Institute of Distance and Open Learning

(IDOL)

**University of Mumbai**



Certificate

This is to certify that Mr. Pralhad Sahantaram pawar of **S.Y.MCA**  Semester 4 has completed the specified term work in the subject of Advance Database Techniques in satisfactorily manner within this institute as laid down by University of Mumbai during the academic year 2018 to 2019.

Subject In charge MCA Co-ordinator

(prof. Smita Patil) (prof. Mahendra Patil)

Examiner

Date: College Seal

Institute of Distance and Open Learning (IDOL)

**University of Mumbai**



INDEX

Subject: Advance Database Techniques

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr. No.** | **Practical** | **Pg. No.** | **Date** | **Grade** | **Sign.** | **Remarks** |
| **1** | To study EER diagram and map it to relational model. | 1-3 |  |  |  |  |
| **2** | To implement views, trigger and assertion. | 4-9 |  |  |  |  |
| **3** | To implement nested and recursive queries. | 10-14 |  |  |  |  |
| **4** | To implement Authorization in Database. | 15-17 |  |  |  |  |
| **5** | To implement database connectivity using JDBC. | 18-19 |  |  |  |  |
| **6** | To implement database hashing technique. | 20-24 |  |  |  |  |
| **7** | To Create and querying an Object database. – Using ODL and OQL. | 25-28 |  |  |  |  |
| **8** | To implement fragmentation in Distributed Database. | 29-32 |  |  |  |  |
| **9** | To study database security techniques – SQL injection. | 33-34 |  |  |  |  |
| **10** | To Design a Data Warehouse using Star Schema a Snowflake Schema. | 35-37 |  |  |  |  |
| **11** | To implement queries for OLAP operations. ( Roll up, Drill down, Slice,  Dice, pivot) | 38-44 |  |  |  |  |

Date: College Seal

**PRACTICAL NO. 1**

**AIM: - To study EER diagram and map it to relational model.**

**THEORY:**

**Step 1:**​ Mapping of Regular Entity Types

For each regular entity type, create a relation R that includes all the simple attributes of E entity relations Each tuple represents an entity instance

**Step 2:​** Mapping of Weak Entity Types

For each weak entity type, create a relation R and include all simple attributes of the entity type as attributes of R

Include primary key attribute of owner as foreign key attributes of R

**Step 3:​** Mapping of Binary 1:1 Relationship Types

For each binary 1:1 relationship type

• Identify relations that correspond to entity types participating in R Possible approaches:

• Foreign key approach

• Merged relationship approach

• Cross reference or relationship relation

**Step 4:**​ Mapping of Binary 1:N Relationship Types

For each regular binary 1:N relationship type

• Identify relation that represents participating entity type at N-side of relationship type

• Include primary key of other entity type as foreign key in S

• Include simple attributes of 1:N relationship type as attributes of Alternative approach

• Use the relationship relation (cross-reference) option as in the third option for binary 1:1 relationships

**Step 5:​** Mapping of Binary M:N Relationship Types

For each binary M:N relationship type

• Create a new relation S

• Include primary key of participating entity types as foreign key attributes in S

• Include any simple attributes of M:N relationship type

**Step 6:​** Mapping of Multivalued Attributes

For each multivalued attribute

• Create a new relation

• Primary key of R is the combination of A and K

• If the multivalued attribute is composite, include its simple components

**Step 7:​** Mapping of N-ary Relationship Types

For each n-ary relationship type R

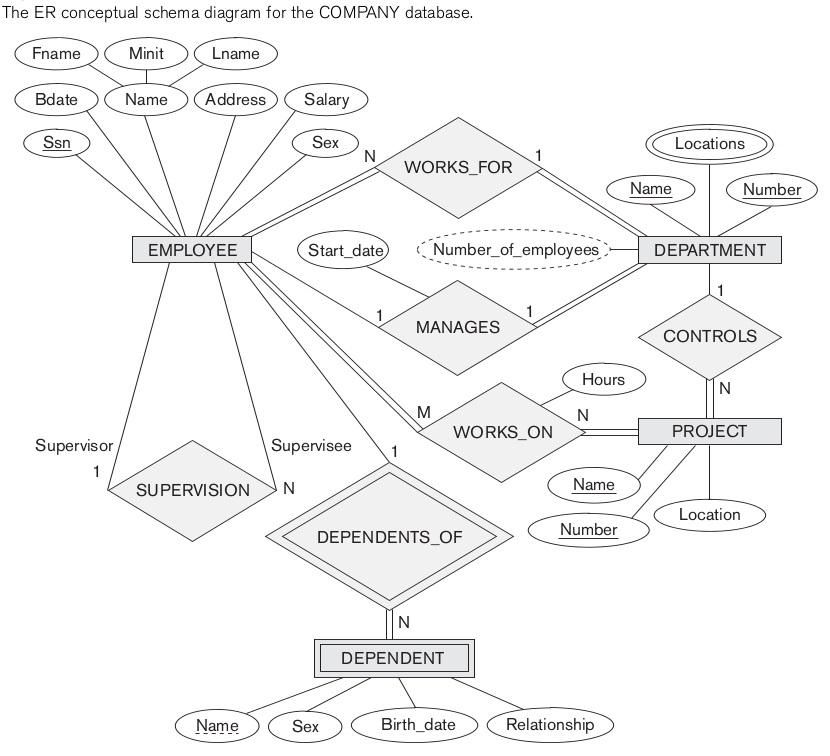
• Create a new relation S to represent R

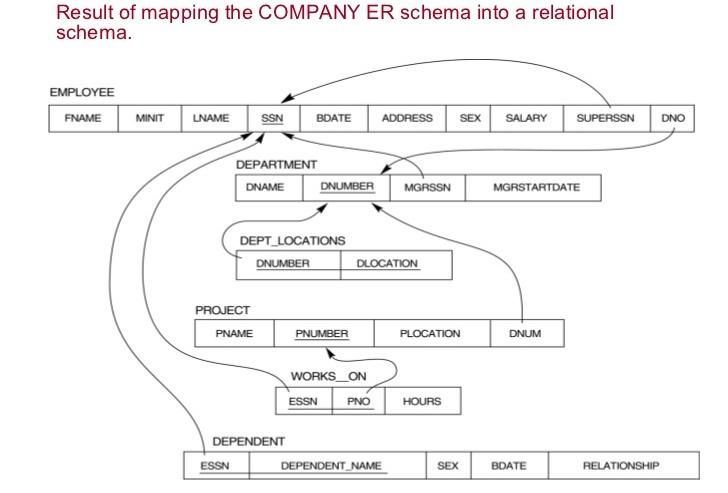
• Include primary keys of participating entity types as foreign keys

• Include any simple attributes as attributes

**CONCLUSION**:

The concepts of mapping a given EER diagram into Relational model was successfully studied and implemented.





**PRACTICAL NO.2**

**AIM: - To study EER diagram and map it to relational model.**

**THEORY:**

A view is nothing more than a SQL statement that is stored in the database with an associated name.

A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following:

* Structure data in a way that users or classes of users find natural or intuitive.
* Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
* Summarize data from various tables which can be used to generate reports.

**Creating Views:**

Database views are created using the ​CREATE VIEW​ statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific

implementation.

The basic CREATE VIEW syntax is as follows:

*CREATE VIEW view\_name AS*

*SELECT column1, column2.....*

*FROM table\_name*

*WHERE [condition];*

You can include multiple tables in your SELECT statement in very similar way as you use them in

normal SQL SELECT query.

