

**A USER FRIENDLY INTERFACE
FOR
OBJECT ORIENTED DATABASES**

Dissertation submitted in partial fulfillment
of the requirements for the
award of the degree of

Master of Technology

in

Computer Science

by

Srikanth Goteti



SCHOOL OF COMPUTER & SYSTEMS SCIENCES

JAWAHARLAL NEHRU UNIVERSITY

NEW DELHI – 110067

December 2001

005.11 TH
G711 Us
TH9476

CERTIFICATE

This is to certify that the project entitled "A user friendly interface for object oriented database systems" being submitted by Srikanth Goteti to the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science is a bonafide work carried by me under the guidance and supervision of Prof. Parimala N.

The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.

G. Srikanth
(G Srikanth) 20/12/01

Parimala N.
Prof. Parimala N. 20/12/01

School of Computers and Systems Sciences,
Jawaharlal Nehru University,
New Delhi-110067.



Prof. K.K. Bharadwaj
Dean,
School of Computers and Systems Sciences,
Jawaharlal Nehru University,
New Delhi-110067.

ACKNOWLEDGEMENTS

I am grateful to my guide, Prof. Parimala N. for suggesting me to do the work in object oriented databases 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 Mr. T.V. Vijay Kumar, research scholar, SC&SS, and all my batch mates for their cooperation and advice.

G. Sukanth
(G Srikanth) 20/12/01

ABSTRACT

Object oriented database systems are software systems integrating techniques from databases, object-oriented languages, programming environments and user interfaces. These 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. However even to use the query language the user must know the underlying database schema.

We propose a user-friendly interface to the object oriented database system. In this system the users need not have any knowledge either about the schema of the database that they are using or about any query language that the system supports. This transparency is achieved as follows:

1. The user doesn't have to know the class of the object in which the desired data exists. This implies that the 'from' clause in the general SQL query structure is eliminated.
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, sets etc. and references to other objects are made transparent to the user.
4. The interface is made intelligent enough to find out the relation between data in different objects so that the user can retrieve data in most desirable form.

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 maps these requests of the user into meaningful queries of the query language supported by the database being used. The system uses these queries to retrieve data from the database and presents this data to user.

The interface is designed in Java and Oracle is used as the backend to store data and to respond to the queries sent.

CONTENTS

CERTIFICATE	i
ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
1. INTRODUCTION	1
2. THE GRAPHICAL USER INTERFACE	9
3. DESIGN	25
4. IMPLEMENTATION	33
5. CONCLUSION	46
REFERENCES	48

CHAPTER 1

INTRODUCTION

1.1 Object Oriented Database Systems:

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

Object oriented database systems are software systems integrating techniques from databases, object-oriented languages, programming environments and user interfaces. An object oriented database management system should be

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

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

Complex objects are built from simpler ones. An object oriented database should support sets, tuples and lists. Sets are a natural way of expressing collections in the real world. Tuples capture the properties of an entity. Lists or arrays depict 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 behaviour 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 (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.

A *type* summarizes the common structure of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. 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 specialised 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* can lead to a hierarchy of classes with each class at a level being a more specialised version of the one at the immediately higher level.

Overloading and *Late binding* occur due to *Polymorphism*. They are explained below.

Because of *overloading*, an object can support many operations that have the same name but each of these operations corresponds to a different

implementation. This implies that many methods can be declared having the same name, as long as their invocations (messages) can be distinguished by their signatures, consisting of the no and types of their arguments.

A name, say O, may denote objects of many different classes that are related by a common superclass (due to multiple inheritance). An operation defined in the superclass can be defined again in one or more sub classes with a different implementation. Since O can denote any of these subclasses, different responses can be obtained for a common set of operations. This leads to *dynamic* or *late binding*.

Computational completeness means that any computable function can be programmed using the system. Though this concept is obvious from programming language point of view, it is new to the database systems.

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.

Persistence is the ability of data to survive the execution of a process and to be eventually reusable in another process.

Disk Management includes the classical features one finds in a DBMS such as index management, data clustering and data buffering.

Concurrency means that the system should ensure harmonious cooperation between users working simultaneously on the database. The system should therefore support the standard notion of atomicity of a sequence of operations and of controlled sharing.

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

The system should also provide an ad hoc *query* facility to the user. The query facility should be

1. *High Level*: The query facility should be reasonably declarative concentrating on what rather than on how.
2. *Optimizable*: The formulation of queries should lend itself to some sort of optimization.
3. *Application Independent*: The query facility should not be targeted to a specific application.

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

1.2 Query Language 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.

As mentioned in the previous section, a query language that provides ad hoc query facility in object oriented database systems should be *high level*, *optimisable* and *application independent*. These characteristics ensure efficient and convenient access to the database.

This section, using example queries, explains how two popular object oriented query languages, OQL and SQL3, facilitate the retrieval of data from the objects stored in the database.

1.2.1 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 (Country: string,
 City: string,
 Street: string)
 Description: 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)

An object of type Place_to_go is created by the statement below

Object Eiffel_Tower: Place_to_go

Example 1:

To know all the activities that cost less than 10 dollars, query that should be used is

```
Select x.name  
From x in Thing_to_do  
Where x.fee < 10
```

Example 2:

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

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

More information about OQL can be obtained from [4] and [5]. The section below gives similar queries in SQL3

1.2.2 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 [6].

Consider the schema containing tables Bars with tuples of 'row type' BarType, Beers with tuples of 'row type' BeerType and Sells with tuples of 'row type' MenuType. The structure of these RowTypes is given below

RowType BarType

```
{
  Name char (20),
  Addr char (20)
};
```

RowType BeerType

```
{
  Name char (20),
  Manf char (20)
};
```

RowType MenuType

```
{  
    Bar Ref (BarType),  
    Beer Ref (BeerType),  
    Price Float  
};
```

Example 3:

To find the beers served by Joe the SQL3 query required is

Select beer->name from

From Sells

Where bar->name = 'Joe's bar';

Here beer is a reference to an object of type BeerType and -> serves as the dereference operator.

