DATA DEFINITION COMMANDS, DATA MANIPULATION COMMANDS FOR INSERTING, DELETING, UPDATING AND RETRIEVING TABLES AND TRANSACTION CONTROL STATEMENTS

Aim:

To create table and Execute Data Definition Commands, Data Manipulation Commands for Inserting, Deleting, Updating And Retrieving Tables and Transaction Control Statements

SQL: create command

Create is a DDL SQL command used to create a table or a database in relational database management system.

Creating a Database

To create a database in RDBMS, **create** command is used. Following is the syntax,

CREATE DATABASE <DB_NAME>;

Example for creating Database

CREATE DATABASE Test;

The above command will create a database named **Test**, which will be an empty schema without any table.

To create tables in this newly created database, we can again use the create command.

Creating a Table

Create command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the **names** and **data types** of various columns in the create command itself.

Following is the syntax,

```
CREATE TABLE <TABLE_NAME>
(
   column_name1 datatype1,
   column_name2 datatype2,
   column_name3 datatype3,
   column_name4 datatype4
);
```

create table command will tell the database system to create a new table with the given table name and column information.

Most commonly used data types for Table columns

Here we have listed some of the most commonly used data types used for columns in tables.

Datatype	Use
INT	used for columns which will store integer values.
FLOAT	used for columns which will store float values.
DOUBLE	used for columns which will store float values.
VARCHAR	used for columns which will be used to store characters and integers, basically a string.
CHAR	used for columns which will store char values(single character).
DATE	used for columns which will store date values.
TEXT	used for columns which will store text which is generally long in length. For example, if you create a table for storing profile information of a social networking website, then for **about me** section you can have a column of type TEXT.

SQL: ALTER command

alter command is used for altering the table structure, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.
- to drop a column from the table.

ALTER Command: Add a new Column

Using ALTER command we can add a column to any existing table. Following is the syntax,

ALTER TABLE table_name ADD(column_name datatype);

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table. Following is the syntax,

ALTER TABLE table_name ADD(column_name1 datatype1, column-name2 datatype2,);

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column. Following is the syntax,

ALTER TABLE table_name ADD(column-name1 datatype1 DEFAULT some_value);

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column. Following is the syntax,

ALTER TABLE table_name modify(column_name datatype);

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column. Following is the syntax,

ALTER TABLE table_name RENAME old_column_name TO new_column_name;

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns. Following is the syntax,

ALTER TABLE table_name DROP(column_name);

TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized. Following is its syntax,

TRUNCATE TABLE table_name

DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it. Following is its syntax,

DROP TABLE table_name

RENAME query

RENAME command is used to set a new name for any existing table. Following is the syntax,

RENAME TABLE old_table_name to new_table_name

DML COMMAND

Using INSERT SQL command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

INSERT INTO table_name VALUES(data1, data2, ...)

Insert value into only specific columns

We can use the INSERT command to insert values for only some specific columns of a row. We can specify the column names along with the values to be inserted like this,

INSERT INTO student(id, name) values(value, value);

The above SQL query will only insert id and name values in the newly inserted record.

Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the **student** table.

INSERT INTO student(id, name) values(value, value);

Or,

INSERT INTO Student VALUES(102,'Alex', null);

The above command will insert only two column values and the other column is set to null.

S_id	S_Name	age
101	Adam	15
102	Alex	

Insert Default value to a column

INSERT INTO Student VALUES(103,'Chris', default)

S_id	S_Name	age
101	Adam	15
102	Alex	
103	chris	14

Suppose the column age in our tabel has a default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT INTO Student VALUES(103,'Chris')

Using UPDATE SQL command

UPDATE command

UPDATE command is used to update any record of data in a table. Following is its general syntax,

UPDATE table_name SET column_name = new_value WHERE some_condition;
WHERE is used to add a condition to any SQL query, we will soon study about it in detail.

Lets take a sample table **student**,

student_id	name	age
101	Adam	15
102	Alex	
103	chris	14

UPDATE student SET age=18 WHERE student_id=102;

S_id	S_Name	age
101	Adam	15
102	Alex	18
103	chris	14

In the above statement, if we do not use the WHERE clause, then our update query will update age for all the columns of the table to **18**.

Updating Multiple Columns

We can also update values of multiple columns using a single UPDATE statement.

UPDATE student SET name='Abhi', age=17 where s_id=103;

The above command will update two columns of the record which has s_id 103.

s_id	name	age
101	Adam	15
102	Alex	18
103	Abhi	17

UPDATE Command: Incrementing Integer Value

UPDATE student SET age = age+1;

As you can see, we have used age = age + 1 to increment the value of age by 1.

NOTE: This style only works for integer values.

Using DELETE SQL command

DELETE command

DELETE command is used to delete data from a table.