**Example:**

Consider the CUSTOMERS table having the following records:

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

*SQL > CREATE VIEW CUSTOMERS\_VIEW AS*

*SELECT name, age*

*FROM CUSTOMERS;*

Now, you can query CUSTOMERS\_VIEW in similar way as you query an actual table. Following is the example: SQL > SELECT \* FROM CUSTOMERS\_VIEW; This would produce the following result:

*SQL > SELECT \* FROM CUSTOMERS\_VIEW;*

This would produce the following result:

*+----------+-----+*

*| name | age |*

*+----------+-----+*

*| Ramesh | 32 |*

*| Khilan | 25 |*

*| kaushik | 23 |*

*| Chaitali | 25 |*

*| Hardik | 27 |*

*| Komal | 22 |*

*| Muffy | 24 |*

*+----------+-----+*

**Updating a View:**

* A view can be updated under certain conditions:
* The SELECT clause may not contain the keyword DISTINCT.
* The SELECT clause may not contain summary functions.
* The SELECT clause may not contain set functions.
* The SELECT clause may not contain set operators.
* The SELECT clause may not contain an ORDER BY clause.
* The FROM clause may not contain multiple tables.
* The WHERE clause may not contain subqueries.
* The query may not contain GROUP BY or HAVING.
* Calculated columns may not be updated.
* All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above-mentioned rules then you can update a view. Following is an

example to update the age of Ramesh:

*SQL > UPDATE CUSTOMERS\_VIEW*

*SET AGE = 35*

*WHERE name='Ramesh';*

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself.

Now, try to query base table, and SELECT statement would produce the following result:

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+-----------+----------+*

**Deleting Rows into a View:**

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT

commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

*SQL > DELETE FROM CUSTOMERS\_VIEW*

*WHERE age = 22;*

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+-----------+----------+*

**Dropping Views:**

Obviously, where you have a view, you need a way to drop the view if it is no longer needed.

Following is an example to drop CUSTOMERS\_VIEW from CUSTOMERS table:

*DROP VIEW CUSTOMERS\_VIEW;*

**Triggers**

Triggers are stored programs, which are automatically executed or fired when some events occur.

Triggers are, in fact, written to be executed in response to any of the following events:

● A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).

● A database definition (DDL) statement (CREATE, ALTER, or DROP).

● A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

**Benefits of Triggers**

Triggers can be written for the following purposes:

● Generating some derived column values automatically

● Enforcing referential integrity

● Event logging and storing information on table access

● Auditing

● Synchronous replication of tables

● Imposing security authorizations

● Preventing invalid transactions

**Creating Triggers**

The syntax for creating a trigger is:

*CREATE [OR REPLACE ] TRIGGER trigger\_name*

*{BEFORE | AFTER | INSTEAD OF }*

*{INSERT [OR] | UPDATE [OR] | DELETE}*

*[OF col\_name]*

*ON table\_name*

*[REFERENCING OLD AS o NEW AS n]*

*[FOR EACH ROW]*

*WHEN (condition)*

*DECLARE*

*Declaration-statements*

*BEGIN*

*Executable-statements*

*EXCEPTION*

*Exception-handling-statements*

*END;*

Where,

● *CREATE [OR REPLACE] TRIGGER trigger\_name*: Creates or replaces an existing trigger

with the trigger\_name.

● *{BEFORE | AFTER | INSTEAD OF}* : This specifies when the trigger would be executed. The

INSTEAD OF clause is used for creating trigger on a view.

● *{INSERT [OR] | UPDATE [OR] | DELETE}*: This specifies the DML operation.

● *[OF col\_name]*: This specifies the column name that would be updated.

● *[ON table\_name]:* This specifies the name of the table associated with the trigger.

● *[REFERENCING OLD AS o NEW AS n]*: This allows you to refer new and old values for

various DML statements, like INSERT, UPDATE, and DELETE.

● *[FOR EACH ROW]*: This specifies a row level trigger, i.e., the trigger would be executed for

each row being affected. Otherwise the trigger will execute just once when the SQL statement

is executed, which is called a table level trigger.

● *WHEN (condition)*: This provides a condition for rows for which the trigger would fire. This

clause is valid only for row level triggers.

**Example:**

To start with, we will be using the CUSTOMERS table we had created and used in the previous

chapters:

*Select \* from customers;*

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

The following program creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

*CREATE OR REPLACE TRIGGER display\_salary\_changes*

*BEFORE DELETE OR INSERT OR UPDATE ON customers*

*FOR EACH ROW*

*WHEN (NEW.ID > 0)*

*DECLARE*

*sal\_diff number;*

*BEGIN*

*sal\_diff := :NEW.salary - :OLD.salary;*

*dbms\_output.put\_line('Old salary: ' || :OLD.salary);*

*dbms\_output.put\_line('New salary: ' || :NEW.salary);*

*dbms\_output.put\_line('Salary difference: ' || sal\_diff);*

*END;*

*/*

When the above code is executed at SQL prompt, it produces the following result:

Trigger created.

Here following two points are important and should be noted carefully:

● OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.

● If you want to query the table in the same trigger, then you should use the AFTER keyword,

because triggers can query the table or change it again only after the initial changes are applied

and the table is back in a consistent state.

● Above trigger has been written in such a way that it will fire before any DELETE or INSERT

or UPDATE operation on the table, but you can write your trigger on a single or multiple

operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Now, let us perform DML operation on the CUSTOMERS table. Here is one UPDATE statement,

which will update an existing record in the table:

*UPDATE customers*

*SET salary = salary + 500*

*WHERE id = 2;*

When a record is updated in CUSTOMERS table, above create trigger display\_salary\_changes will be fired and it will display the following result:

*Old salary: 1500*

*New salary: 2000*

*Salary difference: 500*

**Assertions**

1. An assertion is a predicate expressing a condition we wish the database to always satisfy.

2. Domain constraints, functional dependency and referential integrity are special forms of

assertion.

3. Where a constraint cannot be expressed in these forms, we use an assertion, e.g.

- Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

- Ensuring every loan customer keeps a minimum of $1000 in an account.

**Syntax:**

*create assertion assertion-name check predicate*

Two assertions mentioned above can be written as follows.

Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

*create assertion sum-constraint check*

*(not exists (select \* from branch*

*where (select sum)amount) from loan*

*where (loan.bname = branch.bname >=*

*(select sum)amount) from account*

*where (account.bname = branch.bname)))*

Ensuring every loan customer keeps a minimum of $1000 in an account.

*create assertion balance-constraint check*

*(not exists (select \* from loan L*

*(where not exists (select \* from borrower B, depositor D, account A*

*where L.loan# = B.loan# and B.cname = D.cname*

*and D.account# = A.account#*

*and A.balance >= 1000 )))*

When an assertion is created, the system tests it for validity.

If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated.

This testing may result in significant overhead if the assertions are complex. Because of this, the

assert should be used with great care.

Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

**PRACTICAL NO.3**

**AIM: - To implement nested and recursive queries.**

**THEORY:**

**Nested Queries:**

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

● Subqueries must be enclosed within parentheses.

● A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.

● An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.

● Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.

● The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY,

CLOB, or NCLOB.

● A subquery cannot be immediately enclosed in a set function.

● The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator

can be used within the subquery.

**Subqueries with the SELECT Statement:**

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

*SELECT column\_name [, column\_name ]*

*FROM table1 [, table2 ]*

*WHERE column\_name OPERATOR*

*(SELECT column\_name [, column\_name ]*

*FROM table1 [, table2 ]*

*[WHERE])*

**Example:**

Consider the CUSTOMERS table having the following records:

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+-----------+----------+*

Now, let us check following subquery with SELECT statement:

*SQL> SELECT \* FROM CUSTOMERS WHERE ID IN (SELECT ID FROM CUSTOMERS*

*WHERE SALARY > 4500*) ;

This would produce the following result:

*+----+----------+-----+---------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+---------+----------+*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 8500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+---------+----------+*

**Subqueries with the INSERT Statement:**

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified withany of the character, date or number functions.