1.3 The need for a new interface:

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 even if he prepares syntactically correct queries they may not be the ones that are required 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 2, section 1.2.1, the user must know that *Things_to_do* in *Eiffel_Tower* is a set of objects of type *Thing_to_do*.

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 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 User-Friendly Interface:

The proposed interface to the object oriented database systems is a graphical user friendly one. To use this interface the user need not have prior knowledge about the schema of the underlying database, object oriented concepts etc as in the case of the query language. Using this interface the user can retrieve the desired data just by a few mouse clicks.

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 maps these requests of the user into meaningful queries of the query language supported by the database being used. The system uses these queries to retrieve data from the database and presents this data to user. Since these queries are invisible to the user the interface encapsulates the functionality of the query language.

This interface hides the complexity that exists in retrieving data from different kinds of constructs that are supported by the object oriented database systems. The way in which this transparency is achieved is explained in detail in the chapters that follow.

This interface is not restricted either to a particular application or 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

The Graphical User Interface

1. The Interface:

As mentioned before the system provides the user a graphical user-friendly interface to interact with. The user using the interface retrieves the data that he requires.

This chapter pictorially depicts the way the user interacts with the interface. It also explains how the retrieved data is displayed to the user.

The user initially encounters the screen given below:

OODB Queries

SELECT

Name
Ssn
Addr
Street
City
State
Zip_code
Empno

Done

In the above figure, the box beside the label 'SELECT' displays the schema of the database. In this chapter and in the chapters to come objects of the following classes are assumed to exist in the database.

Persons

```
{  
  Name: string,  
  SSN: integer,  
  Addr  
  {  
    Street: string,  
    City: string,  
    State: string,  
    Zip_code: string  
  }  
}
```

Participants

```
{  
  Empno: integer,  
  Ename: string,  
  Job: string,  
  Mgr: string,  
  Hire_date: date,  
  Sal: integer,  
  Deptno: string  
}
```


Employees

```
{  
  Empnumber: integer,  
  Person_data: references Persons  
  Manager: references Persons  
  Office_addr  
  {  
    Street: string,  
    City: string,  
    State: string,  
    Zip_code: string  
  }  
  Salary: integer,  
  Phone_nums: Set of strings  
}
```

Projects

```
{  
  Name: string,  
  Id: integer,  
  Owner: references Participants  
  Start_date: date,  
  Duration: string,  
  Modules: Set of Objects  
  {  
    Module_id: integer,  
    Module_name: string,  
    Module_start_date: date,  
    Module_duration: string  
  }  
}
```

```

Places_to_go
{
  Name: string,
  Addr
  {
    Street: string,
    City: string,
    State: string,
    Zip_code: string
  }
  Accom
  {
    Name: string,
    Addr
    {
      Street: string,
      City: string,
      State: string,
      Zip_code: string
    }
  }
}

```

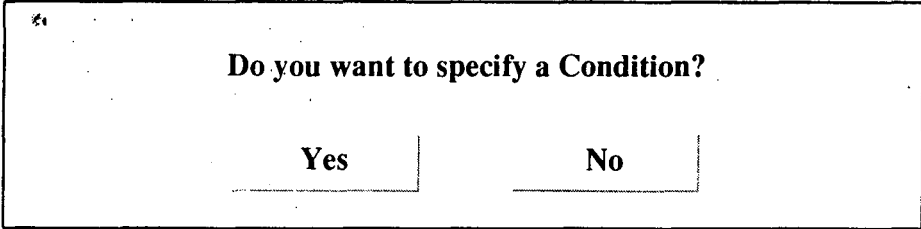
In the above representation, the term 'references' indicates that a member of an object points to some other object. For example, 'owner', the member of objects of 'projects' class points to an object of class 'participants'.

A 'Set' means collection of variable number of objects or members of a type. 'Phone_nums' in 'Employees' is a collection a strings. 'Modules' in 'Projects', a collection of objects, can be considered as a nested table.

The Select box in the screen displays the schema without its structure. The user will not know that 'Persons' is an object and the address 'Addr' of the 'Persons' is another object within that 'Persons' 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 'Persons' and 'Places_to_go', it is listed only once. No distinction is made either for references or sets. Hence the user will not know that 'phone_nums' is a set and 'Manager' refers to some other object.

When the user completes the selection and clicks on the 'Done' button the following dialogue box will appear



Do you want to specify a Condition?

Yes No

Fig 2

If the user wants to impose a condition on the data he wants to retrieve, he has to click on 'Yes' button. If the user clicks on 'No', the processing continues without pausing for the user to specify the condition.

If the user clicks on 'Yes' the screen given below is displayed.

OODB Queries

SELECT

Name
Ssn
Addr
Street
City
State
Zip_code
Empno

CONDITION

SUBMIT

Fig 3

When the user clicks on 'Submit' after specifying the condition the system starts processing the user requests. As mentioned before the system maps the user requests into meaningful queries of the query language supported by the database being used. 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.

1.1 Path Resolution:

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

1. Select o0.name , o0.addr.street, o0.addr.city, o0.addr.state, o0.addr.zip_code
from persons o0
2. Select o1.name, o1.addr.street, o1.addr.city, o1.addr.state, o1.addr.zip_code
from places_to_go o1.
3. Select o1.accom.name, o1.accom.addr.street, o1.accom.addr.city,
o1.accom.addr.state, o1.accom.addr.zip_code from places_to_go 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 'o1.addr.street' for retrieving the 'street' where the person lives. Just a click on the 'street' will serve the purpose for the user.

The user even need not know about the existence of 'Persons' objects in the database. Thus the schema of the existing database is made transparent to the user.

The result for the query no 1 given above is displayed in the following manner.

Results					
Persons					
	Addr				
Name	Street	City	State	Zip_code	
Previous	Next	Exit	New Data	Other Data	Help

Fig 4

The format, in which the data is displayed, represents the structure of the data.

The label "Persons" indicates that 'Name' and 'Addr' are members of 'Person' object. Similarly the label "Addr" indicates that 'Street', 'City', 'State' and 'Zip_code' are members of 'addr' object.

