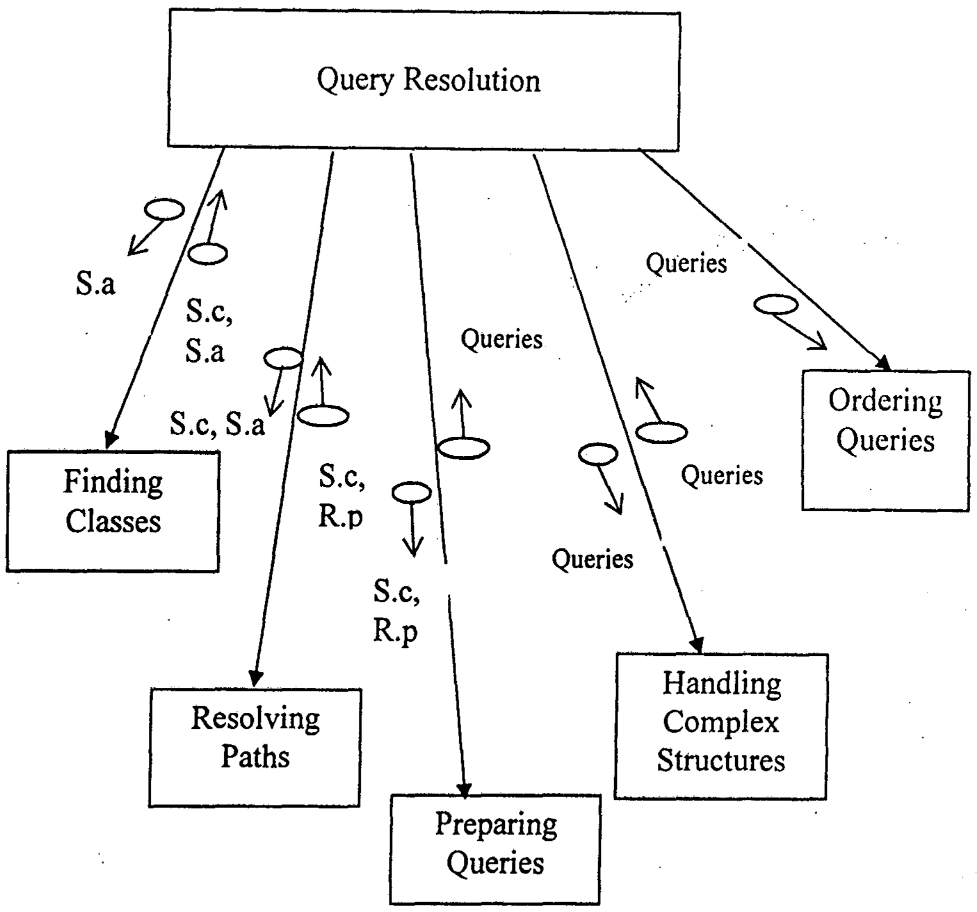


Fig 4.4

In the above structure 'S.a' stands for set of selected attributes, 'S.c, S.a' stands for set of classes corresponding to the selected set of attributes and 'S.c, R.p' stands for set of classes with the resolved paths for each of the selected attributes.

The module "Finding Classes" accepts the set of attributes selected by the user and finds out one or more classes to which each of these attributes belongs. The module "Resolving Paths" performs the path resolution. The module "Preparing Queries" prepares the query statements. The module "Handling Complex Structures" resolves the complex structures in queries. The module "Ordering Queries" order the generated queries using closeness criterion with ones more closer higher in the order.

4.5 Structure Chart for Displaying Results:

The figure 4.5 represents the structure chart for displaying results that consist of two sub modules namely "Get Data" and "Display". These are discussed in detail below.