The basic syntax is as follows:

*INSERT INTO table\_name [ (column1 [, column2 ]) ]*

*SELECT [ \*|column1 [, column2 ]*

*FROM table1 [, table2 ]*

*[ WHERE VALUE OPERATOR ]*

**Example:**

Consider a table CUSTOMERS\_BKP with similar structure as CUSTOMERS table. Now to copy complete CUSTOMERS table into CUSTOMERS\_BKP, following is the syntax:

*SQL> INSERT INTO CUSTOMERS\_BKP*

*SELECT \* FROM CUSTOMERS*

*WHERE ID IN (SELECT ID FROM CUSTOMERS);*

**Subqueries with the UPDATE Statement:**

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple

columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

*UPDATE table*

*SET column\_name = new\_value*

*[ WHERE OPERATOR [ VALUE ]*

*(SELECT COLUMN\_NAME FROM TABLE\_NAME) [ WHERE) ]*

**Example:**

Assuming, we have CUSTOMERS\_BKP table available which is backup of CUSTOMERS table.

Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers

whose AGE is greater than or equal to 27:

*SQL> UPDATE CUSTOMERS*

*SET SALARY = SALARY \* 0.25*

*WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP*

*WHERE AGE >= 27 );*

This would impact two rows and finally CUSTOMERS table would have the following records:

*+----+----------+-----+-----------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+-----------+----------+*

*| 1 | Ramesh | 35 | Ahmedabad | 125.00 |*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 5 | Hardik | 27 | Bhopal | 2125.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+-----------+----------+*

**Subqueries with the DELETE Statement:**

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

*DELETE FROM TABLE\_NAME*

*[ WHERE OPERATOR [ VALUE ]*

*(SELECT COLUMN\_NAME*

*FROM TABLE\_NAME)*

*[ WHERE) ]*

**Example:**

Assuming, we have CUSTOMERS\_BKP table available which is backup of CUSTOMERS table.

Following example deletes records from CUSTOMERS table for all the customers whose AGE is

greater than or equal to 27:

*SQL> DELETE FROM CUSTOMERS*

*WHERE AGE IN (SELECT AGE FROM CUSTOMERS\_BKP*

*WHERE AGE > 27 );*

This would impact two rows and finally CUSTOMERS table would have the following records:

*+----+----------+-----+---------+----------+*

*| ID | NAME | AGE | ADDRESS | SALARY |*

*+----+----------+-----+---------+----------+*

*| 2 | Khilan | 25 | Delhi | 1500.00 |*

*| 3 | kaushik | 23 | Kota | 2000.00 |*

*| 4 | Chaitali | 25 | Mumbai | 6500.00 |*

*| 6 | Komal | 22 | MP | 4500.00 |*

*| 7 | Muffy | 24 | Indore | 10000.00 |*

*+----+----------+-----+---------+----------+*

**Recursive Queries**

Its useful for situations where you can have circular references in the data diagram, for example when the table includes a column which is a reference for another entry in the same table. Usually it would be necessary to write some function which could be called recursively and allow you to traverse this tree layout, however in Oracle there is the CONNECT BY PRIOR syntax.

The syntax is as follows:

*SELECT select\_list*

*FROM table\_expression*

*[ WHERE ... ]*

*[ START WITH start\_expression ]*

*CONNECT BY [NOCYCLE] { PRIOR parent\_expr = child\_expr | child\_expr = PRIOR parent\_expr*

*}*

*[ ORDER SIBLINGS BY column1 [ ASC | DESC ] [, column2 [ ASC | DESC ] ] ...*

*[ GROUP BY ... ]*

*[ HAVING ... ]*

*...*

START WITH specifies the root row(s) of the hierarchy.

CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.

● The NOCYCLE parameter instructs Oracle Database to return rows from a query even if a

CONNECT BY loop exists in the data. Use this parameter along with the CONNECT\_BY\_ISCYCLE

pseudocolumn to see which rows contain the loop.

In a hierarchical query, one expression in condition must be qualified with the PRIOR operator to refer the parent row. For example,

*..PRIOR expr = expr*

*or*

*... expr = PRIOR expr*

*SELECT LEVEL, LPAD (' ', 2 \* (LEVEL - 1)) || ename "employee", empno, mgr "manager"*

*FROM emp START WITH mgr IS NULL*

*CONNECT BY PRIOR empno = mgr;*

The output from the above query would look like:

*level | employee | empno | manager*

*-------+-------------+-------+---------*

*1 | KING | 7839 |*

*2 | JONES | 7566 | 7839*

*3 | SCOTT | 7788 | 7566*

*4 | ADAMS | 7876 | 7788*

*3 | FORD | 7902 | 7566*

*4 | SMITH | 7369 | 7902*

*2 | BLAKE | 7698 | 7839*

*3 | ALLEN | 7499 | 7698*

*3 | WARD | 7521 | 7698*

*3 | MARTIN | 7654 | 7698*

*3 | TURNER | 7844 | 7698*

*3 | JAMES | 7900 | 7698*

*2 | CLARK | 7782 | 7839*

*3 | MILLER | 7934 | 7782*

*(14 rows)*

**PRACTICAL NO. 4**

**AIM: - To implement Authorization in Database.**

**THEORY:**

**DCL** : Data control language(DCL) consists of various commands which are related to security of

data in database. These are

● GRANT

● REVOKE

**Granting Privileges:**

The rights that allow the use of some or all of Oracle’s resources on the server are called Privileges.

Objects that are created by a user are owned and controlled by that user. If a user wishes to access any of the objects belonging to another user, the owner of the object will have to give permission for such access. This is called Granting of Privileges.

**Granting Privileges Using the GRANT statement:**

The GRANT statement provides various types of access to database objects such as tables, views and so on.

**Syntax :**

*GRANT<Object Privileges>*

*ON<ObjectName>*

*TO<UserName>*

**OBJECT PRIVILEGES:**

Each Object Previleges that is granted authorizes the grantee to perform some operation on the object.

A user can gran all the privileges or grant only specific object privileges.

The list of object privileges is as follows:

● **ALTER** Allows the grantee to change the table definition with the ALTER TABLE command

● **DELETE** Allows the grantee to remove the records from the table with the DELETE command

● **INDEX** Allows the grantee to create an index on the table with the CREATE INDEX command

● **INSERT** Allows the grantee to add records to the table with the INSERTcommand

● **SELECT** Allows the grantee to query the table with the SELECT command

● **UPDATE** Allows the grantee to modify the records in the tables with the UPDATE command.

**REVOKING PRIVILEGES GIVEN:**

PRIVILEGES once given can be denied to a user using the REVOKE command. The object owner

can revoke PRIVILEGES granted to another user. A user of an object who is not the owner, but has been granted the GRANT PRiVILEGES, has the power to REVOKE the PRIVILEGES from a

grantee.

**REVOKING Permissions Using The REVOKE statement:**

The REVOKE statement is used to deny the grant given on an object.

**Syntax :**

*REVOKE <Object Privileges>*

*ON<ObjectName>*

*FROM<UserName>*

The REVOKE command cannot be used to revoke the privileges granted through the operating

system.

The REVOKE command is used to revoke object privileges that the user previously granted directly to the grantee.

*SQL> connect sys as sysdba*

*Connected.*

*SQL> create user user1 identified by aaa;*

*User created.*

*SQL> grant create session,create any table to user1;*

*Grant succeeded.*

*SQL> create user user2 identified by bbb;*

*User created.*

*SQL> grant create session, create any table to user2;*

*Grant succeeded.*

*SQL> connect user1/aaa;*

*Connected.*

*SQL> create table stud*

*2 (rollno number(5),*

*3 name varchar2(10),*

*4 marks number(5));*

*Table created.*

*SQL> grant select on stud to user2;*

*Grant succeeded.*

*SQL> grant insert on stud to user2;*

*Grant succeeded.*

*SQL> insert into stud*

*2 values(3,'abc',45);*

*1 row created.*

*SQL> insert into stud*

*2 values(2,'def',56);*

*1 row created.*

*SQL> select \* from stud;*

*ROLLNO NAME MARKS*

*---------- ---------- ----------*

*3 abc 45*

*2 def 56*

*SQL> connect user2/bbb;*

*Connected.*

*SQL> select \* from user1.stud*

*ROLLNO NAME MARKS*

*---------- ---------- ----------*

*3 abc 45*

*2 def 56*

*SQL> insert into user1.stud*

*2 values(5,'ghi',34);*

*1 row created.*

*SQL> connect user1/aaa;*

*Connected.*

*SQL> select \* from stud;*

*ROLLNO NAME MARKS*

*---------- ---------- ----------*

*3 abc 45*

*2 def 56*

*5 ghi 34*

**PRACTICAL NO.5**

**AIM: - To implement database connectivity using JDBC.**

**THEORY:**

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

● Making a connection to a database

● Creating SQL or MySQL statements

● Executing that SQL or MySQL queries in the database

● Viewing & Modifying the resulting records

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executable.