At the bottom of the screen 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 requests retrieve a large amount of data, it will not be possible to display the entire data at one go. These buttons help the user to go through the retrieved data.

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

New Data: On clicking this button the user will encounter the initial screen depicted in Fig.1 so that he can perform the select operation 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. In the case of above example, query no. 2 will be executed and the data retrieved will be displayed to the user.

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

1.2 Resolving References:

Let us consider the case where the user selects 'Manager' and 'Office_addr' in 'Employees'. The member 'Manager' of 'Employee' refers to a 'Person' object. The query generated in this case are given below:

```
Select b0.name, b0.ssn as manager, o0.office_addr.street,  
o0.office_addr.city, o0.office_addr.state, o0.office_addr.zip_code from employees  
o0, persons b0 where ref (b0) = o0.manger
```

The data retrieved by this query is displayed in this manner:

Again the correspondence between the format in which the data is displayed and the structure of the data can be felt. 'Name' and 'SSN' are the members of the 'Persons' object to which the member 'Manager' of 'Employee' points. Although only two members of the 'Persons' object are displayed, it is possible to retrieve and display other members of the 'Persons' object.

1.3 Handling Sets:

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

1. Select o0. id from projects o0.
2. Select o1.module_id, o1.module_name, o1.module_start_date, o1.module_duration from projects a, table (a.modules) o1.

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

Results

Projects

Modules				
Id	Module_id	Module_name	Module_start_date	Module_duration

TH-9476



Previous	Next	Exit	New Data	Other Data	Help
----------	------	------	----------	------------	------

Fig 6



TH-9476

In the above figure, 'Id' and 'Modules' are the members of 'Projects'. Each project has a no of modules. Hence each 'Id' of a 'Project' has a no of 'Modules' attached to it.

1.4 Working out relations:

Let 'Name', 'Ssn' and 'Id' be the attributes selected by the user. 'Name' and 'Ssn' are members of 'Persons' and 'Id' is the member of 'Project'. The system explores the relationship between these two objects by generating the following query

1. `select o0.name, o0.ssn, o1.id from persons o0, projects o1 where o0.name=o1.name.`

The data retrieved will be displayed in the following manner:

As in previous cases, the correspondence between data requested and data retrieved is apparent. 'Name' and 'Ssn' are members of 'Persons' and 'Id' is the member of 'Project'. This is similar to 'join' in relational databases.

2. Transparency:

The transparency achieved through this interface is described below:

1. The user doesn't have to know the class of the object in which the desired data exists. This implies that the 'from' clause in the general SQL query structure is eliminated.
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, sets etc. and references to other objects are made transparent to the user.
4. The interface is made intelligent enough to find out the relationships between different objects so that the user can retrieve data in most desirable form.

The user need not have knowledge even about the object oriented concepts to retrieve data from the database using this interface. The remaining chapters explain the design and implementation details of the system.

CHAPTER 3

DESIGN

2.1 Architectural Structure:

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 following is a pictorial depiction of the architecture of the system

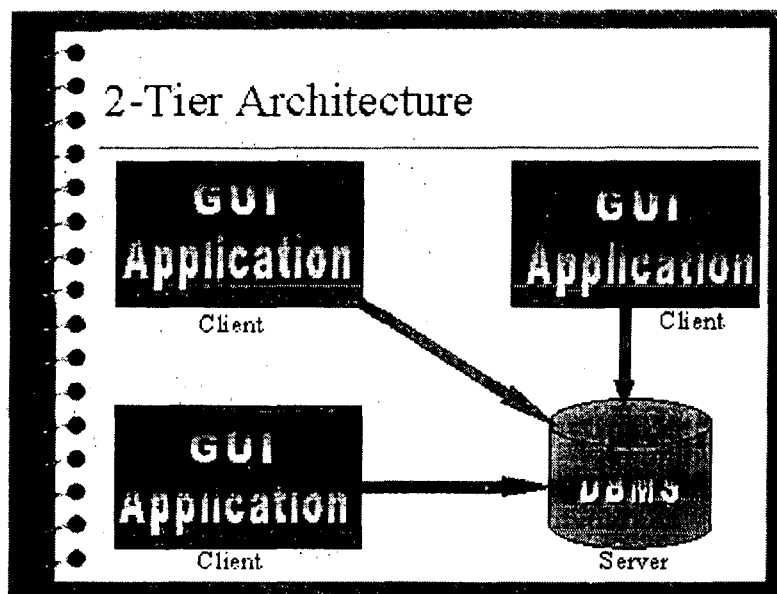


Fig 1

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 backend 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.

1.2 The Design:

The detailed design of this system is given in this section.

This interface is implemented in Java. Oracle is used as the backend.

There are two main modules in this implementation. The functionality of the interface and the database is divided between the modules *intf* and *dbcon* respectively.

intf 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 maps these requests of the user into meaningful queries of the query language supported by the database being used. These queries are handed over to *dbcon* for further processing.

dbcon establishes a connection with the database. It then sends the queries generated by *intf* to retrieve data from the database. It also presents the data retrieved to the user in an orderly and desirable manner.

The diagram below is the structure chart depicting the overall system design. The module *intfdb* coordinates the execution of *intf* and *dbcon*.

