UNIT 1 OBJECT ORIENTED DATABASE

Structure **Page No.**

1.0 INTRODUCTION

Object oriented software development methodologies have become very popular in the development of software systems. Database applications are the backbone of most of these commercial business software developments. Therefore, it is but natural that, object technologies also, have their impact on database applications. Database models are being enhanced in computer systems for developing complex applications. For example, a true hierarchical data representation like generalisation hierarchy scheme in a rational database would require a number of tables, but could be a very natural representation for an object oriented system. Thus, object oriented technologies have found their way into database technologies. The present day commercial RDBMS supports the features of object orientation.

This unit provides an introduction to various features of object oriented databases. In this unit, we shall discuss, the need for object oriented databases, the complex types used in object oriented databases, how these may be supported by inheritance etc. In addition, we also define object definition language (ODL) and object manipulation language (OML). We shall discuss object-oriented and object relational databases as well.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- define the need for object oriented databases;
- explain the concepts of complex data types;
- use SQL to define object oriented concepts;

- familiarise yourself with object definition and query languages, and
- define object relational and object-oriented databases.

1.2 WHY OBJECT ORIENTED DATABASE?

An object oriented database is used for complex databases. Such database applications require complex interrelationships among object hierarchies to be represented in database systems. These interrelationships are difficult to be implement in relational systems. Let us discuss the need for object oriented systems in advanced applications in more details. However, first, let us discuss the weakness of the relational database systems.

1.2.1 Limitation of Relational Databases

Relational database technology was not able to handle complex application systems such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and Computer Integrated Manufacturing (CIM), Computer Aided Software Engineering (CASE) etc. The limitation for relational databases is that, they have been designed to represent entities and relationship in the form of two-dimensional tables. Any complex interrelationship like, multi-valued attributes or composite attribute may result in the decomposition of a table into several tables, similarly, complex interrelationships result in a number of tables being created. Thus, the main asset of relational databases viz., its simplicity for such applications, is also one of its weaknesses, in the case of complex applications.

The data domains in a relational system can be represented in relational databases as standard data types defined in the SQL. However, the relational model does not allow extending these data types or creating the user's own data types. Thus, limiting the types of data that may be represented using relational databases.

Another major weakness of the RDMS is that, concepts like inheritance/hierarchy need to be represented with a series of tables with the required referential constraint. Thus they are not very natural for objects requiring inheritance or hierarchy.

However, one must remember that relational databases have proved to be commercially successful for text based applications and have lots of standard features including security, reliability and easy access. Thus, even though they, may not be a very natural choice for certain applications, yet, their advantages are far too many. Thus, many commercial DBMS products are basically relational but also support object oriented concepts.

1.2.2 The Need for Object Oriented Databases

As discussed in the earlier section, relational database management systems have certain limitations. But how can we overcome such limitations? Let us discuss some of the basic issues with respect to object oriented databases.

The objects may be complex, or they may consists of low-level object (for example, a window object may consists of many simpler objects like menu bars scroll bar etc.). However, to represent the data of these complex objects through relational database models you would require many tables – at least one each for each inherited class and a table for the base class. In order to ensure that these tables operate correctly we would need to set up referential integrity constraints as well. On the other hand, object

oriented models would represent such a system very naturally through, an inheritance hierarchy. Thus, it is a very natural choice for such complex objects.

Consider a situation where you want to design a class, (let us say a Date class), the advantage of object oriented database management for such situations would be that they allow representation of not only the structure but also the operation on newer user defined database type such as finding the difference of two dates. Thus, object oriented database technologies are ideal for implementing such systems that support complex inherited objects, user defined data types (that require operations in addition to standard operation including the operations that support polymorphism).

Another major reason for the need of object oriented database system would be the seamless integration of this database technology with object-oriented applications. Software design is now, mostly based on object oriented technologies. Thus, object oriented database may provide a seamless interface for combining the two technologies.

The Object oriented databases are also required to manage complex, highly interrelated information. They provide solution in the most natural and easy way that is closer to our understanding of the system. **Michael Brodie** related the object oriented system to human conceptualisation of a problem domain which enhances communication among the system designers, domain experts and the system end users.

The concept of object oriented database was introduced in the late 1970s, however, it became significant only in the early 1980s. The initial commercial product offerings appeared in the late 1980s. Today, many object oriented databases products are available like Objectivity/DB (developed by Objectivity, Inc.), ONTOS DB (developed by ONTOS, Inc.), VERSANT (developed by Versant Object Technology Corp.), ObjectStore (developed by Object Design, Inc.), GemStone (developed by Servio Corp.) and ObjectStore PSE Pro (developed by Object Design, Inc.). An object oriented database is presently being used for various applications in areas such as, e-commerce, engineering product data management; and special purpose databases in areas such as, securities and medicine.

Figure 1 traces the evolution of object oriented databases. *Figure 2* highlights the strengths of object oriented programming and relational database technologies. An object oriented database system needs to capture the features from both these world. Some of the major concerns of object oriented database technologies include access optimisation, integrity enforcement, archive, backup and recovery operations etc.