*SQL> create table student*

*2 (roll\_no number(5),*

*3 f\_name varchar2(10),*

*4 l\_name varchar2(10));*

*Table created.*

*SQL> insert into student*

*2 values(1,'Sagar', 'Soni');*

*1 row created.*

*SQL> insert into student*

*2 values(2,'Yash','Shah');*

*1 row created.*

*SQL> insert into student*

*2 values(3,'Priya','Sethi');*

*1 row created.*

*SQL> select \* from student;*

*ROLL\_NO F\_NAME L\_NAME*

*---------- ---------- ----------*

*1 Sagar Soni*

*2 Yash Shah*

*3 Priya Sethi*

*SQL> commit;*

*Commit complete*.

To select this SQL table through java code :

*import java.sql.\*;*

*class OracleJava*

*{*

*public static void main(String args[])*

*{*

*try*

*{*

*//step1 load the driver class*

*Class.forName("oracle.jdbc.driver.OracleDriver");*

*//step2 create the connection object*

*Connection con=DriverManager.getConnection(*

*"jdbc:oracle:thin:@localhost:1521:ace","scott","tiger");*

*//step3 create the statement object*

*Statement stmt=con.createStatement();*

*//step4 execute query*

*ResultSet rs=stmt.executeQuery("select \* from student");*

*while(rs.next())*

*System.out.println(rs.getInt(1)+" "+rs.getString(2)+"*

*"+rs.getString(3));*

*//step5 close the connection object*

*con.close();*

*}*

*catch(Exception e)*

*{*

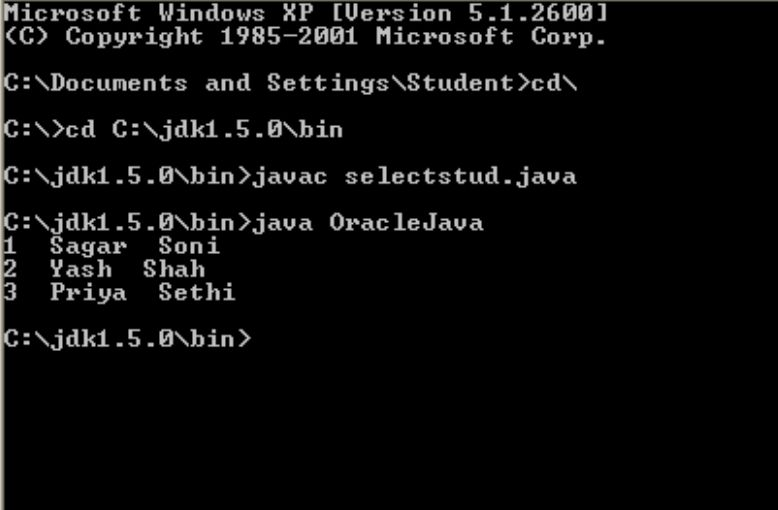
*System.out.println(e);*

*}*

*}*

*}*

**OUTPUT:**



**PRACTICAL NO. 6**

**AIM: - To implement database hashing technique.**

**THEORY:**

For a huge database structure it is not sometime feasible to search index through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate direct location of data record on the disk without using index structure.

It uses a function, called hash function and generates address when called with search key as

parameters. Hash function computes the location of desired data on the disk.

**Hash** **Organization**

● **Bucket**:​ Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket

typically stores one complete disk block, which in turn can store one or more records.

● **Hash Function:**​ A hash function h, is a mapping function that maps all set of search-keys K to

the address where actual records are placed. It is a function from search keys to bucket

addresses.

**Static Hashing**

In static hashing, when a search-key value is provided the hash function always computes the same address. For example, if mod-4 hash function is used then it shall generate only 5 values. The output address shall always be same for that function. The numbers of buckets provided remain same at all times.

**Dynamic Hashing**

Problem with static hashing is that it does not expand or shrink dynamically as the size of database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.

Hash function, in dynamic hashing, is made to produce large number of values and only a few are used initially.

**Program for hashing:**

i*mport java.util.Arrays;*

*// If we think of a Hash Table as an array then a hash function is used to generate*

*// a unique key for every item in the array.*

*// The position the item goes in is known as the slot. Hashing doesn't work very well*

*// in situations in which duplicate data is stored. Also it isn't good for searching*

*// for anything except a specific key.*

*// However a Hash Table is a data structure that offers fast insertion and searching capabilities.*

*public class HashFunction {*

*String[] theArray;*

*int arraySize;*

*int itemsInArray = 0;*

*public static void main(String[] args) {*

*HashFunction theFunc = new HashFunction(30);*

*// Simplest Hash Function*

*// String[] elementsToAdd = { "1", "5", "17", "21", "26" };*

*// theFunc.hashFunction1(elementsToAdd, theFunc.theArray);*

*// Mod Hash Function*

*// This contains exactly 30 items to show how collisions will work*

*String[] elementsToAdd2 = { "100", "510", "170", "214", "268", "398",*

*"235", "802", "900", "723", "699", "1", "16", "999", "890",*

*"725", "998", "978", "988", "990", "989", "984", "320", "321",*

*"400", "415", "450", "50", "660", "624" };*

*theFunc.hashFunction2(elementsToAdd2, theFunc.theArray);*

*// Locate the value 660 in the Hash Table*

*theFunc.findKey("660");*

*theFunc.displayTheStack();*

*} // Simple Hash Function that puts values in the same*

*// index that matches their value*

*public void hashFunction1(String[] stringsForArray, String[] theArray) {*

*for (int n = 0; n < stringsForArray.length; n++) {*

*String newElementVal = stringsForArray[n];*

*theArray[Integer.parseInt(newElementVal)] = newElementVal;*

*}*

*}*

*// Now let's say we have to hold values between 0 & 999but we never plan to have more // than 15 values in all. It wouldn't make sense to make a 1000 item array, so*

*// what can we do? One way to fit these numbers into a 30 item array is*

*// to use the mod function. All you do is take the modulus of the value versus the*

*// array size The goal is to make the array big enough to avoid collisions, but not // so big that we waste memory*