Following is its general syntax,

DELETE FROM table_name;

Let's take a sample table **student**:

s_id	**name**	**age**
101	Adam	15
102	Alex	18
103	Abhi	17

Delete all Records from a Table

DELETE FROM student;

The above command will delete all the records from the table **student**.

Delete a particular Record from a Table

In our **student** table if we want to delete a single record, we can use the WHERE clause to provide a condition in our DELETE statement.

DELETE FROM student WHERE s_id=103;

The above command will delete the record where s_id is 103 from the table **student**.

S_id	S_Name	age
101	Adam	15
102	Alex	18

Isn't DELETE same as TRUNCATE

TRUNCATE command is different from DELETE command. The delete command will delete all the rows from a table whereas truncate command not only deletes all the records stored in the table, but it also re-initializes the table(like a newly created table).

COMMIT, ROLLBACK AND SAVEPOINT SQL COMMANDS

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

COMMIT command

COMMIT command is used to permanently save any transaction into the database.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's syntax,

COMMIT;

ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

Following is rollback command's syntax,

ROLLBACK TO savepoint_name;

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINT savepoint_name;

Using Savepoint and Rollback

Following is the table **class**,

id	name

1	Abhi
2	Adam
4	Alex

Let’s use some SQL queries on the above table and see the results.

```
INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINT A;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINT B;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINT C;

SELECT * FROM class;
```

NOTE: SELECT statement is used to show the data stored in the table.

The resultant table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

Now let's use the ROLLBACK command to roll back the state of data to the **savepoint B**.

```
ROLLBACK TO B;

SELECT * FROM class;
```

Now our **class** table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK command to roll back the state of data to the **savepoint A**

ROLLBACK TO A;

SELECT * FROM class;

Now the table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

So now you know how the commands COMMIT, ROLLBACK and SAVEPOINT works.

Result : Table created successfully , and applied all the DDL, DML, TCL commands.

DATABASE QUERYING – SIMPLE QUERIES, NESTED QUERIES, SUB QUERIES AND JOINS

Aim:

To Create Table and Apply Simple Queries Nested Queries, Sub Queries and Joins.

SQL - SELECT Query

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

The basic syntax of the SELECT statement is as follows −

SELECT column1, column2, columnN FROM table_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

SELECT * FROM table_name;

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 |
+----+----------+-----+-----------+----------+
```

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result –

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

SQL> SELECT * FROM CUSTOMERS;

This would produce the result as shown below.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

SUB QUERY IN ORACLE

Query

While creating a database if we want to extract some information regarding the data in the database then we use a Query. In other words, if we want to retrieve some data from a table or some tables that we created earlier then we write/use a Query.

Example: If we write a simple Query to create a table:

```
1. CREATE TABLE Product
2. (
3. Prod_Id Number Not Null,
4. Prod_Name Varchar2(50),
5. Quantity Varchar2(15),
6. Price Number
7. );
```

Then, the result will be as in the following.

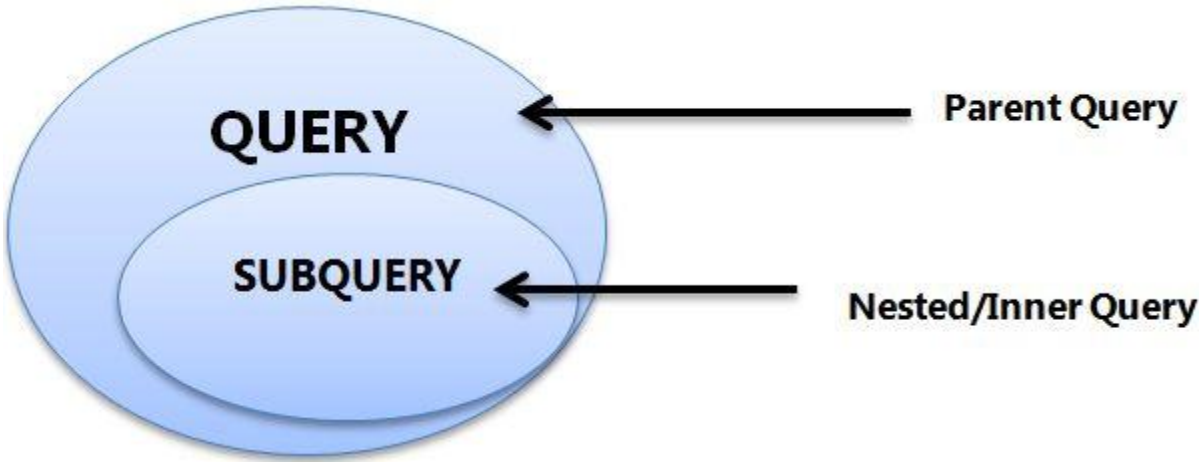
Product Table

Prod_id	Prod_Name	Quantity	Price

Sub Query

If a Query that contains another Query, then the Query inside the main Query is called a *Sub Query*and the main Query is known as the parent Query. In Oracle the Sub Query will executed on the prior basis and the result will be available to the parent Query and then the execution of the parent/main Query takes place. Sub Queries are very useful for selecting rows from a table having a condition that depends on the data of the table itself. A Sub Query can also be called a Nested/Inner Query. These Sub Queries can be used with:

- WHERE Clause
- SELECT Clause
- FROM Clause



Syntax