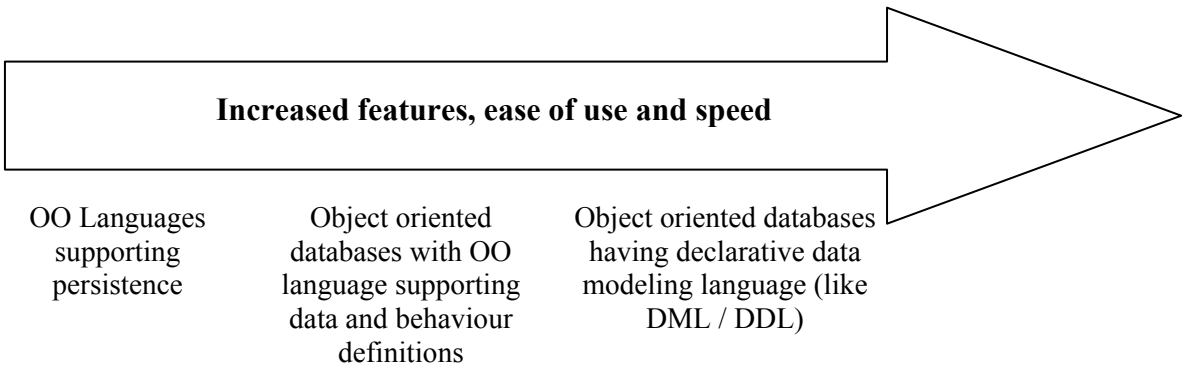

Figure 1: The evolution of object-oriented databases

The major standard bodies in this area are Object Management Group (OMG), Object Database Management Group (ODMG) and X3H7.

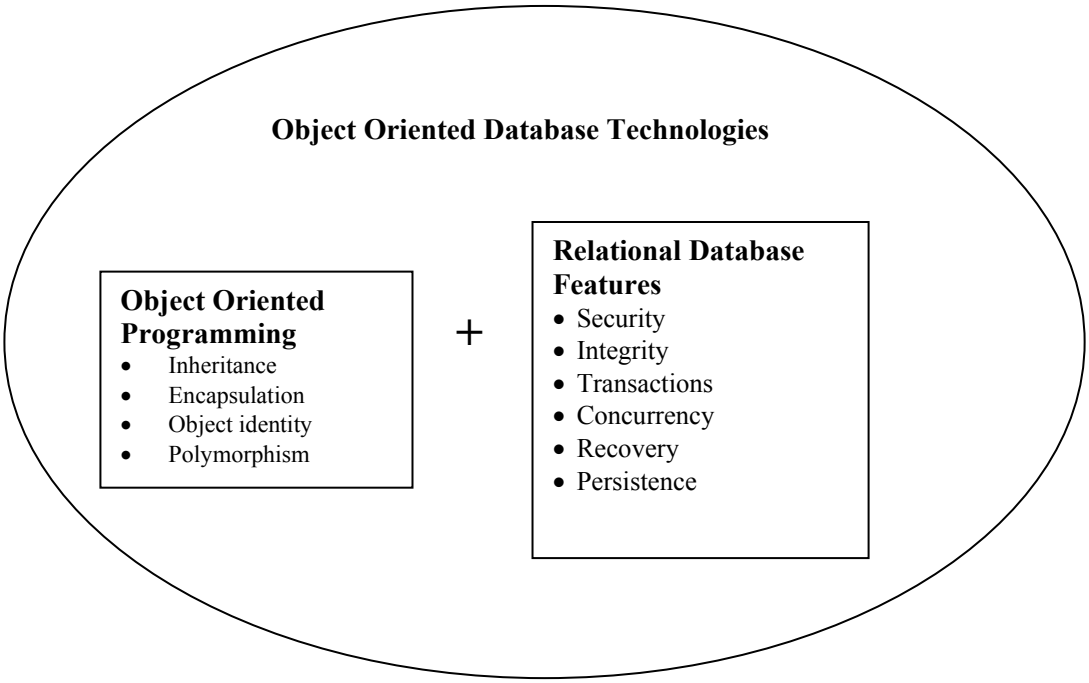

Figure 2: Makeup of an Object Oriented Database

Now, the question is, how does one implement an Object oriented database system? As shown in *Figure 2* an object oriented database system needs to include the features of object oriented programming and relational database systems. Thus, the two most natural ways of implementing them will be either to extend the concept of object oriented programming to include database features – OODBMS or extend the relational database technology to include object oriented related features – Object Relational Database Systems. Let us discuss these two viz., the object relational and object oriented databases in more details in the subsequent sections.

1.3 OBJECT RELATIONAL DATABASE SYSTEMS

Object Relational Database Systems are the relational database systems that have been enhanced to include the features of object oriented paradigm. This section provides details on how these newer features have been implemented in the SQL. Some of the basic object oriented concepts that have been discussed in this section in the context of their inclusion into SQL standards include, the complex types, inheritance and object identity and reference types.