*public void hashFunction2(String[] stringsForArray, String[] theArray) {*

*for (int n = 0; n < stringsForArray.length; n++) {*

*String newElementVal = stringsForArray[n];*

*// Create an index to store the value in by taking*

*// the modulus*

*int arrayIndex = Integer.parseInt(newElementVal) % 29;*

*System.out.println("Modulus Index= " + arrayIndex + " for value "*

*+ newElementVal);*

*// Cycle through the array until we find an empty space*

*while (theArray[arrayIndex] != "-1") {*

*++arrayIndex;*

*System.out.println("Collision Try " + arrayIndex + " Instead");*

*// If we get to the end of the array go back to index 0*

*arrayIndex %= arraySize;*

*}*

*theArray[arrayIndex] = newElementVal;*

*}*

*}*

*// Returns the value stored in the Hash Table*

*public String findKey(String key) {*

*// Find the keys original hash key*

*int arrayIndexHash = Integer.parseInt(key) % 29;*

*while (theArray[arrayIndexHash] != "-1") {*

*if (theArray[arrayIndexHash] == key) {*

*// Found the key so return it*

*System.out.println(key + " was found in index "*

*+ arrayIndexHash);*

*return theArray[arrayIndexHash];*

*}*

*// Look in the next index*

*++arrayIndexHash;*

*// If we get to the end of the array go back to index 0*

*arrayIndexHash %= arraySize;*

*}*

*// Couldn't locate the key*

*return null;*

*}*

*HashFunction(int size) {*

*arraySize = size;*

*theArray = new String[size];*

*Arrays.fill(theArray, "-1");*

*} public void displayTheStack() {*

*int increment = 0;*

*for (int m = 0; m < 3; m++) {*

*increment += 10;*

*for (int n = 0; n < 71; n++)*

*System.out.print("-");*

*System.out.println();*

*for (int n = increment - 10; n < increment; n++) {*

*System.out.format("| %3s " + " ", n);*

*}*

*System.out.println("|");*

*for (int n = 0; n < 71; n++)*

*System.out.print("-");*

*System.out.println();*

*for (int n = increment - 10; n < increment; n++) {*

*if (theArray[n].equals("-1"))*

*System.out.print("| ");*

*else*

*System.out.print(String.format ("| %3s " + " ", theArray[n]));*

*}*

*System.out.println("|");*

*for (int n = 0; n < 71; n++)*

*System.out.print("-");*

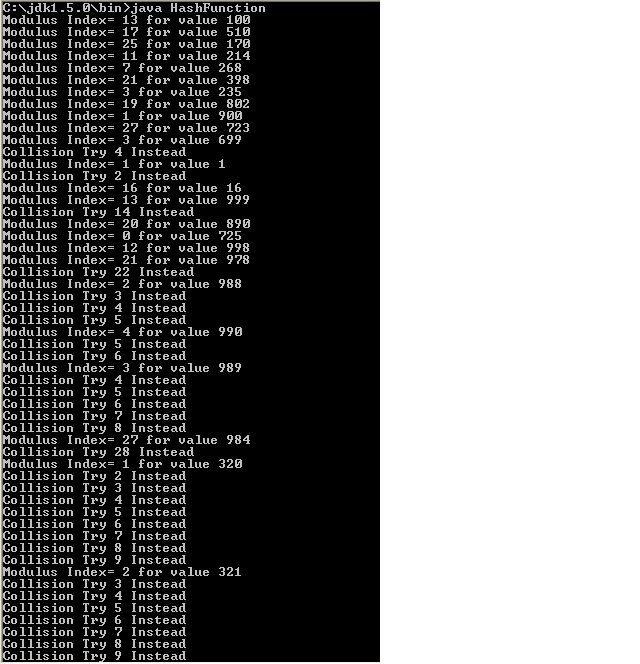
*System.out.println();*

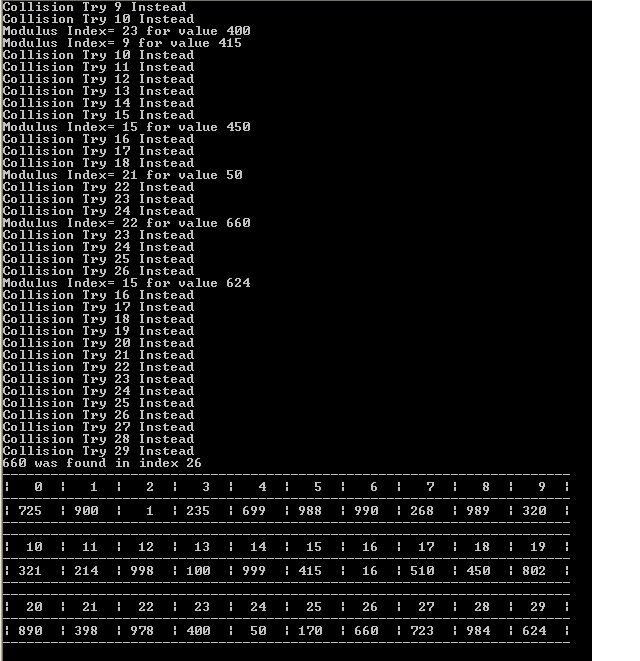
*}*

*}*

*}*

**OUTPUT:**





**PRACTICAL NO. 7**

**AIM: - To Create and querying an Object database. – Using ODL and OQL.**

**THEORY:**

An object-oriented database must provide support for all data types not just the built in data types such as character, integer, and float. To understand abstract data types lets take two steps back by taking off the abstract and then the data from abstract data type. We now have a type, a type would be defined as a collection of a type values. A simple example of this is the Integer type, it consists of values 0, 1, 2, 3, etc. If we add the word data back in we would define data type as a type and the set of operations that will manipulate the type. If we expand off our integer example, a data type would be an integer variable, an integer variable is a member of the integer data type. Addition, subtraction, and multiplication are examples of operations that can be performed on the integer data type.

If we now add the word abstract back in we can define an abstract data type (ADT) as a data type, that is a type and the set of operations that will manipulate the type. The set of operations are only defined by their inputs and outputs. The ADT does not specify how the data type will be implemented, all of the ADT’s details are hidden from the user of the ADT. This process of hiding the details is called encapsulation. If we extend the example for the integer data type to an abstract data type, the operations might be delete an integer, add an integer, print an integer, and check to see if a certain integer exists. Notice that we do not care how the operation will be done but simply how do invoke the operation.

The ability to create new data types when needed and then use these data types is called data

abstraction, and the new data types are called abstract data types (ADTs).

Examples:

To define a new datatype to store a person's address

*CREATE OR REPLACE TYPE persons\_address AS OBJECT (*

*streetNumber NUMBER,*

*streetName VARCHAR2(30),*

*citySuburb VARCHAR2(30),*

*state VARCHAR2(4),*

*postCode NUMBER*

*);*

2. Define a employee\_type; define a "raise\_sal" member function; and create a table based on our new type:

C*REATE TYPE employee\_t AS OBJECT (*

*name VARCHAR2(30),*

*ssn VARCHAR2(11),*

*salary NUMBER,*

*MEMBER FUNCTION raise\_sal RETURN NUMBER)*

*/*

*CREATE TYPE BODY employee\_t AS*

*MEMBER FUNCTION raise\_sal RETURN NUMBER IS*

*BEGIN*

*RETURN salary \* 1.1;*

*END;*

*END;*

*/*

*-- Test the memer function*

*SELECT employee\_t('Frank', '12345', 1000).raise\_sal() from dual;*

*-- Create table based on employee\_t*

*CREATE TABLE emp2 OF employee\_t;*

*INSERT INTO emp2 VALUES ( employee\_t('Frank', '12345', 1000) );*

*SELECT x.raise*\_sal() FROM emp2 x;

Oracle Code and Output

*SQL> CREATE OR REPLACE TYPE ADDRTYPE AS OBJECT*

*( PINCODE NUMBER,*

*STREET VARCHAR2(10),*

*CITY VARCHAR2(10),*

*STATE VARCHAR2(10));*

*/*

*Type created.*

*SQL> CREATE OR REPLACE TYPE AUTHORTYPE AS OBJECT*

*(NAME VARCHAR2(10),*

*ADDRESS ADDRTYPE);*

*/*

*Type created.*

*SQL> CREATE TABLE AUTHORS OF AUTHORTYPE;*

*Table created.*

*SQL> INSERT INTO AUTHORS VALUES*

*('NAVATHE',ADDRTYPE(4321,'MANPADA','PANVEL','MH'));*

*1 row created*

*SQL> INSERT INTO AUTHORS VALUES*

*('SANGHOI',ADDRTYPE(3245,'LBSMARG','DADAR','TN'));*

*1 row created.*

*SQL> SELECT \* FROM AUTHORS;*

*NAME*

*----------*

*ADDRESS(PINCODE, STREET, CITY, STATE)*

*--------------------------------------------------------------------------*

*SUJATA*

*ADDRTYPE(400020, 'CST', 'MUMBAI', 'MAH')*

*NAVATHE*

*ADDRTYPE(4321, 'MANPADA', 'PANVEL', 'MH')*

*SANGHOI*

*ADDRTYPE(3245, 'LBSMARG', 'DADAR', 'TN')*

*SQL> DESC AUTHORS;*

*Name*

*Null? Type*

*----------------------------------------- -------- ----------------------*

*NAME*

*VARCHAR2(10)*

*ADDRESS*

*ADDRTYPE*

*SQL> DESC ADDRTYPE;*

*Name*

*Null? Type*

*----------------------------------------- -------- ----------------------*

*PINCODE*

*NUMBER*

*STREET*

*VARCHAR2(10)*

*CITY*

*VARCHAR2(10)*

*STATE*

*VARCHAR2(10)*

*SQL> set desc depth 2;*

*SQL> desc authors;*

*Name*

*Null? Type*

*----------------------------------------- -------- ----------------------------*

*NAME*

*VARCHAR2(10)*

*ADDRESS*

*ADDRTYPE*

*PINCODE*

*NUMBER*

*STREET*

*VARCHAR2(10)*

*CITY*

*VARCHAR2(10)*

*STATE*

*VARCHAR2*(10)

Retrive city & state from author table?