```
1.  SELECT <column, ...>
2.  FROM <table>
3.  WHERE expression operator
4.    (
5.     SELECT <column, ...>
6.     FROM <table>
7.     WHERE <condition>
8.     );
```

Or

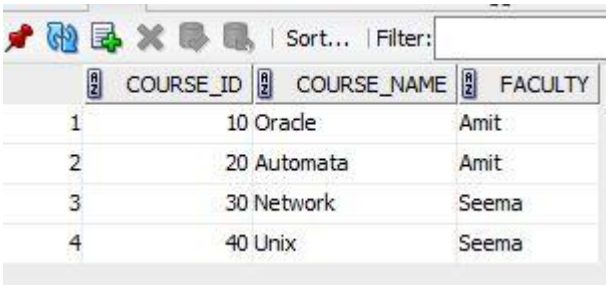
```
SELECT Col_name [, Col_name]
FROM table1 [,table2]
WHERE Col_name OPERATOR
  (
  SELECT Col_name [,Col_name]
  FROM table1 [,table2]
  [WHERE]
  );
```

Now let us explain the Sub Query using all the three clauses. For that we are assuming the following tables.

STUDENT TABLE

	STUD_ID	STUD_NAME	AGE	EMAIL	COURSE_ID
1	1	Anjali	26	anjali@a...	10
2	2	Shweta	27	shweta...	30
3	3	Sapna	25	sapna@...	30
4	4	Doli	26	doli@ab...	10

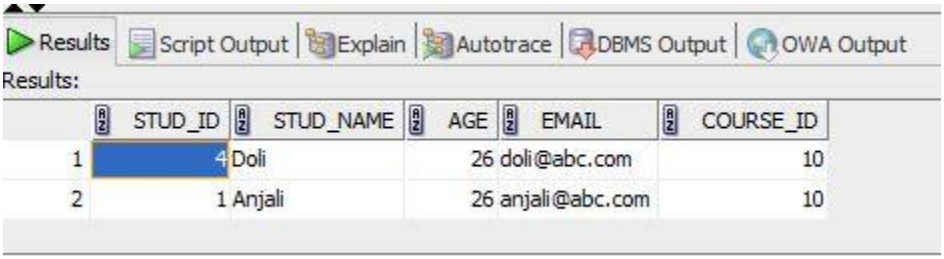
SUBJECT TABLE



	COURSE_ID	COURSE_NAME	FACULTY
1	10	Oracle	Amit
2	20	Automata	Amit
3	30	Network	Seema
4	40	Unix	Seema

1. Sub Query using WHERE Clause

```
SELECT * FROM student
WHERE course_id in (SELECT course_id
FROM subject
WHERE course_name = 'Oracle')
```



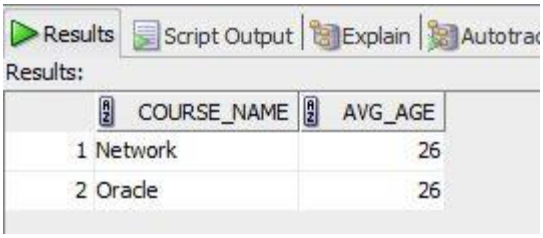
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	STUD_ID	STUD_NAME	AGE	EMAIL	COURSE_ID
1	4	Doli	26	doli@abc.com	10
2	1	Anjali	26	anjali@abc.com	10

2. Sub Query using FROM Clause

