**Question 1. What is difference between object oriented Vs Object relation database?**

**Answer:** An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it, and these messages are implemented by the methods

Difference between object oriented and object relation database

| **Object Oriented database** | **Object Relational database** |
|---|---|
| Built around persistent programming language | Built on top of relational model |
| Persistent programming languages are more susceptible to data corruption by programming error. | SQL language provide good protection of data from programming error, and make high level of optimization , such as reduced I/O, relatively easy |
| Persistent programming language has high level of performance. They provide low overhead access to persistent data and eliminate the need to data translations it data required to be manipulated by programming language. | SQL imposes a significant performance penalty for those type of application which run primarily run in main memory and that perform a larage number of access to database. |

**Question 2. Why is it not possible to model overlapping generalization of ER Model, using SQL statement?**
**Answer:**

Multiple inheritances of tables are not supported by the SQL.

In table inheritance, The consistency requirement of subtable are:

1. Each tuple in supertable can corresponds to at most one in each of its immediate subtable.
2. SQL has additional constraints that all the tuple corresponding to each other must be derived from one tuple.

**Question 3. How do you store multivalued attributes in object relational database using SQL ? Illustrate with example.**

**Answer:** For each attribute, there is a set of permitted value, called the domain, or value set, of that attribute. If an attribute has a set of values for a specific entity, it is called multivalued. An employee may have zero, one or several phone number, and different employee may have different number of phones. This type of attribute is said to be multivalued.

Multivalued attribute can be stored in database using collection types : SQL support tow collection type: array and multiset . The elements of an array are ordered whereas multiset is an unordered collection, where an element may occur multiple times.

Example:

We want to record information about book including set of keywords for each book and name of authors of book. Name of authors can be stored in an array, unlike elements in multiset, the element of an array are ordered, so we can distinguish the first author from the second author, keyword can be stored in multiset.

```
create type Publisher as
                (name varchar(20),
                 branch varchar(20))
```

```
create type Book as
          (  title varchar(20),
             author_array varchar(2) array[10],
             pub_date  date,
              publisher Publisher,
            keyword_set  varchar(20), multiset)

   Create table book of Book.
```

We have used array instead of multiset, to store the names of authors, since the ordering of authors generally has some significance, where as the ordering of keyword associated with book is not significant.

Insert record in book table

```
 Insert into book
     Value('compilers',array['Smith','Jone'],
           New Publisher ('Magraw-Hill','New York'),
         Multiset['parsing', 'analysis'])
```

**Question 4. How do initialize a self-referential attribute? Illustrate with example?**

**or**

**How do you use SQL reference type to declare a Foreign key? What are its advantages? Illustrate with examples?**

**Answer:** Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL we can define a type Department with a field name and a field head that is a reference to the type Person, and a table departments of type Department, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people
);
create table departments of Department;
```

Here, the reference is restricted to tuples of the table people. The restriction of the scope of a reference to tuples of a table is mandatory in SQL, and it makes references behave like foreign keys. We can omit the declaration scope people from the type declaration and instead make an addition to the create table statement:

```
create table departments of Department
        (head with options scope people);
```

The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the self-referential attribute, by adding a ref is clause to the create table statement:

```
create table people of Person
ref is person_id system generated;
```

Here, person id is an attribute name, not a keyword, and the create table statement specifies that the identifier is generated automatically by the database. In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query. Thus, to create a tuple with the reference value, we may first create the tuple with a null reference and then set the reference separately:

```
insert into departments
values ('CS', null);
update departments
set head = (select p.person id
from people as p
where name = 'John')
where name = 'CS';
```

An alternative to system-generated identifiers is to allow users to generate identifiers. The type of the self-referential attribute must be specified as part of the type definition of the referenced table, and the table definition must specify that the reference is user generated:

```
create type Person
(name varchar(20),
address varchar(20))
ref using varchar(20);
create table people of Person
ref is person id user generated;
```

When inserting a tuple in people, we must then provide a value for the identifier:

```
insert into people (person id, name, address) values
('01284567', 'John', '23 Coyote Run');
```

No other tuple for people or its supertables or subtables can have the same identifier. We can then use the identifier value when inserting a tuple into departments, without the need for a separate query to retrieve the identifier:

```
insert into departments
values ('CS', '01284567');
```

It is even possible to use an existing primary-key value as the identifier, by including the ref from clause in the type definition:

```
create type Person
(name varchar(20) primary key,
address varchar(20))
ref from(name);
create table people of Person
ref is person id derived;
```

Note that the table definition must specify that the reference is derived, and must still specify a self-referential attribute name. When inserting a tuple for departments, we can then use:

```
insert into departments
values ('CS', 'John');
```

**Question 5 : Can you use unnest operator to access elements of an array ? If so how ?**
**Answer :**

The transformation of a nested relation to a fewer number of relations is called unnesting. Unnest operator can be used to access individual elements of an array.
Create a structured type *Book* consists of two attributes title and author_array, which is an array of authors.