*SQL> select z.address.city from authors z ;*

*ADDRESS.CI*

*----------*

*MUMBAI*

*PANVEL*

*DADAR*

*SQL> select z.address.city,z.address.state from authors z;*

*ADDRESS.CI ADDRESS.ST*

*---------- ----------*

*MUMBAI MAH*

*PANVEL MH*

*DADAR*

*TN*

Q)Change the city value for authors who lives in Mumbai to Delhi?

*SQL> update authors a*

*2 set a.address.city='delhi'*

*3 where name='sujata';*

*0 rows updated. SQL> update authors a*

*2 set a.address.city='delhi'*

*3 where a.address.state='MAH';*

*1 row updated*.

Q)Delete the record for the authors who lives in Mumbai.

*SQL> delete from authors a*

*2 where a.address.city='DADAR';*

*1 row deleted.*

*SQL> select \* from authors;*

*NAME*

*----------*

*ADDRESS(PINCODE, STREET, CITY, STATE)*

*----------------------------------------------------------*

*SUJATA*

*ADDRTYPE(400020, 'CST', 'delhi', 'MAH')*

*NAVATHE*

*ADDRTYPE(4321, 'MANPADA', 'PANVEL'*, 'MH')

Q) Drop type Address\_ty.

*SQL> drop type addrtype;*

*drop type addrtype*

*\**

*ERROR at line 1:*

*ORA-02303: cannot drop or replace a type with type or table dependents*

*SQL> drop type addrtype FORCE;*

*Type dropped.*

**PRACTICAL NO. 8**

**AIM: - To implement fragmentation in Distributed Database.**

**THEORY:**

Centralised databases are more powerful and useful when sites are not geographically very long. Cost of transferring data becomes a problem when distance increases, also the problem of failure of links, failure of central server are the ones which hampers the on-line work of the organization. This gives the idea of distribution and careful replication of data, to overcome the problems of link failures and the transportaion of data for each and every query, retrival or updation of database.

Oracle 8i supports data fragmentation by portioning the tables horizontaly on different tablespaces.

Partionoing is transparent to the user and need not worry about which partion to be used while

inserting/delting/quering table. Use of partion name is also allowed but not necessary. Of course if you put invalid data for that partion it will not allow; but if you don’t mention partion name it will transperntly go to the proper partion.

Syntax for creating Fragments(Partitions)

*create table worker(*

*Name varchar2(25),*

*age number,*

*constraint worker\_pk primary key(Name)*

*)*

*partition by range (ag)*

*(partition part1 values less than (18),*

*partition part2 values less than (40),*

*partition part3 values less than (60),*

*partition part1 values less than (MAXVALUE));*

**Example:**

*SQL> create table stud (*

*roll\_no integer primary key,*

*name varchar2(10),*

*percentage integer)*

*partition by range(percentage)*

*(partition fail values less than(40),*

*partition pass values less than (50),*

*partition second values less than (60),*

*partition first values less than (70));*

*Table created.*

*SQL> insert into stud values (2,'naitik',69);*

*1 row created.*

*SQL> insert into stud values (3,'sneha',65);*

*1 row created.*

*SQL> insert into stud values (4,'pratiksha',66);*

*1 row created.*

*SQL> insert into stud values (5,'punit',40);*

*1 row created*.

*SQL> insert into stud values (6,'sam',35);*

*1 row created.*

*SQL> insert into stud values (7,'kunal',62);*

*1 row created.*

*SQL> insert into stud values (8,'sonam',56);*

*1 row created.*

*SQL> select \* from stud;*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*6 sam 35*

*5 punit 40*

*8 sonam 56*

*2 naitik 69*

*3 sneha 65*

*4 pratiksha 66*

*7 kunal 62*

*7 rows selected.*

*SQL> select \* from stud partition(first);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*2 naitik 69*

*3 sneha 65*

*4 pratiksha 66*

*7 kunal 62*

*SQL> select \* from stud partition(fail);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*6 sam 35*

*SQL> select \* from stud partition(second);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*8 sonam 56*

*SQL> select \* from stud partition(pass);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*5 punit 40*

*SQL> insert into stud values (1,'sagar',70);*

*insert into stud values (1,'sagar',70)*

*\**

*ERROR at line 1:*

*ORA-14400: inserted partition key does not map to any partition*

*SQL> alter table stud add partition distinction values less than(100);*

*Table altered.*

*SQL> insert into stud values (1,'sagar',70);*

*1 row created.*

*SQL> select \* from stud partition(distinction);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*1 sagar 70*

*SQL> alter table stud rename partition distinction to superdistinction;*

*Table altered.*

*SQL> select \* from stud partition(superdistinction);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*1 sagar 70*

*SQL> select \* from stud partition(distinction);*

*select \* from stud partition(distinction)*

*\**

*ERROR at line 1:*

*ORA-02149: Specified partition does not exist*

*SQL> alter table stud merge partitions first,superdistinction into partition*

*distinction;*

*Table altered.*

*SQL> select \* from stud partition(distinction);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*2 naitik 69*

*3 sneha 65*

*4 pratiksha 66*

*7 kunal 62*

*1 sagar 70*

*SQL> select \* from stud partition(first);*

*select \* from stud partition(first)*

*\**

*ERROR at line 1:*

*ORA-02149: Specified partition does not exist*

*SQL> select \* from stud partition(superdistinction);*

*select \* from stud partition(superdistinction)*

*\**

*ERROR at line 1:*

*ORA-02149: Specified partition does not exist*

*SQL> Alter table stud split partition distinction at(65) into(partition*

*lower,partition upper);*

*Table altered.*

*SQL> select \* from stud partition(upper);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*2 naitik 69*

*3 sneha 65*

*4 pratiksha 66*

*1 sagar 70*

*SQL> select \* from stud partition(lower);*

*ROLL\_NO NAME PERCENTAGE*

*---------- ---------- ----------*

*7 kunal 62*

**PRACTICAL NO. 9**

**AIM: - To study database security techniques – SQL injection.**

**THEORY:**

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via web page input. Injected SQL commands can alter SQL statement and compromise the security of a web application.

Web applications allow legitimate website visitors to submit and retrieve data to/from a database over the Internet using their preferred web browser. Databases are central to modern websites – they store data needed for websites to deliver specific content to visitors and render information to customers, suppliers, employees and a host of stakeholders. User credentials, financial and payment information, company statistics may all be resident within a database and accessed by legitimate users through off-the-shelf and custom web applications. Web applications and databases allow you to regularly run your business.

SQL Injection is the hacking technique which attempts to pass SQL commands (statements) through a web application for execution by the backend database. If not sanitized properly, web applications may result in SQL Injection attacks that allow hackers to view information from the database and/or even wipe it out.

**SQL Injection Based on 1=1 is Always True**

Look at the example above, one more time.