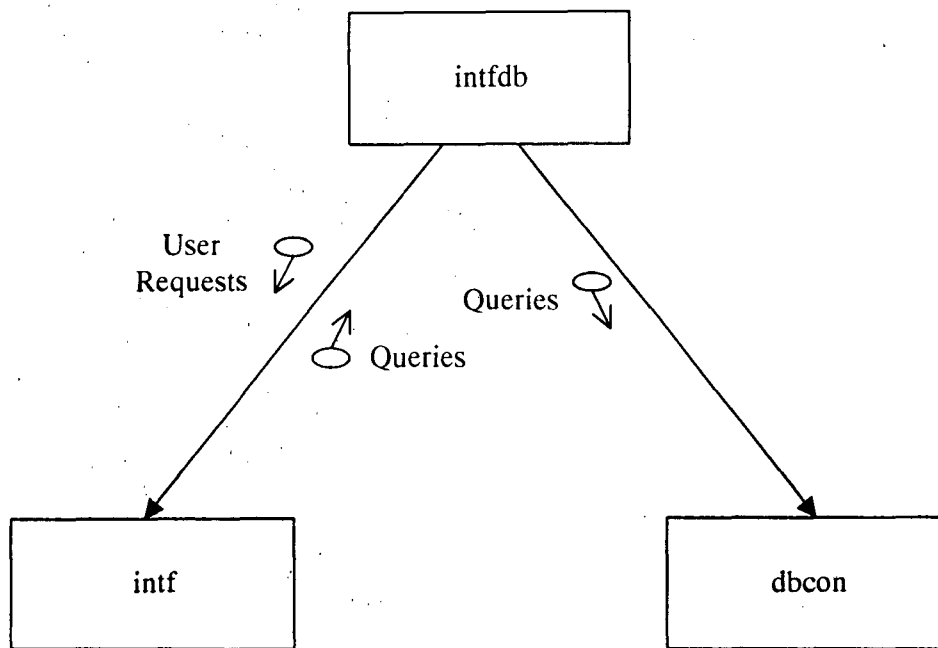


Fig 2

In the above diagram, the module “intf” maps the user requests into meaningful queries in the query language supported by the underlying database. These queries are sent to the module “dbcon”, which establishes a connection with the database, retrieves the data and displays it.

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

The "intf" 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.

As the user indicates the completion of selection (say, by the press of a button) the interface proceeds with further processing which is given below.

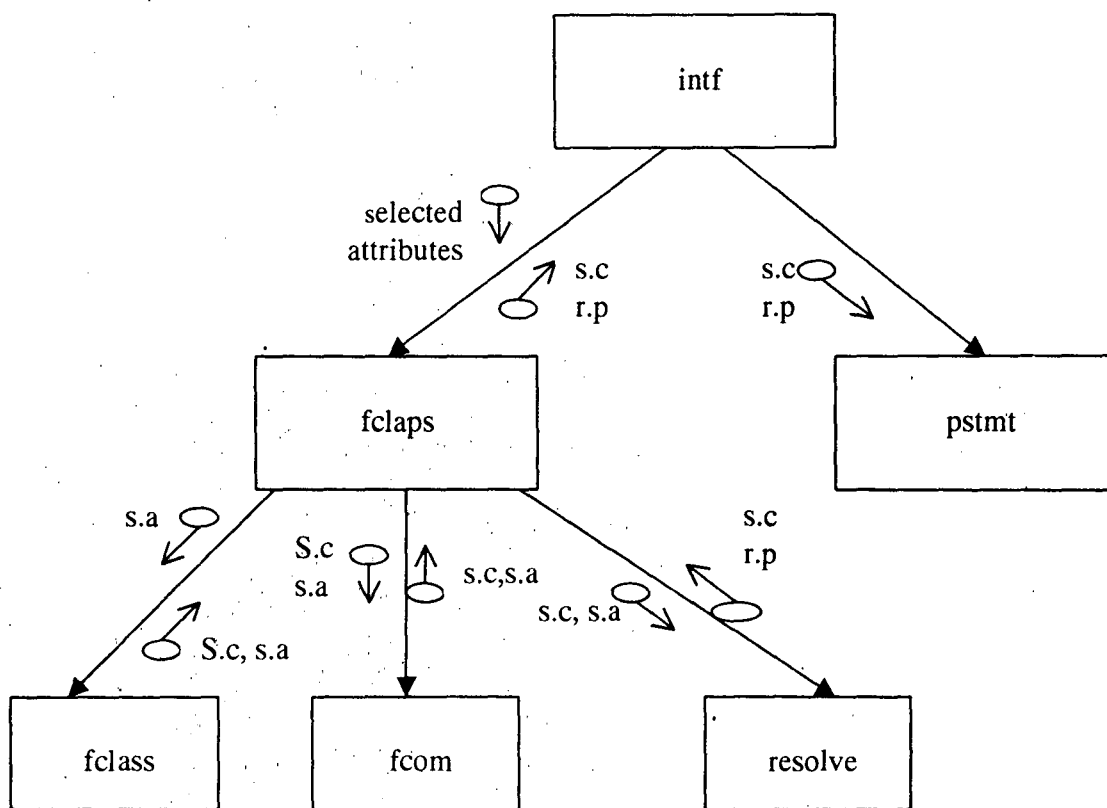


Fig 3

The structure diagram for pstmt is given below

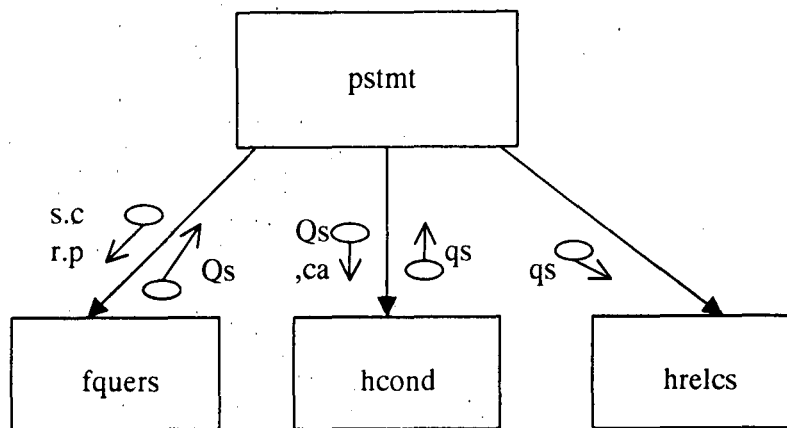


Fig 4

In the above diagrams

s.a : set of selected attributes

S.c,s.a : set of classes corresponding to the selected set of attributes.

s.c,s.a : s.c = a subset of S.c such that each class in that subset has all the selected attributes as its members. If no such subset exists then s.c is same as S.c

s.c,r.p : set of classes with the resolved paths for each of the selected attributes.