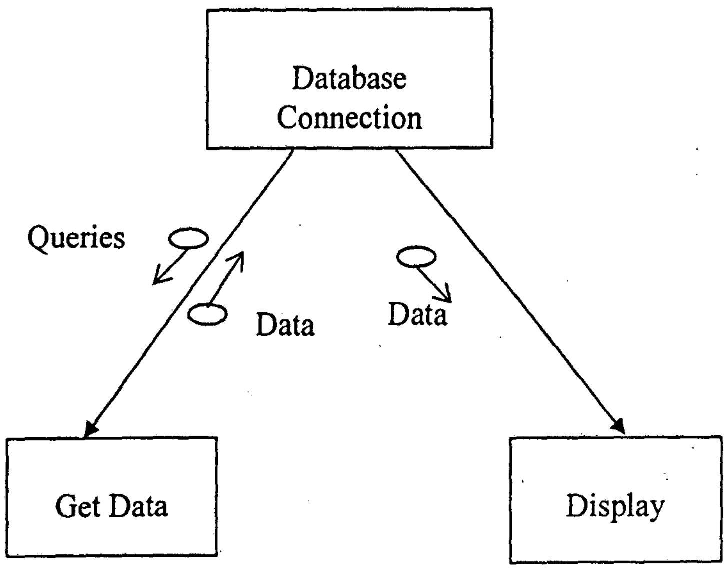


Fig 4.5

The module "Get Data" is used to retrieve the data from the database. The module "Display" displays the data retrieved. All these modules are explained in chapter 5.

CHAPTER 5

IMPLEMENTATION

This chapter explains how the input given by the user is transformed into the output that the user finally views. In our implementation we used Java to provide a user friendly environment and oracle is used as the backend to store data and respond to the queries sent.The various stages involved in this process are
query resolution, query resolution order, query rephrasing and displaying results.

5.1 Query Processing:

This stage involves various modules namely Finding out classes, Resolving the paths, Preparing queries, Handling conditions and resolving complex structures. All these modules are explained in detail below.

The module "Finding Classes" searches the database schema to find out the classes in which the selected attributes are members. This search yields a set of classes for each of the selected attributes such that each attribute is a member of each class in the set corresponding to that attribute. If a1, a2 and a3 are the attributes selected by the user then s1, s2 and s3 are the sets of classes such that a1 is a member of each class (or one of its component) in s1 and so on. Let 'name' and 'city' be the attributes selected by the user. The module "Finding Classes" on searching the schema, finds that the attribute 'name' is mentioned in 'monuments', 'archeology', and 'HotelChain' say set s1, and the attribute 'city' is mentioned in 'monuments', 'archeology', 'HotelChain' and 'Tour', say set s2.

In the previous module, a set of classes is identified for each attribute. Let C be an element in the set of classes for an attribute A. A may be a member of C or a member of a component class of C or a member of a component class of component class of C and so on. In this context, path of the attribute A describes the location of

A in C. Recognizing this path is necessary to generate the query to retrieve the attribute A. This process can be termed as "Path resolution".

Let us consider 'name' and 'city' as the attributes selected by the user. The module "Resolving Paths" performs the path resolution. Let o1 be an object of 'Monument', o2 be an object of 'HotelChain', o3 be an object 'Archeology' and o4 be an object of 'Tour'. The paths generated for 'name' are: o1.name, o2.name, o3.name,o3.Hotels.name and the paths generated for 'city' are: o1.address.city, o1.details.architect.address.city, o2.address.city, o3.address.city, o4.city

The module "Resolving Paths" on scanning the schema outputs two lists. One list contains the resolved paths for the selected attributes and the other list contains the corresponding class for each of the paths. These lists are later used to generate queries.

The module "Preparing Queries" using the lists generated by the module "Resolving Paths" queries is formed in this stage. As in SQL, the query language supported by oracle to retrieve data from the objects contains a "Select" clause, a "From" clause and a "Where" clause. Each of these clauses is stored in a separate ordered list. The use of three ordered lists makes the resolution of complex structures such as nested tables and references is easier. Hence if n queries are generated, then each query is split into 3 parts. Query no i, i<=n, is represented by the i'th element in each of these lists.

Let the 'name' and 'city' as the selected attributes, then the few generated queries are

1. Select o1.name , o1.address.city from Monuments o1
2. Select o1.name, o1.details.architect.address.city from Monuments o1
3. Select o2.name, o2.address.city from Archeology o2.