Let's say that the original purpose of the code was to create an SQL statement to select a user with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId: 105 or 1=1

Server Result

*SELECT \* FROM Users WHERE UserId = 105 or 1=1*

The SQL above is valid. It will return all rows from the table Users, since **WHERE 1=1** is always

true.

Does the example above seem dangerous? What if the Users table contains names and passwords?

The SQL statement above is much the same as this:

*SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1*

A smart hacker might get access to all the user names and passwords in a database by simply inserting 105 or 1=1 into the input box.

**SQL Injection Based on ""="" is Always True**

Here is a common construction, used to verify user login to a web site:

User Name: \_\_\_\_\_\_\_\_\_

Password: \_\_\_\_\_\_\_\_\_\_\_\_

Server Code

*uName = getRequestString("UserName");*

*uPass = getRequestString("UserPass");*

*sql = "SELECT \* FROM Users WHERE Name ='" + uName + "' AND Pass ='" + uPass + "'"*

A smart hacker might get access to user names and passwords in a database by simply inserting "or" "=" into the user name or password text box.

The code at the server will create a valid SQL statement like this:

Result

*SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""*

The result SQL is valid. It will return all rows from the table Users, since WHERE ""="" is always

true.

**Parameters for Protection**

Some web developers use a "blacklist" of words or characters to search for in SQL input, to prevent SQL injection attacks.

This is not a very good idea. Many of these words (like delete or drop) and characters (like semicolons and quotation marks), are used in common language, and should be allowed in many types of input.

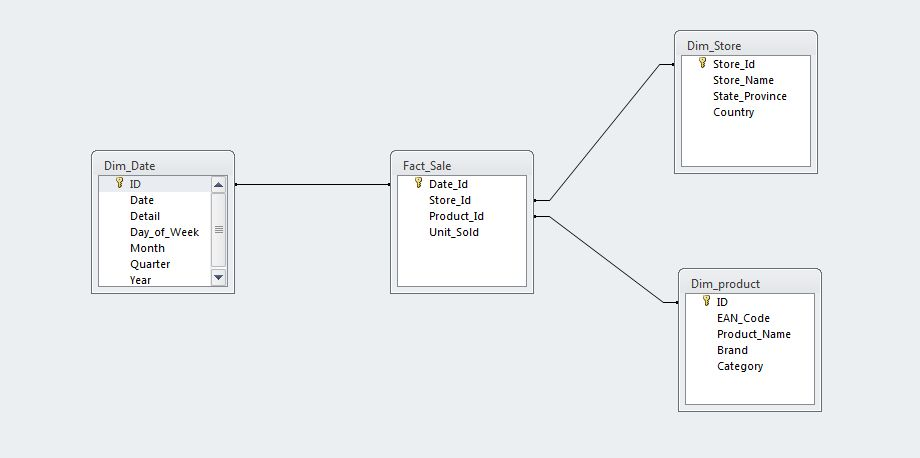
(In fact it should be perfectly legal to input an SQL statement in a database field.)

The only proven way to protect a web site from SQL injection attacks, is to use SQL parameters.

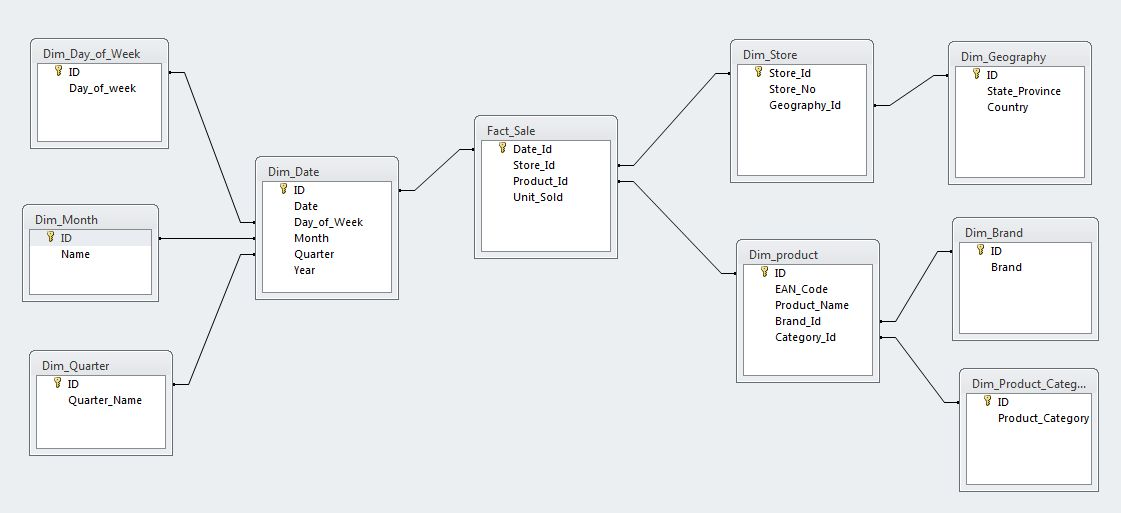
SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

**PRACTICAL NO.10**

**AIM: -To Design a Data Warehouse using Star Schema a Snowflake Schema.**



**STAR SCHEMA**



**SNOWFLAKE SCHEMA**

//Fact\_Sales

*create table fact\_sales(id varchar(25),store\_idvarchar(25),product\_idvarchar(25),unit\_idvarchar(25))*

*Table created.*

*insert into fact\_sales*

*values('D10','S10','P10',100),*

*values('D20','S20','P20',150),*

*values('D30','S30','P30',300)*

*3 row(s) inserted.*

*select \* from fact\_sales*

*ID STORE\_ID PRODUCT\_ID UNIT\_ID*

*D10 S10 P10 100*

*D20 S20 P20 150*

*D30 S30 P30 300*

//Dim\_Store

*create table dim\_store(id varchar(25),store\_novarchar(25),state\_provincevvarchar(25),country varchar(25))*

*Table created.*

*insert into dim\_store*

*values('D10',1001,'Mumbai','INDIA'),*

*values('D20',1002,'Gujrat','INDIA'),*

*values('D30',1003,'GOA','INDIA')*

*3 row(s) inserted.*

*Select \* form dim\_store*

*ID STORE\_NO STATE\_PROVINCEV COUNTRY*

*D10 1001 Mumbai INDIA*

*D20 1002 Gujrat INDIA*

*D20 1002 GOA INDIA*

*D30 1003 GOA INDIA*

//Dim\_Product

*create table dim\_product(id varchar(25),Ean\_codevarchar(25),product\_namevarchar(25),Brand*

*varchar(25), product\_categoryvarchar(25))*

*Table created.*

*insert into dim\_product*

*values('P10','E100','c7','Nokia','Mobile'),*

*values('P20','E200','Lumia510','Nokia','Mobile'),*

*values('P30','E300','LENOVO','IBM','Computer')*

*3 row(s) inserted.*

*select \* from dim\_product*

*ID EAN\_CODE PRODUCT\_NAME BRAND PRODUCT\_CATEGORY*

*P10 E100 c7 Nokia Mobile*

*P20 E200 Lumia510 Nokia Mobile*

*P30 E300 LENOVO IBM Computer*

//Dim\_Date

*CREATE TABLE Dim\_Date(Idvarchar(25), Dates varchar(25), Days varchar(25), Months*

*varchar(25), Quatersvarchar(25), Years varchar(25))*

*insert into Dim\_Date values('D10',26,'Wed',9,'3mn',2007)*

*1 row(s) inserted.*

*insert into Dim\_Date values('D20',21,'Mon',4,'3mn',2008)*

*1 row(s) inserted.*

*insert into Dim\_Date values('D30',20,'Sat',9,'4mn',2008)*

*1 row(s) inserted*

*select \* from Dim\_date*

*ID DATES DAYS MONTHS QUATERS YEARS*

*D10 26 Wed 9 3mn 2007*

*D20 21 Mon 4 3mn 2008*

*D30 20 Sat 9 4mn 2008*

**PRACTICAL NO.11**

**AIM: - To implement queries for OLAP operations. ( Roll up, Drill down, Slice, Dice, pivot).**

**THEORY:**

**OLAP Operations**

As we know that the OLAP server is based on the multidimensional view of data hence we will

discuss the OLAP operations in multidimensional data.