Qs : Queries (without condition if at all a condition is specified by the user)

ca : attributes mentioned in the condition

qs : queries (with condition, if it is specified by the user.).

The module "fclaps" finds the classes of the selected attributes and resolves their paths. The module "fclass" 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 "fcom" checks whether a common set of classes to which all the selected attributes belong, exists or not. The module "resolve" performs the path resolution, which is explained in chapter no.4.

The module "pstmt" prepares the query statements. The structure chart given below depicts the modules used by "pstmt".

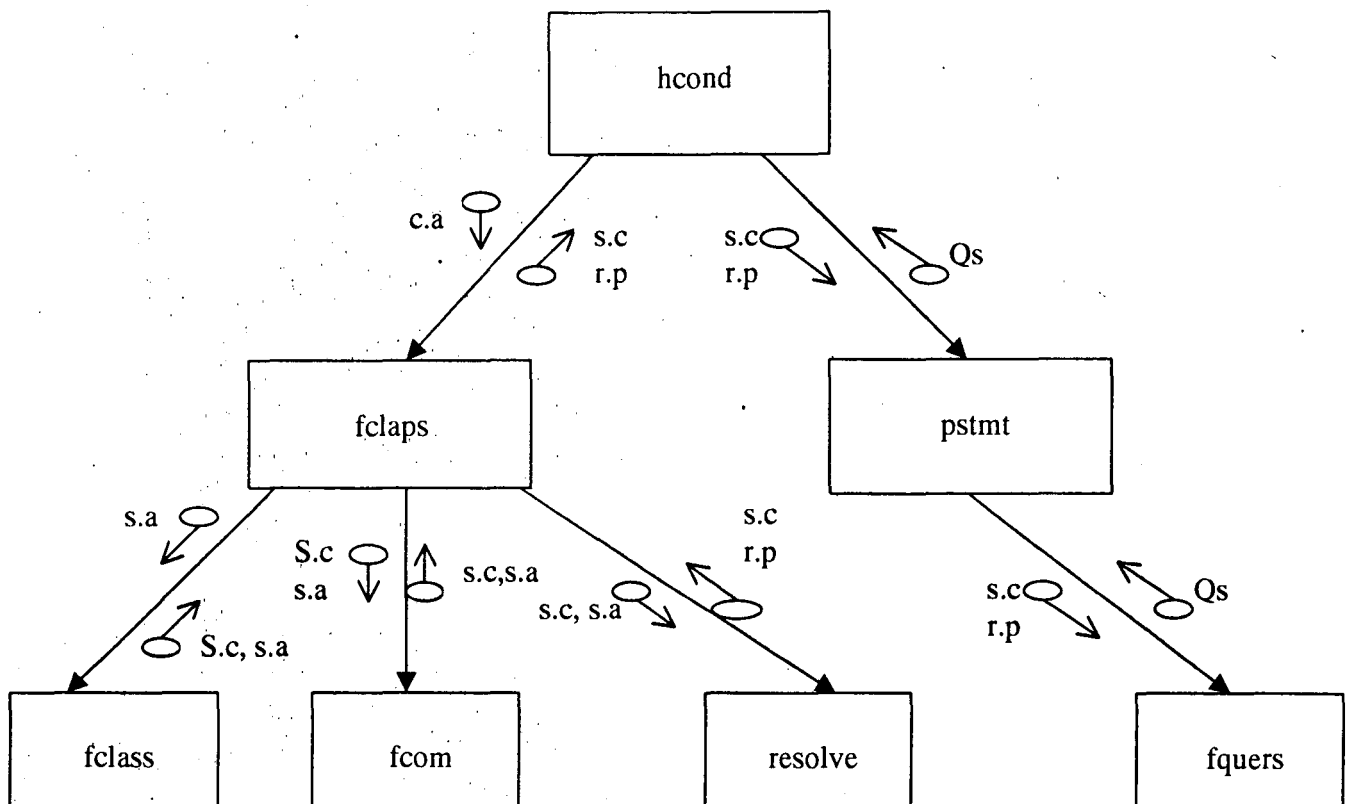


Fig 5

In the above diagram,

- s.a : set of attributes mentioned in the condition
- S.c,s.a : set of classes corresponding to the set of attributes mentioned in the condition.
- s.c,s.a : s.c = a subset of S.c such that each class in that subset has all the attributes mentioned in the condition as its members. If no such subset exists then s.c is same as S.c
- s.c,r.p : set of classes with the resolved paths for each of the attributes.

Qs : Queries (with condition specified by the user)

The module "hcond" is called by "pstmt" if the user imposes a condition on the data selected by him. As in the case of the attributes selected by the user, the previous procedure of calling "fclaps" and "pstmt" is repeated for the attributes that are mentioned in the condition imposed by the user.

The structure chart for the module "hrelcs" is given below:

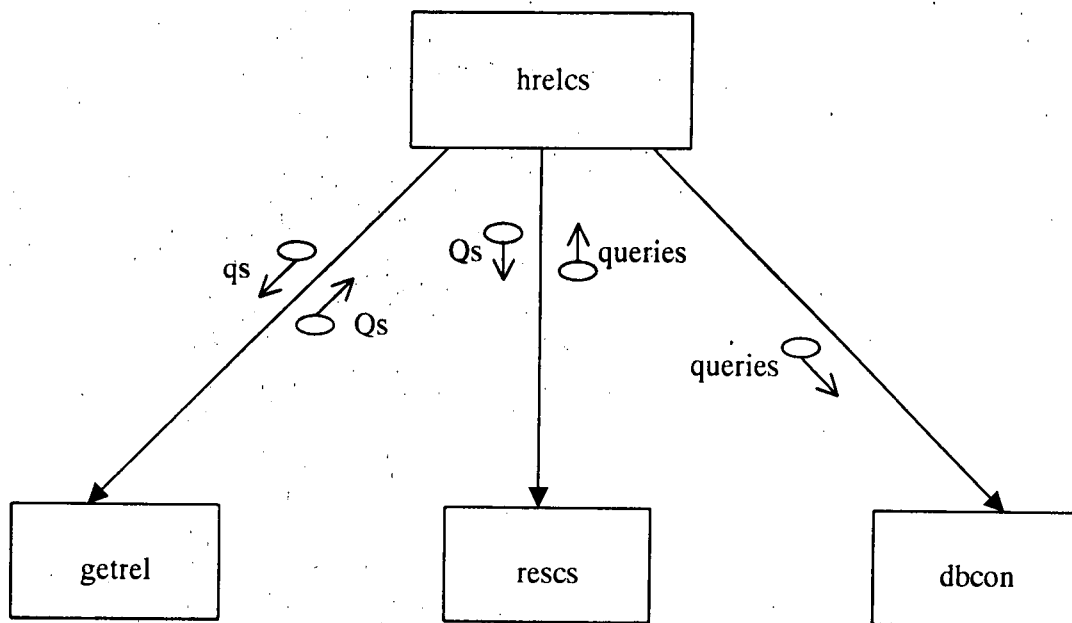


Fig 6

In the above diagram,

qs : The queries generated by the module "fquers" + the queries generated by the module "hcond".

Qs : qs + additional queries representing the relation between the selected attributes.

queries : Qs modified to handle the complexities

The module "hrelcs" which is called by "pstmt" resolves complex structures and uses the module "getrel" to find out whether a relation exists between the selected attributes. The module "rescs" resolves the complex structures in the queries. The module "hrelcs" passes on the queries to the module "dbcon". The structure chart depicting the modules used by it given below.

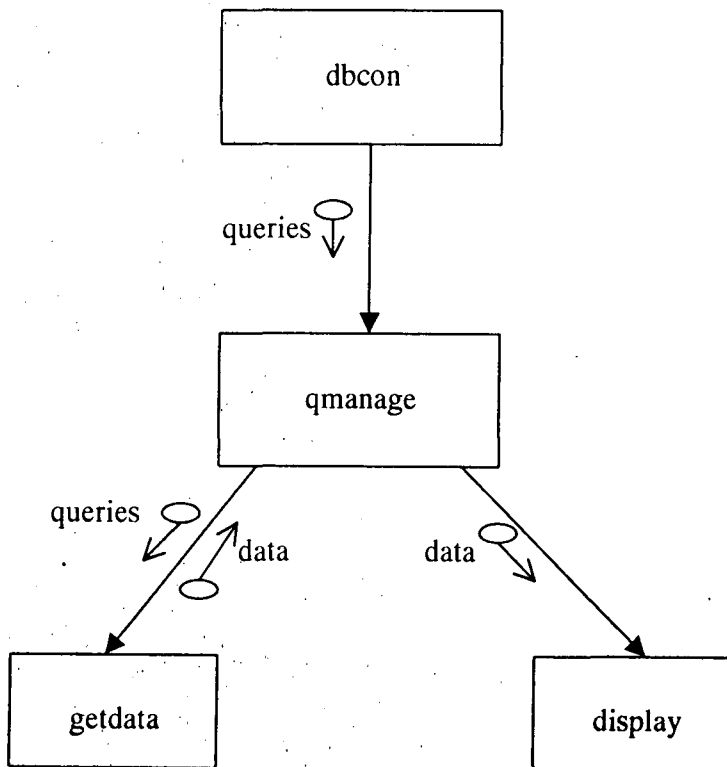


Fig 7

The module "qmanage" acts as the 'query manager'. "getdata" is used to retrieve the data from the database. The module "display" displays the data retrieved.

CHAPTER 4

IMPLEMENTATION

This chapter explains how the input given by the user is transformed into the output that the user finally views. The various stages involved in this process are

1. Finding the classes
2. Resolving the paths
3. Framing queries
4. Handling conditions
5. Deducing relationships.
6. Resolving complex structures.
7. Retrieving data
8. Displaying data

The first six stages can be considered as performing the mapping of user requests into meaningful queries supported by the query language of the underlying database. The last two stages are meant for establishing connection with the database, retrieving the data and displaying it to the user. These are explained below

1. Generation of queries:

1.1 Finding out classes:

As mentioned before, 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 requests by finding out the classes in which the selected attributes are members.

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 a_1 , a_2 and a_3 are the attributes selected by the user then s_1 , s_2 and s_3 are the sets of classes such that a_1 is a member of each class (or one of its component) in s_1 and so on. The resulting sets of classes are further processed to find whether these sets have any subset in common. If such a subset exists, it implies that each of the selected attributes is a member of each class (or one of its components) in that subset and the remaining elements, other than those in that subset, in each of the sets for different attributes are ignored.

The modules 'fclass' and 'fcom' perform the functions described above. Let 'Name' and 'Street' be the attributes selected by the user. 'fclass' on searching the schema, finds that the attribute 'Name' is mentioned in 'Persons' and 'Places_to_go', say set s_1 , and the attribute 'Street' is mentioned in 'Persons', 'Places_to_go' and 'Employee', say set s_2 . 'fcom' on examining these two sets finds that they have 'Persons' and 'Places_to_go', say set s , in common. Hence the class 'Employee' in set s_2 is ignored. This is explained below.

It is most likely that the user desires to view data from the instances of classes containing all the selected attributes as members. Hence those classes are preferred to the classes containing only a few of the selected attributes as members while mapping the user requests into queries.

1.2 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. Recognising this path is necessary to generate the query to retrieve the attribute A. This process can be termed as “path resolution”.

Again let us consider ‘Name’ and ‘Street’ as the attributes selected by the user. The module ‘resolve’ performs the path resolution. Let o1 be an object of ‘Persons’ and o2 be an object of ‘Places_to_go’. The paths generated for ‘Name’ are:

o1.name
o2.name
o2.accom.name

The paths generated for ‘Street’ are:

o1.addr.street
o2.addr.street
o2.accom.addr.street

The module ‘resolve’ 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.

1.3 Framing queries:

Using the lists generated by the module ‘resolve’ 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 sets easier which becomes evident in sections 1.5 and 1.6. Hence if n queries are generated, then each query is split into 3 parts. Query no i, $i \leq n$, is represented by the i^{th} element in each of these lists.