1.3.1 Complex Data Types

In the previous section, we have used the term complex data types without defining it. Let us explain this with the help of a simple example. Consider a composite attribute – *Address*. The address of a person in a RDBMS can be represented as:

House-no and apartment
Locality
City
State
Pin-code

When using RDBMS, such information either needs to be represented as set attributes as shown above, or, as just one string separated by a comma or a semicolon. The second approach is very inflexible, as it would require complex string related operations for extracting information. It also hides the details of an address, thus, it is not suitable.

If we represent the attributes of the address as separate attributes then the problem would be with respect to writing queries. For example, if we need to find the address of a person, we need to specify all the attributes that we have created for the address viz., House-no, Locality…. etc. The question is –Is there any better way of representing such information using a single field? If, there is such a mode of representation, then that representation should permit the distinguishing of each element of the address? The following may be one such possible attempt:

```
CREATE TYPE Address  AS    (
            House              Char(20)
            Locality           Char(20)
            City               Char(12)
            State              Char(15)
            Pincode            Char(6)
            ) ;
```

Thus, *Address* is now a new type that can be used while showing a database system scheme as:

```
CREATE TABLE STUDENT    (
            name               Char(25)
            address            Address
            phone              Char(12)
            programme          Char(5)
            dob                ???
            ) ;
```

* Similarly, complex data types may be extended by including the date of birth field (dob), which is represented in the discussed scheme as??? This complex data type should then, comprise associated fields such as, day, month and year. This data type should also permit the recognition of difference between two dates; the day; and the year of birth. But, how do we represent such operations. This we shall see in the next section.

But, what are the advantages of such definitions?

Consider the following queries:

Find the name and address of the students who are enrolled in MCA programme.

```
SELECT      name, address
FROM        student
WHERE       programme = 'MCA' ;
```

Please note that the attribute 'address' although composite, is put only once in the query. But can we also refer to individual components of this attribute?

Find the name and address of all the MCA students of Mumbai.

```
SELECT      name, address
FROM        student
WHERE       programme = 'MCA' AND address.city = 'Mumbai';
```

Thus, such definitions allow us to handle a composite attribute as a single attribute with a user defined type. We can also refer to any of the component of this attribute without any problems so, the data definition of attribute components is still intact.

Complex data types also allow us to model a table with multi-valued attributes which would require a new table in a relational database design. For example, a library database system would require the representation following information for a book.

Book table:

- ISBN number
- Book title
- Authors
- Published by
- Subject areas of the book.

Clearly, in the table above, authors and subject areas are multi-valued attributes. We can represent them using tables (ISBN number, author) and (ISBN number, subject area) tables. (Please note that our database is not considering the author position in the list of authors).

Although this database solves the immediate problem, yet it is a complex design. This problem may be most naturally represented if, we use the object oriented database system. This is explained in the next section.

1.3.2 Types and Inheritances in SQL

In the previous sub-section we discussed the data type – Address. It is a good example of a structured type. In this section, let us give more examples for such types, using SQL. Consider the attribute:

- Name – that includes given name, middle name and surname
- Address – that includes address details, city, state and pincode.
- Date – that includes day, month and year and also a method for distinguish one data from another.

SQL uses Persistent Stored Module (PSM)/PSM-96 standards for defining functions and procedures. According to these standards, functions need to be declared both within the definition of type and in a CREATE METHOD statement. Thus, the types such as those given above, can be represented as:

```
CREATE TYPE          Name  AS (
                 given-name Char (20),
                 middle-name        Char(15),
                 sur-name           Char(20)
)
FINAL

CREATE TYPE          Address AS (
                 add-det            Char(20),
                 city               Char(20),
                 state              Char(20),
                 pincode            Char(6)

)
NOT FINAL
```

```
CREATE TYPE     Date  AS  (
            dd              Number(2),
            mm              Number(2),
            yy              Number(4)
)
FINAL
METHOD difference (present Date)
RETURNS INTERVAL days ;
```

This method can be defined separately as:

```
CREATE INSTANCE METHOD difference (present Date)
            RETURNS INTERVAL days FOR Date
BEGIN
// Code to calculate difference of the present date to the date stored in the object. //
// The data of the object will be used with a prefix SELF as: SELF.yy, SELF.mm etc.
//
// The last statement will be RETURN days that would return the number of days//
END
```

These types can now be used to represent class as:

```
CREATE  TYPE    Student AS (
            name        Name,
            address     Address,
            dob         Date
)
```

'FINAL' and 'NOT FINAL' key words have the same meaning as you have learnt in JAVA. That is a final class cannot be inherited further.

There also exists the possibility of using constructors but, a detailed discussion on that is beyond the scope of this unit.

Type Inheritance

In the present standard of SQL, you can define inheritance. Let us explain this with the help of an example.