Here is the list of OLAP operations.

● Roll-up

● Drill-down

● Slice and dice

● Pivot (rotate)

**Roll-up**

This operation performs aggregation on a data cube in any of the following way:

● By climbing up a concept hierarchy for a dimension

● By dimension reduction.

● The roll-up operation is performed by climbing up a concept hierarchy for the dimension

location.

● Initially the concept hierarchy was "street < city < province < country".

● On rolling up the data is aggregated by ascending the location hierarchy from the level of city

to level of country.

● The data is grouped into cities rather than countries.

● When roll-up operation is performed then one or more dimensions from the data cube are

removed.

**Drill-down**

Drill-down operation is reverse of the roll-up. This operation is performed by either of the following way:

● By stepping down a concept hierarchy for a dimension.

● By introducing new dimension.

● The drill-down operation is performed by stepping down a concept hierarchy for the dimension time.

● Initially the concept hierarchy was "day < month < quarter < year."

● On drill-up the time dimension is descended from the level quarter to the level of month.

● When drill-down operation is performed then one or more dimensions from the data cube are added.

● It navigates the data from less detailed data to highly detailed data.

**Slice**

The slice operation performs selection of one dimension on a given cube and give us a new sub cube.

The Slice operation is performed for the dimension time using the criterion time ="Q1".

● It will form a new sub cube by selecting one or more dimensions.

**Dice**

The Dice operation performs selection of two or more dimension on a given cube and give us a new subcube

The dice operation on the cube based on the following selection criteria that involve three dimensions.

● (location = "Toronto" or "Vancouver")

● (time = "Q1" or "Q2")

● (item =" Mobile" or "Modem").

**Pivot**

The pivot operation is also known as rotation.It rotates the data axes in view in order to provide an alternative presentation of data

*SQL> create table data*

*2 (num number(2),*

*3 gender varchar2(10),*

*4 marrried varchar2(10),*

*5 city varchar2(20));*

*Table created.*

*SQL> alter table data add primary key(num);*

*Table altered.*

*SQL> desc data;*

*Name Null? Type*

*----------------------------------------- -------- ----------------------------*

*NUM NOT NULL NUMBER(2)*

*GENDER VARCHAR2(10)*

*MARRRIED VARCHAR2(10)*

*CITY VARCHAR2(20)*

*SQL> insert into data*

*values(&num,'&gender','&married','&city')*

*Enter value for num: 1*

*Enter value for gender: male*

*Enter value for married: single*

*Enter value for city: mumbai*

*old 1: insert into data*

*values(&num,'&gender','&married','&city')*

*new 1: insert into data*

*values(1,'male','single','mumbai')*

*1 row created.*

*SQL> select \* from data;*

*NUM GENDER MARRRIED CITY*

*---------- ---------- ---------- --------------------*

*1 male single mumbai*

*2 male divorced delhi*

*3 female single mumbai*

*4 male married mumbai*

*5 female married mumbai*

*6 female married madras*

*7 male single delhi*

*8 female divorced madras*

*8 rows selected.*

*SQL> select gender, count(\*) from data group by gender*

*GENDER COUNT(\*)*

*---------- ----------*

*female 4*

*male 4*

*SQL> select marrried, count(\*) from data group by marrried*

*MARRRIED COUNT(\*)*

*---------- ----------*

*divorced 2*

*married 3*

*single 3*

*SQL> select marrried,gender,city, count(\*) from data group by*

*cube(marrried,gender,city)*

*MARRRIED GENDER CITY COUNT(\*)*

*---------- ---------- -------------------- ----------*

*divorced female madras 1*

*divorced female 1*

*divorced male delhi 1*

*divorced male 1*

*divorced delhi 1*

*divorced madras 1*

*divorced 2*

*married female madras 1*

*married female mumbai 1*

*married female 2*

*married male mumbai 1*

*MARRRIED GENDER CITY COUNT(\*)*

*---------- ---------- -------------------- ----------*

*married male 1*

*married madras 1*

*married mumbai 2*

*married 3*

*single female mumbai 1*

*single female 1*

*single male delhi 1*

*single male mumbai 1*

*single male 2*

*single delhi 1*

*single mumbai 2*

*MARRRIED GENDER CITY COUNT(\*)*

*---------- ---------- -------------------- ----------*

*single 3*

*female madras 2*

*female mumbai 2*

*female 4*

*male delhi 2*

*male mumbai 2*

*male 4*

*delhi 2*

*madras 2*

*mumbai 4*

*8 33 rows selected.*

*SQL> select marrried, count(\*) from data group by cube(marrried)*

*MARRRIED COUNT(\*)*

*---------- ----------*

*divorced 2*

*married 3*

*single 3*

*8*

*SQL> select marrried,gender, count(\*) from data group by*

*cube(marrried,gender)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*divorced male 1*

*divorced 2*

*married female 2*

*married male 1*

*married 3*

*single female 1*

*single male 2*

*single 3*

*female 4*

*male 4*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*8*

*12 rows selected.*

*SQL> select marrried,gender, count(\*) from data group by*

*rollup(marrried,gender)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*divorced male 1*

*divorced 2*

*married female 2*

*married male 1*

*married 3*

*single female 1*

*single male 2*

*single 3*

*8*

*10 rows selected.*

*SQL> select marrried,gender, count(\*) from data group by*

*rollup(gender,marrried)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*married female 2*

*single female 1*

*female 4*

*divorced male 1*

*married male 1*

*single male 2*

*male 4*

*8*

*9 rows selected.*

*SQL> select marrried,gender, count(\*) from data group by*

*rollup(gender,marrried) having gender = 'ma*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced male 1*

*married male 1*

*single male 2*

*male 4*

*SQL> select marrried,gender, count(\*) from data group by cube(gender,marrried)*

*having gender = 'male'*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced male 1*

*married male 1*

*single male 2*

*male 4*

*SQL> select marrried,gender, count(\*) from data group by cube(gender,marrried)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*married female 2*

*single female 1*

*female 4*

*divorced male 1*

*married male 1*

*single male 2*

*male 4*

*divorced 2*

*married 3*

*single 3*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*8*

*12 rows selected*.

*SQL> select marrried,gender, count(\*) from data group by cube(marrried,gender)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*divorced male 1*

*divorced 2*

*married female 2*

*married male 1*

*married 3*

*single female 1*

*single male 2*

*single 3*

*female 4*

*male 4*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*8*

*12 rows selected*.

*SQL> select marrried,gender, count(\*) from data group by*

*rollup(marrried,gender)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*divorced male 1*

*divorced 2*

*married female 2*

*married male 1*

*married 3*

*single female 1*

*single male 2*

*single 3*

*8*

*10 rows selected*.

*SQL> select marrried,gender, count(\*) from data group by*

*rollup(gender,marrried)*

*MARRRIED GENDER COUNT(\*)*

*---------- ---------- ----------*

*divorced female 1*

*married female 2*

*single female 1*

*female 4*

*divorced male 1married male 1*

*single male 2*

*male 4*

*8*

*9 rows selected*.

*SQL> select city, count(\*) from data group by rollup(city,gender,marrried) having*

*marrried = 'marrried'*

*CITY COUNT(\*)*

*-------------------- ----------*

*mumbai* 1

*SQL>select city, count(\*) from data group by rollup(city,gender,marrried) having*

*marrried = 'single'*

*CITY COUNT(\*)*

*-------------------- ----------*

*delhi 1*

*mumbai* 1

*SQL> select city, count(\*) from data group by (marrried,gender,city) having*

*marrried = 'single' and gender = 'male'*

*CITY COUNT(\*)*

*-------------------- ----------*

*delhi 1*

*mumbai 1*