The module 'fquers' using the lists generated by the module 'resolve' creates the above mentioned ordered lists. Let the ordered lists Sq, Fq and Wq represent the Select clause, From clause and Where clause respectively of the queries generated. Assuming 'Name' and 'Street' as the selected attributes, the contents of these lists are given below

Qno	Sq	Fq	Wq
1	o1.name, o1.addr.street	Persons o1	-
2	o2.name, o2.addr.street	Places_to_go o2	-
3	O2.accom.name, o2.accom.addr.street	Places_to_go o2	-

Table 1

From the above ordered lists, queries can be formed by appending "Select" to an element in Sq and "From" to the corresponding element in Fq. Hence queries generated are

Select Sq[i] from Fq[i]. 1 <= i <= 3

The reasoning behind the formation of these queries is discussed in the remaining part of this section.

If the attributes selected by the user belong to two different classes then the module 'getrel' checks whether a relation exists between those two different classes. This is explained in the section 1.5.

If the attributes selected do not belong to the same class or one of its component classes and if there are two or more members in the class with the same name as that of a selected attribute then the nesting level of the attributes determines the queries generated. This is illustrated with an example below:

In this example places_to_go (section 1,chapter2) is modified as given

below

```
Places_to_go
{
  Name: string
  Addr
  {
    Street: string,
    City: string,
    State: string,
    Zip_code: string,
  }
  Accom
  {
    Taddr
    {
      Street: string,
      City: string,
      State: string,
      Zip_code: string,
    }
  }
}
```

If the user selects 'Name' and 'Street' the queries generated are

1. Select o0.name , o0.addr.street from persons o0.
2. Select o0.name, o0.accom.taddr.street from persons o0.

When compared with the level of nesting of 'name', o0.accom.taddr.street has deeper nesting than o0.addr.street. The lesser the difference in the level of nesting between the two selected attributes greater is the priority given to that

pair while forming the queries. Hence o0.addr.street is given preference over o0.accom.taddr.street

1.4 Handling Conditions:

In this stage the condition imposed by the user is incorporated into the queries generated. If the user doesn't impose any condition, the module devoted for this purpose is ignored.

As shown in Chapter 2, the user can enter the condition in the text area labeled 'Condition'. This condition may contain some attributes that are not selected by the user. The process of finding the classes and resolving paths is repeated for these attributes. The module 'fquers' is used to generate the ordered lists mentioned in section 1.3 so that they can be used in embedding the condition in the queries.

The module 'hcond' handles the condition imposed by the user. Let 'Name' and 'Street' be the attributes selected by the user. The constraints given below must be satisfied while giving the condition.

1. The statement representing the condition must be of the form

attrib oper const/attrib

Here

attrib is one of the names listed in the initial screen (Fig. 1, chapter2)

oper is a relational operator such as '=', '<' etc

const may be a number or a string.

2. Strings must be enclosed between ' " '. For example, name = "John".
3. Numbers must not be enclosed between ' " ' or any other symbol. They must be entered as it is, as shown in the following example.

Ssn = 123456.

4. The interface doesn't impose any restriction on the number of conditions that the user can impose.

Let 'Name' and 'Zip_code' be the attributes selected by the user. Let City = "New Delhi" be the condition imposed by the user. The module 'hcond' generates the following queries to satisfy these requests

1. Select o0.name, o0.addr.zip_code from persons o0 where o0.addr.city= 'New Delhi'.
2. Select o1.name, o1.addr.zip_code from places_to_go o1 where o1.addr.city= 'New Delhi'.
3. Select o1.accom.name, o1.accom.addr.zip_code from places_to_go o1 where o1.accom.addr.city= 'New Delhi'.
4. Select o0.name, o0.addr.zip_code from persons o0.
5. Select o1.name, o1.addr.zip_code from places_to_go o1.
6. Select o1.accom.name, o1.accom.addr.zip_code from places_to_go o1

If the condition imposed by the user fails to retrieve data from the database, then queries 4,5 and 6 are used to get the data ignoring the condition.

1.5 Deducing relations:

If all the selected attributes do not belong to the same class then the module 'getrel' finds out whether a relationship exists between the different classes to which these attributes belong. If the selected attributes all are members of a single class or a set of few classes, this stage is not encountered during the processing of user requests.

An exponential search method is used to find whether a relation exists or not. Let a1 and a2, in classes c1 and c2 be the attributes selected by the user. Let A be a variable that represents a class in the schema. The search method involves the steps given below:

1. A=c1
2. If there is an attribute with the same name in A and c2 then go to 5.
3. Find a class c3 with at least one member having the name same as that of a member in class A. If no such class exists report the lack of relation between the selected attributes.
4. A=c3. Go to 2.
5. Generate the query.

In step 3, if it is found that the attributes selected are not related, the user encounters the following dialogue box:

UNRELATED DATA	
Do you want to Continue?	
Yes	No

If he clicks no 'No' the processing is stopped. However, if the user clicks on 'Yes', the system proceeds further. For example, if the user selects 'Ssn' and 'Module_name' which are not related the user encounters the above dialogue box. If the user clicks on 'Yes' the data retrieved by the following queries is displayed one after the another

1. select o0.ssn from persons o0
2. select o1.module_name from projects a, table (a.modules) o1.