Consider a type University-person defined as:

```
CREATE TYPE     University-person AS (
            name        Name,
            address     Address
)
```

Now, this type can be inherited by the Staff type or the Student type. For example, the Student type if inherited from the class given above would be:

```
CREATE TYPE     Student
            UNDER   University-person  (
            programme Char(10),
            dob             Number(7)
)
```

Similarly, you can create a sub-class for the staff of the University as:

```
CREATE TYPE     Staff
```

```
        UNDER     University-person  (
        designation  Char(10),
        basic-salary Number(7)
)
```

Notice, that, both the inherited types shown above-inherit the name and address attributes from the type University-person. Methods can also be inherited in a similar way, however, they can be overridden if the need arises.

Table Inheritance

The concept of table inheritance has evolved to incorporate implementation of generalisation/ specialisation hierarchy of an E-R diagram. SQL allows inheritance of tables. Once a new type is declared, it could be used in the process of creation of new tables with the usage of keyword "OF". Let us explain this with the help of an example.

Consider the University-person, Staff and Student as we have defined in the previous sub-section. We can create the table for the type University-person as:

```
CREATE TABLE University-members OF University-person ;
```

Now the table inheritance would allow us to create sub-tables for such tables as:

```
CREATE TABLE student-list OF Student
        UNDER University-members ;
```

Similarly, we can create table for the University-staff as:

```
CREATE TABLE staff OF Staff
        UNDER University-members ;
```

Please note the following points for table inheritance:

- The type that associated with the sub-table must be the sub-type of the type of the parent table. This is a major requirement for table inheritance.
- All the attributes of the parent table – (University-members in our case) should be present in the inherited tables.
- Also, the three tables may be handled separately, however, any record present in the inherited tables are also implicitly present in the base table. For example, any record inserted in the student-list table will be implicitly present in university-members tables.
- A query on the parent table (such as university-members) would find the records from the parent table and all the inherited tables (in our case all the three tables), however, the attributes of the result table would be the same as the attributes of the parent table.
- You can restrict your query to – only the parent table used by using the keyword – ONLY. For example,

```
        SELECT NAME FROM university-member ONLY ;
```

1.3.3 Additional Data Types of OOP in SQL

The object oriented/relational database must support the data types that allows multi-valued attributes to be represented easily. Two such data types that exit in SQL are:

- Arrays – stores information in an order, and
- Multisets – stores information in an unordered set.

Let us explain this with the help of example of book database as introduced in section 1.3. This database can be represented using SQL as:

```
CREATE TYPE        Book AS  (
            ISBNNO              Char (14),
            TITLE               Char (25),
            AUTHORS Char (25) ARRAY [5],
            PUBLISHER           Char (20),
            KEYWORDS            Char (10) MULTISET
)
```

Please note, the use of the type ARRAY. Arrays not only allow authors to be represented but, also allow the sequencing of the name of the authors. Multiset allows a number of keywords without any ordering imposed on them.

But how can we enter data and query such data types? The following SQL commands would help in defining such a situation. But first, we need to create a table:

```
CREATE TABLE       library  OF  Book ;
```

```
INSERT INTO library VALUES.
('008-124476-x', 'Database Systems', ARRAY ['Silberschatz', 'Elmasri' ], 'XYZ
PUBLISHER', multiset [ 'Database', 'Relational', 'Object Oriented']) ;
```

The command above would insert information on a hypothetical book into the database.

Let us now write few queries on this database:

Find the list of books related to area Object Oriented:

```
SELECT ISBNNO, TITLE
FROM library
WHERE 'Object Oriented' IN ( UNNEST ( KEYWORDS)) ;
```

Find the first author of each book:

```
SELECT ISBNNO, TITLE, AUTHORS [1]
FROM  library
```

You can create many such queries, however, a detailed discussion on this, can be found in the SQL 3 standards and is beyond the scope of this unit.

1.3.4 Object Identity and Reference Type Using SQL

Till now we have created the tables, but what about the situation when we have attributes that draws a reference to another attribute in the same table. This is a sort of referential constraint. The two basic issues related such a situation may be:

- How do we indicate the referenced object? We need to use some form of identity, and
- How do we establish the link?

Let us explain this concept with the help of an example; consider a book procurement system which provides an accession number to a book:

```
CREATE TABLE        book-purchase-table  (

                ACCESSION-NO    CHAR (10),
                ISBNNO REF (Book) SCOPE (library)
) ;
```

The command above would create the table that would give an accession number of a book and will also refer to it in the library table.

However, now a fresh problem arises how do we insert the books reference into the table? One simple way would be to search for the required ISBN number by using the system generated object identifier and insert that into the required attribute reference. The following example demonstrates this form of insertion:

```
INSERT INTO book-purchase-table VALUES ('912345678', NULL) ;

UPDATE book-table
SET ISBNNO = (SELECT book_id
                    FROM library
                   WHERE ISBNNO = '83-7758-476-6')
WHERE ACCESSION-NO  = '912345678'
```

Please note that, in the query given above, the sub-query generates the object identifier for the ISBNNO of the book whose accession number is 912345678. It then sets the reference for the desired record in the book-purchase-table.