In this stage the condition imposed by the user is incorporated into the queries generated. The process of finding the classes and resolving paths is same for these attributes. The module "Handle Condition" handles the condition imposed by the user. The interface doesn't impose any restriction on the number of conditions that the user can impose.

Let 'name' and 'street' be the attributes selected by the user. Let city contains 'New Delhi' be the condition imposed by the user. The module "Handle Condition" generates the following queries to satisfy these requests

1. Select o0.name, o0.address.street from Monuments o0 where o0.address.city like '%New Delhi%'.
2. Select o0.name, o0.details.architect.address.street from Monuments o0 where o0.address.city like '%New Delhi%'.
3. Select o0.name, o0.details.architect.address.street from Monuments o0 where o0.details.architect.address.city like '%New Delhi%'.
4. Select o1.name, o1.address.street from archeology o1 where o1.address.city like '%New Delhi%'.

The queries generated in the earlier stages are not able to retrieve data from complex structures such as nested tables, references, etc. While generating the queries, the module "Preparing Queries" doesn't check whether the attribute being retrieved is a nested table or a reference or any other complex structure. In this stage the module "Resolve Complex Structures" parses the queries generated and updates them, if required, thus making them capable of handling complex structures.

Let us consider the query given below

Select o0.city, o0.what_to_see from tour o0.

The member 'what_to_see' of 'tour' refers to a 'monument_type' object. The module "Resolve Complex Structures", uses the list that contains the attributes which are references, identifies that 'what_to_see' refers to a 'tour' object and updates the query. The modified query is given below:

Select o0.city, b0.name, b0.address.street, b0.address.state, b0.address.city, b0.address.pincode as what_to_see from tour o0, Monuments b0   where ref(b0) = o0.Monuemnts.

Let us consider the following queries

Select o0.name from HotelChain o0.

Select o1.address.street from Hotels o1.

The module "Resolve Complex Structures" observes that 'Hotels' is a nested table within HotelChain and modifies the above to generate the queries given below:

Select o0.name from HotelChaino0.

Select o1.address.street from HotelChain a, table (a.Hotels) o1.

The module "Resolve Complex Structures" ensures the execution of second query whenever the first query is executed, by maintaining an ordered list. With this stage the mapping of user requests into queries is completed and these queries are used in the later stages to retrieve data.

The module "Ordering Queries" ordered the queries formed in previous module using the closeness criterion. The procedure for ordering queries is explained below.

1 First order the generated queries according to Distance starting with minimum Distance.
2 Within queries having equal Distance, order the queries according to Height starting with minimum Height.
3 Within queries with equal Height, order the queries in the descending order of Weighted Count.

In the above ordering, the query with minimal Distance and Height, and maximum WC is the most 'close' query.

The module 'Query Rephrasing' rephrase the generated SQL queries into simple English queries. The manner in which this is done is explained in section 2.7.

If the generated query is

```
select a. name, a. address
from Monuments a
where a.address.city like '%New Delhi%'.
```

Then the query is rephrased as

'You are fetching the name of the Monuments and the address of the Monuments where the city of the address of the Monuments is New Delhi'

5.2. Displaying Results:

This stage involves two modules namely Get Data and Display. These two modules are explained below.

The module 'Get Data' establishes the connection with the database and stores the retrieved data. Initially, the interface requests the database server for connection. Once the connection is established the queries formed are used to retrieve the data from the database. The retrieved data is stored in a data structure. In this implementation the data retrieved is stored in an ordered list (array).

The module 'Display' scans the query and prepares the image of the screen that should be displayed to the user. Let us consider the query

```
Select o0.name, o0.address.street, o0.address.city, o0.address.state,
o0.address.pincode from Monuments o0.
```

By scanning this query, the module 'Display' prepares an ordered list of strings whose contents are given below:

```
Monuments
address (2-5)
name, street, city, state, pincode
```

This list determines the way in which the data displayed should be labeled. Each element in the list represents a row on the screen. The first row contains the label 'Monuments', the second row contains 'address' from column 2 to column 5 and the third row contains the attributes whose values are being displayed. This format resembles the actual structure of the data in the database.