If the relation appears to exist between the selected attributes then the query expressing the relation is generated in step 5. To illustrate the search procedure with an example, let us assume that the schema of 'Participants' (refer to chapter 2, section 1) is modified and now 'Id' is a member in it. Let 'Ssn' and 'Ename' be the attributes selected by the user. The relevant part of the modified schema is shown below:

```

Persons
{
  Name: string,
  SSN: integer,
  Addr
  {
    Street: string,
    City: string,
    State: string,
    Zip_code: string
  }
}

```

Participants

```
{  
  Id: integer,  
  Empno: integer,  
  Ename: string,  
  Job: string,  
  Mgr: string,  
  Hire_date: date,  
  Sal: integer,  
  Deptno: string  
}
```

Projects

```
{  
  Name: string,  
  Id: integer,  
  Owner: references Participants  
  Start_date: date,  
  Duration: string,  
  Modules: Set of Objects  
  {  
    Module_id: integer,  
    Module_name: string,  
    Module_start_date: date,  
    Module_duration: string  
  }  
}
```

'Ssn' is a member of 'Persons' and 'Ename' is a member of 'Participants'. These two classes have no member in common. In step 3 of the search

procedure, it is identified that 'Name' is common in both 'Projects' and 'Persons'. In step 2 it is found that 'Projects' and 'Participants' have a member 'Id' in common. Finally in step 5 the following query is generated

```
select o0.ssn, o1.id from persons o0, participants o1, projects fr0 where o0.name = fr0.name and fr0.id=o1.id.
```

To reduce the complexity of the search, for each attribute in the schema a data structure containing the set of classes, in which the attribute is a member, is maintained. Also a list containing the set of classes already searched is constantly updated in the search procedure, to avoid searching the same class again and again. In this implementation a file is used for each attribute to store the set of classes in which it is a member. A separate module is written to update these files whenever the schema is updated.

1.6 Resolving Complex structures:

The queries generated in the earlier stages are not capable of retrieving data from complex structures such as sets. While generating the queries, the module 'fquers' doesn't check whether the attribute being retrieved is a set or a reference or any other complex structure. In this stage the module 'rescs' parses the queries generated and updates them, if required, thus making them capable of handling complex structures.

The module 'fclass', while searching the schema for finding the classes in which the selected attributes are members, updates lists that contain information about the complex structures. The module 'rescs' uses these lists to identify the attributes that represent complex structures.

Consider the query given below

```
Select o0.empnumber, o0.manager from projects o0.
```

The member 'manager' of 'Employees' refers to a 'Persons' object. The module 'rescs', through the list that contains the attributes which are references, identifies

that owner refers to a 'Persons' object and updates the query. The modified query is given below:

Select o0.empnumber, b0.name, b0.ssn as manager from employees o0, persons b0 where ref(b0) = o0.manager.

The module 'fquers' responds to the user request for 'Name' and 'Module_name' with the following queries

Select o0.name from projects o0.

Select o1.module_name from modules o1.

The module 'rescs' observes that 'Modules' is a nested table within projects and modifies the above to generate the queries given below:

Select o0.name from projects o0.

Select o1.module_name from projects a, table (a.modules) o1.

The module ensures the execution of second query whenever the first query is executed, by maintaining an ordered list, say dep (for dependency). The number of elements in the list is n where n is equal to the no of queries generated. In this case dep[1]=2 and dep[2]=-1. dep[1]=2 indicates the dependency of query no 1 on 2. Similarly a negative value doesn't indicate a dependency.

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. Retrieving and Displaying Data:

2.1 Retrieving Data:

In this stage, the queries formed in the previous stages are used to retrieve data from the database.

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 'getdata' establishes the connection with the database and stores the retrieved data. It also handles the dependencies between queries mentioned above.

1.8 Displaying Data:

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

The module 'display' scans the query and prepares the image of the screen that should be displayed to the user. Consider the query

```
Select o0.name, o0.addr.street, o0.addr.city, o0.addr.state and o0.addr.zip_code  
from persons o0.
```

Scanning this query, the module 'display' prepares an ordered list of strings whose contents are given below:

```
Persons  
Addr (2-5)  
Name, Street, City, State, Zip_code
```

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 'Persons', the second row contains 'Addr' from column 2 to column 5 and the third row contains the attributes whose values are being displayed (refer to section 1.1 in chapter 2). 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 2 pictorially depicts the way the data is displayed for different types of queries.

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 98 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 hassles such as pointers that exist in it. The main reason for choosing Java is its built in support for designing graphical user interfaces. The abstract window toolkit (awt) that comes with Java offers various widgets (buttons, text boxes, lists etc) which make the interface both graphical and user friendly.

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 sets 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 5

CONCLUSION

Object Oriented Databases are getting more and more popular day by day. This application is a graphical user interface to retrieve data from object oriented database systems. 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 maps the user requests into meaningful queries of the query language supported by the database being used. The system uses these queries to retrieve data from the database and presents it to user.

When the user completes the selection, 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 which involves finding out the location or the depth of each attribute in the class. Several issues are considered while framing the queries. The application infers that the selected attributes are related if there exists atleast one class in which all the selected attributes are members. If no such class exists it finds out whether relationships exist between different classes to which the selected attributes belong. If it finds that the attributes selected are not related it reports the same to the user and proceeds further if the user wants to do so. In the same class if two or more attributes with the same name exist then attributes at the same level are preferred. The queries formed at this stage are modified to make them capable of retrieving data from complex structures. 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 interface is to provide 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 O2*
Francois Bancilhon, Claude Delobel, Paris Kanellakis(eds.)
2. The following URLs give introductory information about object oriented databases.
 1. <http://misdb.bpa.arizona.edu/~mis696g/Reports/ObjectDB/oodb.htm>
 2. <http://www.cs.vu.nl/~fgouwer/study/oop/introduction.html>
 3. <http://www.dis.port.ac.uk/~chandler/OOLectures/database/database.htm>
3. Query Languages for Object Oriented databases can be found at the following URLs
 1. www.infobiogen.fr/services/eyedb/pub/manual/node7.htm
 2. <http://www.soberit.hut.fi/~kta/odm/sld022.htm>
4. An overview of Object Query Language, a SQL like declarative language can be obtained from
<http://www.odmg.org/standard/standardoverview.htm>
5. The user manual for OQL can be obtained from the website
<http://www.cis.upenn.edu/~cis550/oql.pdf>
6. The URLs given below give information about SQL3
 1. <http://www.objs.com/x3h7/sql3.htm>
 2. <http://www.soberit.hut.fi/~kta/odm/sld033.htm>
 3. <http://www.csee.umbc.edu/461/spring98/lectures/lecture20/ppframe.htm>
 4. <http://datasplash.cs.berkeley.edu:8000/sequoia/dba/montage/FAQ/SQL.html>