This is a long procedure, instead in the example as shown above, since, we have the ISBNNO as the key to the library table, therefore, we can create a user generated object reference by simply using the following set of SQL statements:

```
CREATE TABLE        book-purchase-table  (

                ACCESSION-NO    CHAR (10),
                ISBNNO REF (Book) SCOPE (library) USER GENERATED

) ;

INSERT INTO book-purchase-table VALUES ('912345678', '83-7758-476-6') ;
```

☞ Check Your Progress 1

1) What is the need for object-oriented databases?

……..

……

……

2) How will you represent a complex data type?

……..

……

……

3) Represent an address using SQL that has a method for locating pin-code information.

……..

……

……

4) Create a table using the type created in question 3 above.

……..

……

……

5) How can you establish a relationship with multiple tables?

……..

……

……

1.4 OBJECT ORIENTED DATABASE SYSTEMS

Object oriented database systems are the application of object oriented concepts into database system model to create an object oriented database model. This section describes the concepts of the object model, followed by a discussion on object definition and object manipulation languages that are derived SQL.

1.4.1 Object Model

The ODMG has designed the object model for the object oriented database management system. The Object Definition Language (ODL) and Object Manipulation Language (OML) are based on this object model. Let us briefly define the concepts and terminology related to the object model.

Objects and Literal: These are the basic building elements of the object model. An object has the following four characteristics:

- A unique identifier
- A name
- A lifetime defining whether it is persistent or not, and
- A structure that may be created using a type constructor. The structure in OODBMS can be classified as atomic or collection objects (like Set, List, Array, etc.).

A literal does not have an identifier but has a value that may be constant. The structure of a literal does not change. Literals can be atomic, such that they correspond to basic data types like int, short, long, float etc. or structured literals (for example, current date, time etc.) or collection literal defining values for some collection object.

Interface: Interfaces defines the operations that can be inherited by a user-defined object. Interfaces are non-instantiable. All objects inherit basic operations (like copy object, delete object) from the interface of Objects. A collection object inherits operations – such as, like an operation to determine empty collection – from the basic collection interface.

Atomic Objects: An atomic object is an object that is not of a collection type. They are user defined objects that are specified using *class* keyword. The properties of an atomic object can be defined by its attributes and relationships. An example is the book object given in the next sub-section. **Please note** here that a *class* is instantiable.

Inheritance: The interfaces specify the abstract operations that can be inherited by classes. This is called behavioural inheritance and is represented using ": " symbol. Sub-classes can inherit the state and behaviour of super-class(s) using the keyword EXTENDS.

Extents: An extent of an object that contains all the persistent objects of that class. A class having an extent can have a key.

In the following section we shall discuss the use of the ODL and OML to implement object models.

1.4.2 Object Definition Language

Object Definition Language (ODL) is a standard language on the same lines as the DDL of SQL, that is used to represent the structure of an object-oriented database. It uses unique object identity (OID) for each object such as library item, student, account, fees, inventory etc. In this language objects are treated as records. Any class in the design process has three properties that are attribute, relationship and methods. A class in ODL is described using the following syntax:

```
class <name>
{
      <list of properties>
};
```

Here, class is a key word, and the properties may be attribute method or relationship. The attributes defined in ODL specify the features of an object. It could be simple, enumerated, structure or complex type.

```
class Book
{
      attribute string ISBNNO;
      attribute string TITLE;
      attribute enum CATEGORY
            {text,reference,journal}    BOOKTYPE;
      attribute struct AUTHORS
            {string   fauthor,   string   sauthor,   string
            tauthor}
                                        AUTHORLIST;
};
```

Please note that, in this case, we have defined authors as a structure, and a new field on book type as an enum.

These books need to be issued to the students. For that we need to specify a relationship. The relationship defined in ODL specifies the method of connecting one object to another. We specify the relationship by using the keyword "relationship". Thus, to connect a student object with a book object, we need to specify the relationship in the student class as:

```
relationship set <Book> receives
```

Here, for each object of the class student there is a reference to book object and the set of references is called receives.

But if we want to access the student based on the book then the "inverse relationship" could be specified as

```
relationship set <Student> receivedby
```

We specify the connection between the relationship receives and receivedby by, using a keyword "inverse" in each declaration. If the relationship is in a different class, it is referred to by the relationships name followed by a double colon(::) and the name of the other relationship.