The queries containing Sets and references are also dealt in a similar fashion. Chapter 3 pictorially depicts the way the data is displayed for different types of queries.

5.3. The Platform:

Java is used in designing the graphical user interface and Oracle is used to store the data and respond to the queries sent. Windows 2000 Professional is the operating system on which this application is developed.

Java language is created by Sun Microsystems by adapting essential features of C++ and removing the complexities like pointers in it. The Java programming language platform provides a *portable, interpreted, high-performance, simple, object-oriented* programming language and supporting run-time environment. The main reason for choosing Java is its built in support for designing graphical user interfaces. Java consists of advanced swing packages, which provides complex structures like table structure views and Abstract Window Toolkit (AWT) for designing graphical user interface. It provides net connectivity to access and retrieve the data. It provides iconified buttons and many widgets for user friendly to use. This also provides good look and feel of Java components. Apart from these the execution time of the programs is very less on this platform.

Java is an object oriented programming language and hence carries all the advantages of object oriented programming such as Encapsulation, Polymorphism and Inheritance. The unique feature of Java is its portability. Programs written in Java are platform independent because of the 'bytecode' concept introduced by the Sun Microsystems. Java programs on compilation will be converted into bytecodes. These bytecodes will then be executed by the Java Virtual Machine (JVM). The JVM can be implemented either in the hardware or software on a particular machine. Hence the statement 'compile once, run any where'.

Oracle is not a complete object oriented database. But it allows the creation of complex objects and the existence of complex structures such as references and nested tables. It also supports a query language, which can be used to retrieve data from the objects stored. Hence Oracle is used in this implementation to store the data.

CHAPTER 6

CONCLUSION

In this thesis we developed a user-friendly query interface for object oriented databases. Here the user will be provided with graphical user interface for building the query. In this system the user need not be aware of what information that exists in each database. Even user need not to specify explicitly the databases while submitting query. After submission of query, every query is interpreted in its context.

If the user chooses to execute the query in a new context, this application begins processing by finding out for each attribute the set of classes in which it is a member. It then performs the path resolution that involves finding out the location or the depth of each attribute in the class. After that queries are prepared. The queries formed at this stage are modified to make them capable of retrieving data from complex structures. After that these queries are ordered using the closeness criterion with the ones more close higher in the order.

If the user chooses to execute the query in old context, then the context left behind by the previous queries is taken into account to translate the user query into system query. For a new query, the 'commonality' factor between each generated query and the previous query is computed. These are then ordered according to the commonality value with the ones with larger value higher in the order. Then these queries are rephrased using English like syntax. The rephrased queries are presented to the user. The interface uses these queries to retrieve the data and the retrieved data is displayed to the user.

Unlike the query language, this interface doesn't require the user to have prior knowledge of the syntax of the language, object oriented concepts and database schema. Hence this interface will be extremely useful to the users who don't have knowledge about databases and object oriented concepts. By encapsulating the functionality of the query language this interface make various intricacies associated with the query language transparent.

The main purpose of this implementation is to provide the user an easy way to retrieve the data from the database. The scope for extending this application lies in adding the capability to modify and update the database.

REFERENCES

[1]. BUILDING AN OBJECT ORIENTED DATABASE SYSTEM – THE STORY OF 02 -- Francois Bancilhon, Claude Delobel, Paris Kanellakis(eds.)

[2]. The following URLs give introductory information about object oriented databases.

http://misdb.bpa.arizona.edu/~mis696g/Reports/ObjectDB/oodb.html

http://www.aiai.ed.ac.uk/project/plinth/oodb/what.html

[3]. The user manual for OQL can be obtained from the web site

http://www.cis.upenn.edu/~cis550/oql.pdf

[4]. The URL given below give information about OQL tutorial

http://www.db.ucsd.edu/People/michalis/ notes/O2/OQLTutorial.htm

[5]. SQL-3 Implementing the Object Relational Database
Dr. Paul J. Fortier

[6]. The URL given below give information about SQL3

http://www.objs.com/x3h7/sql3.html

[7]. Java 2 platform - Jamie Jaworski