```
create type Book as
( title         varchar(20),
author_array  varchar(20) array [10]);
```

Now Create a table *books* of type *Book.*

```
create table books of Book;
```

Now unnest operator can be used on author_array to access individual elements of array. The SQL statement for selecting title and array elements is given below:

```
select B.title, A.author
from books as B, unnest(B.author array) as A(author);
```

Since the *author array* attribute of *books* is a collection-valued field, **unnest**(*B.author_array*) can be used in a **from** clause, where a relation is expected. Note that the tuple variable *B* is visible to this expression since it is defined *earlier* in the **from** clause.

**Question 6. Explain about table inheritance in SQL:**

**Create table people of person; its super table.**
**Create table students of student Under people ;**
**Create table teachers of teacher under people;**

In this people is super table, and students,teachers are sub tables under people.
So any tuple present in people will also be implicitly present instudent or teacher as we declared the subtables under people.
And also attributes present in subtables will also present in people.
So there will be duplicate tuples present in the sub table
*So whenever we querryfor some attribute ,then all three tables will display the reults. So to get tuples from people "only" conjection is used in selection .*

Subtables in SQL correspond to the E-R notion of specialization/generalization. The types of the subtables are subtypes of the type of the parent table . As a result, every attribute present in the table is also present in the subtables
Conceptually, multiple inheritance is possible with tables, just as it is possible with types. For example, we can create a table of type TeachingAssistant:

**create table teaching assistants**
**of TeachingAssistant**
**under students, teachers;**

Tuples in a subtable and parent table correspond if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.
The consistency requirements for subtables are:
1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).
For example, without the first condition, we could have two tuples in students (or teachers) that correspond to the same person. The second condition rules out a tuple in people corresponding to both a tuple in students and a tuple in teachers, unless all these tuples are implicitly present because a tuple was inserted in a table teaching assistants, which is a subtable of both teachers and students. **Since SQL does not support multiple inheritance, the second condition actually prevents a person from being both a teacher and a student. Even if multiple inheritance were supported, the same problem would arise if the subtable teaching assistants were absent. Obviously it would be useful to model a situation where a person can be a teacher and a student, even if a common subtable teaching assistants is not present. Thus, it can be useful to remove the second consistency constraint. Doing would allow an object to have multiple types, without requiring it to have a most-specific type.**

**Question 7. How do you crete,access,and query collection valued attributes in SQL?Illustrate with example?**

**OR**

**Explain about nesting and unnesting operation in object relational system?**

We create a collection in databse,for arrays,multiset values to store authors for a book, and multiple keywords that may be used in this book type as below.
**Ex: Create type book as(title vachar(20),author_array varchar(20) array[10],**
**Publisher varchar(20), keyword_set varchar(20) multiset);**
Create table books of book;

We can create collections like
**array[‘smith’,’john’,’jones’]**
**multiset[‘database’,’complers’,’SQL’]**
we can create tuples books type as
**(‘compilers’,array[‘smith’,’john’],’MCGrawhills’,multiset[‘database’,SQL’])**

To insert into books:

**Insert into books**

```
values('compilers',array['smith','john'],'MCGrawhills',multiset['database',SQ
L']);
```

We can querry the collections as below:

```
Ex: select title from books
        Where 'database' in (unnest(keyword_set));
```

Here a multiset keyword_set is unnested ,to get each word from it and can be compared to a string 'database' and only those matching with this string only selected from books;

**Ans1.** The transformation of a nested relation into a form with fewer (or no) relation valued attributes is called unnesting. The books relation has two attributes, author array and keyword set, that are collections, and two attributes, title and publisher, that are not. Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

```
select title, A.author, publisher.name as pub name, publisher.branch
        as pub branch, K.keyword
from books as B, unnest(B.author array) as A(author),
        unnest (B.keyword set) as K(keyword);
```

The variable B in the from clause is declared to range over books. The variable A is declared to range over the authors in author array for the book B, and K is declared to range over the keywords in the keyword set of the book B.

Nesting is the opposite of unnesting, creating a collection-valued attribute. Nesting can be done in a manner similar to aggregation, but using the function collect() in place of an aggregation operation, to create a multiset. To nest the flat_books relation on the attribute keyword:

```
select title, author, Publisher (pub_name, pub_branch ) as publisher,
        collect (keyword)  as keyword_set
from flat_books
group by title, author, publisher
```

If we want to nest the author attribute also into a multiset, we can use the query:

```
select title, collect (author ) as author_set,
         Publisher (pub_name, pub_branch) as publisher,
              collect  (keyword ) as keyword_set
from   flat_books
group by title, publisher
```

Another approach to creating nested relations is to use subqueries in the select clause. An advantage of the subquery approach is that an order by clause can be used in the subquery to generate results in the order desired for the creation of an array. The following query illustrates this approach; the keywords array and multiset specify that an array and multiset (respectively) are to be created from the results of the subqueries.

```
select title,
        array (select author
        from authors as A
```

```
                where A.title = B.title
                order by A.position) as author_array,
                Publisher (pub-name, pub-branch) as publisher,
                multiset (select keyword
                from keywords as K
                where K.title = B.title) as keyword_set,
        from books4 as B;
```

**8.What are different approaches to make object persistent? Which of these approaches makes an entire data structure persistent by declaring its root as persistent?**

Object-oriented programming languages already have a concept of objects, a type system to define object types, and constructs to create objects. However, these objects are *transient*—they vanish when the program terminates. If we wish to turn such a language into a database programming language, the first step is to provide a way to make objects persistent. Several approaches have been proposed.

- **Persistence by class.** The simplest way is to declare a class is persistent. All objects of the class are then persistent objects by default. Objects of non-persistent classes are all transient. This approach is not flexible, since it is often useful to have both transient and persistent objects in a single class.
- **Persistence by creation.** In this approach, new syntax is introduced to create persistent objects, by extending the syntax for creating transient objects. Thus, an object is either persistent or transient, depending on how it was created. Several object-oriented database systems follow this approach.
- **Persistence by marking.** In this approach, mark objects as persistent after they are created. All objects are created as transient objects, but, if an object is to persist beyond the execution of the program, it must be marked explicitly as persistent before the program terminates. This approach postpones the decision on persistence or transience until after the object is created.
- **Persistence by reachability.** One or more objects are explicitly declared as (root) persistent objects. All other objects are persistent if they are reachable from the root object through a sequence of one or more references. Thus, all objects referenced by the root persistent objects are persistent. But also, all objects referenced from these objects are persistent, and objects to which they refer are in turn persistent, and so on.

A benefit of this scheme is that it is easy to make entire data structures persistent by merely declaring the root of such structures as persistent.

**9.What is difference between object relational mapping systems and persistent programming language?**

**Answer:**

Object relational mapping system is built on the top of traditional relational database and allow the programmer to define mapping between tuple in database relation and object in programming language. Objects are transient and there is no permanent object identity. Persistence programming language, objects are not transient and are permanent and stored in file.

**10. 11. EX: create type Name as(**
**Firstname varchar(20),Lastname varchar (20),Middlename varchar(20) )**
**final;**

**Here Name is a composite attribute with firstname,lastname,middlename as component attributes.**

```
Create type Address as(
Dno varchar(20), Street varchar(20),City varchar(20),zipcode varchar (20))
 final;


Create table  person(
name Name,address Address, dateofBirth date);
```

In this table to access first name city from the person table

```
Select name.firstname ,address.city from person;
```

. operator is used to access the components of composite attributes.

The same can be created using row types

```
Create table person_r(
name row (Firstname varchar(20),Lastname varchar(20), Middlename varchar(20)),
address row (Dno varchar(20),street varchar (20 ),city varchar(20)),
dateofbirth date);
```

**12.**
**Describe the features of persistent Java System ?**

**Answer:**

Java programs has JDO model of object persistence.

- **Persistence by reachability**: Objects are not explicitly created for database. Explicitly registering a object as persistent makes a object persistent.
- **Bytecode enhancement** : instead of declaring a class as persistent in java code, persistent classes are specified in registration ( with suffix .jdo)
- **Database mapping**:JDO does not define how data are stored in the database. The enhancer program may create an appropriate schema in database to store the class objects. Some attribute could be mapped as to relational attribute, which other may be stored in serialize form, treated as binary object by database.
- **Class extent**: class extent are created and maintained automatically for each class declared to be persistent. All objects made persistent are added automatically to be class extent corresponding their class.
- **Single Reference Type** : There is no difference in type between reference to transient object and reference to persistent object.