The relationship could be specified as:

```
class Book
{
      attribute string ISBNNO;
      attribute string TITLE;
      attribute integer PRICE;
      attribute string PUBLISHER;
      attribute enum CATEGORY
            {text,reference}BOOKTYPE;
      attribute struct AUTHORS
            {string   fauthor,   string   sauthor,   string
            tauthor} AUTHORLIST;
      relationship set <Student> receivedby
            inverse Student::receives;
      relationship set <Supplier> suppliedby
            inverse Supplier::supplies;
};
class Student
{
      attribute string ENROLMENT_NO;
      attribute string NAME;
      attribute integer MARKS;
      attribute string COURSE;
      relationship set <Book> receives
            inverse Book::receivedby;
};
class Supplier
{
      attribute string SUPPLIER_ID;
      attribute string SUPPLIER_NAME;
      attribute string SUPPLIER_ADDRESS;
      attribute string SUPPLIER_CITY;
      relationship set <Book> supplies
            inverse Book::suppliedby;
};
```

Methods could be specified with the classes along with input/output types. These declarations are called "signatures". These method parameters could be in, out or inout. Here, the first parameter is passed by value whereas the next two parameters are passed by reference. Exceptions could also be associated with these methods.

```
class Student
{
      attribute string ENROLMENT_NO;
      attribute string NAME;
      attribute string st_address;
      relationship set <book> receives
```

```
            inverse Book::receivedby;
      void findcity(in set<string>,out set<string>)
            raises(notfoundcity);
};
```

In the method find city, the name of city is passed referenced, in order to find the name of the student who belongs to that specific city. In case blank is passed as parameter for city name then, the exception notfoundcity is raised.

The ODL could be atomic type or class names. The basic type uses many class constructors such as set, bag, list, array, dictionary and structure. We have shown the use of some in the example above. You may wish to refer to the further readings section.

Inheritance is implemented in ODL using subclasses with the keyword “extends”.

```
class Journal extends Book
{
      attribute string VOLUME;
      attribute string emailauthor1;
      attribute string emailauthor2;
};
```

Multiple inheritance is implemented by using extends separated by a colon (:). If there is a class Fee containing fees details then multiple inheritance could be shown as:

```
class StudentFeeDetail extends Student:Fee
{
      void deposit(in set <float>, out set <float>)
                  raises(refundToBeDone)
};
```

Like the difference between relation schema and relation instance, ODL uses the class and its extent (set of existing objects). The objects are declared with the keyword “extent”.

```
class Student (extent firstStudent)
{
      attribute string ENROLMENT_NO;
      attribute string NAME;
      ..........
};
```

It is not necessary in case of ODL to define keys for a class. But if one or more attributes have to be declared, then it may be done with the declaration on key for a class with the keyword “key”.

```
class student (extent firstStudent key ENROLMENT_NO)
{
      attribute string ENROLMENT_NO;
      attribute string NAME;
      ..........
};
```

Assuming that the ENROLMENT_NO and ACCESSION_NO forms a key for the issue table then:

```
class Issue (extent thisMonthIssue key (ENROLMENT_NO,
ACCESSION_NO))
```

```
{
      attribute string ENROLMENT_NO;
      attribute string ACCESSION_NO;
      ..........
};
```

The major considerations while converting ODL designs into relational designs are as follows:

a) It is not essential to declare keys for a class in ODL but in Relational design now attributes have to be created in order for it to work as a key.

b) Attributes in ODL could be declared as non-atomic whereas, in Relational design, they have to be converted into atomic attributes.

c) Methods could be part of design in ODL but, they can not be directly converted into relational schema although, the SQL supports it, as it is not the property of a relational schema.

d) Relationships are defined in inverse pairs for ODL but, in case of relational design, only one pair is defined.

For example, for the book class schema the relation is:

```
Book(ISBNNO,TITLE,CATEGORY,fauthor,sauthor,tauthor)
```

Thus, the ODL has been created with the features required to create an object oriented database in OODBMS. You can refer to the further readings for more details on it.

1.4.3 Object Query Language

Object Query Language (OQL) is a standard query language which takes high-level, declarative programming of SQL and object-oriented features of OOPs. Let us explain it with the help of examples.

Find the list of authors for the book titled “The suitable boy”

```
SELECT b.AUTHORS
FROM Book b
WHERE b.TITLE=”The suitable boy”
```

The more complex query to display the title of the book which has been issued to the student whose name is Anand, could be

```
SELECT b.TITLE
FROM Book b, Student s
WHERE s.NAME =”Anand”
```

This query is also written in the form of relationship as

```
SELECT b.TITLE
FROM Book b
WHERE b.receivedby.NAME =”Anand”
```

In the previous case, the query creates a bag of strings, but when the keyword DISTINCT is used, the query returns a set.

```
SELECT DISTINCT b.TITLE
FROM Book b
```

```
WHERE b.receivedby.NAME =”Anand”
```

When we add ORDER BY clause it returns a list.

```
SELECT b.TITLE
FROM Book b
WHERE b.receivedby.NAME =”Anand”
ORDER BY b.CATEGORY
```

In case of complex output the keyword “Struct” is used. If we want to display the pair of titles from the same publishers then the proposed query is:

```
SELECT DISTINCT Struct(book1:b1,book2:b2)
FROM Book b1,Book b2
WHERE b1.PUBLISHER =b2.PUBLISHER
AND b1.ISBNNO < b2.ISBNNO
```

Aggregate operators like SUM, AVG, COUNT, MAX, MIN could be used in OQL. If we want to calculate the maximum marks obtained by any student then the OQL command is

```
Max(SELECT s.MARKS FROM Student s)
```

Group by is used with the set of structures, that are called “immediate collection”.

```
SELECT cour, publ,
                AVG(SELECT p.b.PRICE FROM partition
          p)
FROM Book b
GROUP BY cour:b.receivedby.COURSE, publ:b.PUBLISHER
```

HAVING is used to eliminate some of the groups created by the GROUP by commands.

```
SELECT cour, publ,
                AVG(SELECT p.b.PRICE FROM partition
          p)
FROM Book b
GROUP BY cour:b.receivedby.COURSE, publ:b.PUBLISHER

HAVING AVG(SELECT p.b.PRICE FROM partition p)>=60
```

Union, intersection and difference operators are applied to set or bag type with the keyword UNION, INTERSECT and EXCEPT. If we want to display the details of suppliers from PATNA and SURAT then the OQL is

```
(SELECT DISTINCT su
FROM Supplier su
WHERE su.SUPPLIER_CITY=”PATNA”)
UNION
(SELECT DISTINCT su
FROM Supplier su
WHERE su.SUPPLIER_CITY=”SURAT”)
```

The result of the OQL expression could be assigned to host language variables. If, costlyBooks is a set <book> variable to store the list of books whose price is below Rs.200 then

```
costlyBooks =   SELECT DISTINCT b
```

```
                    FROM Book b
                    WHERE b.PRICE > 200
```

In order to find a single element of the collection, the keyword "ELEMENT" is used. If costlySBook is a variable then

```
costlySBook =   ELEMENT (SELECT DISTINCT b
                         FROM Book b
                         WHERE b.PRICE > 200
                         )
```

The variable could be used to print the details a customised format.

```
bookDetails =   SELECT DISTINCT b
                FROM Book b
                ORDER BY b.PUBLISHER,b.TITLE;
bookCount = COUNT(bookDetails);
for (i=0;i<bookCount;i++)
{
     nextBook = bookDetails[i];
     cout<<i<<"\t"<<nextBook.PUBLISHER <<"\t"<<
                          nextBook.TITLE<<"\n";

}
```

☞ **Check Your Progress 2**

1) Create a class staff using ODL that also references the Book class given in section 1.5.

……..

……

……

2) What modifications would be needed in the Book class because of the table created by the above query?

……..

……

……

3) Find the list of books that have been issued to "Shashi".

……..

……

……

1.5 IMPLEMENTATION OF OBJECT ORIENTED CONCEPTS IN DATABASE SYSTEMS

Database systems that support object oriented concepts can be implemented in the following ways:

- Extend the existing RDBMSs to include the object orientation; Or

- Create a new DBMS that is exclusively devoted to the Object oriented database.

Let us discuss more about them.

1.5.1 The Basic Implementation Issues for Object-Relational Database Systems

The RDBMS technology has been enhanced over the period of last two decades. The RDBMS are based on the theory of relations and thus are developed on the basis of proven mathematical background. Hence, they can be proved to be working correctly. Thus, it may be a good idea to include the concepts of object orientation so that, they are able to support object-oriented technologies too. The first two concepts that were added include the concept of complex types, inheritance, and some newer types such as multisets and arrays. One of the key concerns in object-relational database are the storage of tables that would be needed to represent inherited tables, and representation for the newer types.

One of the ways of representing inherited tables may be to store the inherited primary key attributes along with the locally defined attributes. In such a case, to construct the complete details for the table, you need to take a join between the inherited table and the base class table.

The second possibility here would be, to allow the data to be stored in all the inherited as well as base tables. However, such a case will result in data replication. Also, you may find it difficult at the time of data insertion.

As far as arrays are concerned, since they have a fixed size their implementation is straight forward However, the cases for the multiset would desire to follow the principle of normalisation in order to create a separate table which can be joined with the base table as and when required.

1.5.2 Implementation Issues of OODBMS

The database system consists of persistent data. To manipulate that data one must either use data manipulation commands or a host language like C using embedded command. However, a persistent language would require a seamless integration of language and persistent data.

Please note: The embedded language requires a lot many steps for the transfer of data from the database to local variables and vice-versa. The question is, can we implement an object oriented language such as C++ and Java to handle persistent data? Well a persistent object-orientation would need to address some of the following issues:

Object persistence: A practical approach for declaring a persistent object would be to design a construct that declares an object as persistent. The difficulty with this approach is that it needs to declare object persistence at the time of creation, An alternative of this approach may be to mark a persistent object during run time. An interesting approach here would be that once an object has been marked persistent then all the objects that are reachable from that object should also be persistent automatically.

Object Identity: All the objects created during the execution of an object oriented program would be given a system generated object identifier, however, these identifiers become useless once the program terminates. With the persistent objects it is necessary that such objects have meaningful object identifiers. Persistent object identifiers may be implemented using the concept of persistent pointers that remain valid even after the end of a program.

Storage and access: The data of each persistent object needs to be stored. One simple approach for this may be to store class member definitions and the implementation of methods as the database schema. The data of each object, however, needs to be stored individually along with the schema. A database of such objects may require the collection of the persistent pointers for all the objects of one database together. Another, more logical way may be to store the objects as collection types such as sets. Some object oriented database technologies also define a special collection as **class extent** that keeps track of the objects of a defined schema.

1.6 OODBMS VERSUS OBJECT RELATIONAL DATABASE

An object oriented database management system is created on the basis of persistent programming paradigm whereas, a object relational is built by creating object oriented extensions of a relational system. In fact both the products have clearly defined objectives. The following table shows the difference among them:

Object Relational DBMS	Object Oriented DBMS
The features of these DBMS include: • Support for complex data types • Powerful query languages support through SQL • Good protection of data against programming errors	The features of these DBMS include: • Supports complex data types, • Very high integration of database with the programming language, • Very good performance • But not as powerful at querying as Relational.
One of the major assets here is SQL. Although, SQL is not as powerful as a Programming Language, but it is none-the-less essentially a fourth generation language, thus, it provides excellent protection of data from the Programming errors.	It is based on object oriented programming languages, thus, are very strong in programming, however, any error of a data type made by a programmer may effect many users.
The relational model has a very rich foundation for query optimisation, which helps in reducing the time taken to execute a query.	These databases are still evolving in this direction. They have reasonable systems in place.
These databases make the querying as simple as in relational even, for complex data types and multimedia data.	The querying is possible but somewhat difficult to get.
Although the strength of these DBMS is SQL, it is also one of the major weaknesses from the performance point of view in memory applications.	Some applications that are primarily run in the RAM and require a large number of database accesses with high performance may find such DBMS more suitable. This is because of rich programming interface provided by such DBMS. However, such applications may not support very strong query capabilities. A typical example of one such application is databases required for CAD.

☞ Check Your Progress 3

State True or False.

1) Object relational database cannot represent inheritance but can represent complex database types. T ☐ F ☐

2) Persistence of data object is the same as storing them into files. T ☐ F ☐

3) Object- identity is a major issue for object oriented database especially in the context of referencing the objects. T ☐ F ☐

4) The class extent defines the limit of a class. T ☐ F ☐

5) The query language of object oriented DBMS is stronger than object relational databases. T ☐ F ☐

6) SQL commands cannot be optimised. T ☐ F ☐

7) Object oriented DBMS support very high integration of database with OOP. T ☐ F ☐

1.7 SUMMARY

Object oriented technologies are one of the most popular technologies in the present era. Object orientation has also found its way into database technologies. The object oriented database systems allow representation of user defined types including operation on these types. They also allow representation of inheritance using both the type inheritance and the table inheritance. The idea here is to represent the whole range of newer types if needed. Such features help in enhancing the performance of a database application that would otherwise have many tables. SQL support these features for object relational database systems.

The object definition languages and object query languages have been designed for the object oriented DBMS on the same lines as that of SQL. These languages tries to simplify various object related representations using OODBMS.

The object relational and object oriented databases do not compete with each other but have different kinds of applications areas. For example, relational and object relational DBMS are most suited for simple transaction management systems, while OODBMS may find applications with e- commerce, CAD and other similar complex applications.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The object oriented databases are need for:

- Representing complex types.
- Representing inheritance, polymorphism
- Representing highly interrelated information
- Providing object oriented solution to databases bringing them closer to OOP.

2) Primarily by representing it as a single attribute. All its components should also be referenced separately.

3)

```
CREATE TYPE Addrtype AS
     (
          houseNo    CHAR(8),
          street     CHAR(10),
          colony     CHAR(10),
          city       CHAR(8),
          state CHAR(8),
          pincode    CHAR(6),
     );
```

```
      METHOD pin() RETURNS CHAR(6);
      CREATE METHOD pin() RETURNS CHAR(6);
      FOR Addrtype
      BEGIN
      . . . . .
      END
```

4)

```
CREATE TABLE address OF Addrtype
(
      REF IS addid SYSTEM GENERATED,
      PRIMARY KEY (houseNo,pincode)
};
```

5) The relationship can be established with multiple tables by specifying the keyword "SCOPE". For example:

```
Create table mylibrary
{
      mybook REF(Book) SCOPE library;
      myStudent REF(Student) SCOPE student;
      mySupplier REF(Supplier) SCOPE supplier;
};
```

Check Your Progress 2

1)

```
class Staff
{
      attribute string STAFF_ID;
      attribute string STAFF_NAME;
      attribute string DESIGNATION;
      relationship set <Book> issues
            inverse Book::issuedto;
};
```

2) The Book class needs to represent the relationship that is with the Staff class. This would be added to it by using the following commands:

```
RELATIONSHIP SET < Staff > issuedto
      INVERSE :: issues Staff
```

3)

```
SELECT DISTINCT b.TITLE
FROM BOOK b
WHERE b.issuedto.NAME = "Shashi"
```

Check Your Progress 3

1) False 2) False 3) True 4) False 5) False 6) False 7) True