```
1. SELECT a.course_name, b.Avg_Age
2. FROM subject a, (SELECT course_id, Avg(Age) as Avg_Age
3. FROM student GROUP BY course_id) b
4. WHERE b.course_id = a.course_id
```

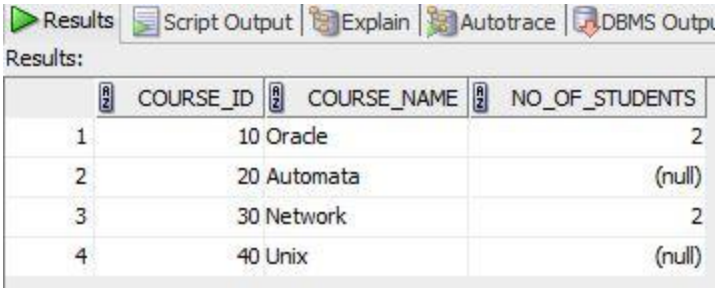

Results | Script Output | Explain | Autotrac

Results:

	COURSE_NAME	AVG_AGE
1	Network	26
2	Oracle	26

3. Sub Query using SELECT Clause

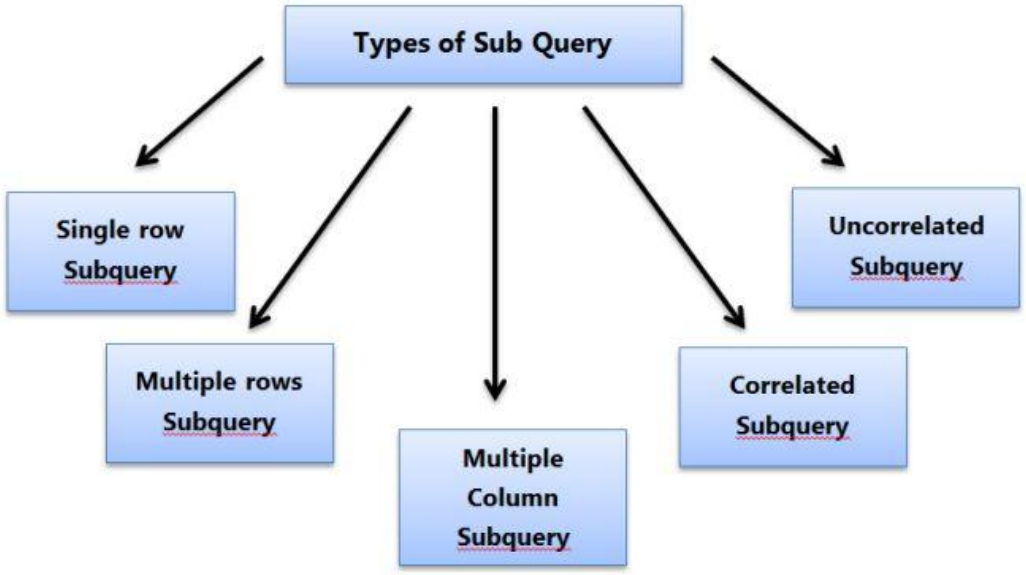
```
1. SELECT course_id, course_name,
2. (
3. SELECT count (course_id) as num_of_student
4. FROM student a
5. WHERE a.course_id = b.course_id
6. GROUP BY course_id
7. )  No_of_Students
8. FROM subject b
```


Results | Script Output | Explain | Autotrace | DBMS Outpu

Results:

	COURSE_ID	COURSE_NAME	NO_OF_STUDENTS
1	10	Oracle	2
2	20	Automata	(null)
3	30	Network	2
4	40	Unix	(null)

Types of Sub Queries



Here, for all the types of Sub Queries we will use the default Scott Schema. And the following are the default tables of the Scott Schema.

EMPLOYEE TABLE with Column Name

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

Actions...

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	1	Primary key o...
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	(null)	First name of ...
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	(null)	Last name of ...
EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	(null)	Email id of the...
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	(null)	Phone numbe...
HIRE_DATE	DATE	No	(null)	6	(null)	Date when th...
JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	(null)	Current job of...
SALARY	NUMBER(8,2)	Yes	(null)	8	(null)	Monthly salar...
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	(null)	Commission p...
MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	(null)	Manager id of...
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	(null)	Department id...

EMPLOYEE TABLE with Data

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

Sort... | Filter:

	EM...	FIRST_NAME	LAS...	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	S...	COMM...	MA...	DE...
1	198	Donald	OConnell	DOCON...	650.507.9833	21-JUN-07	SH_CLERK	2600	(null)	124	50
2	199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-08	SH_CLERK	2600	(null)	124	50
3	200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-03	AD_ASST	4400	(null)	101	10
4	201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-04	MK_MAN	13000	(null)	100	20
5	202	Pat	Fay	PFAY	603.123.6666	17-AUG-05	MK_REP	6000	(null)	201	20
6	203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-02	HR_REP	6500	(null)	101	40
7	204	Hermann	Baer	HBAER	515.123.8888	07-JUN-02	PR_REP	10000	(null)	101	70
8	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-02	AC_MGR	12008	(null)	101	110
9	206	William	Gietz	WGIETZ	515.123.8181	07-JUN-02	AC_ACC...	8300	(null)	205	110
10	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24000	(null)	(null)	90
11	101	Neena	Kochhar	NKOCHH...	515.123.4568	21-SEP-05	AD_VP	17000	(null)	100	90
12	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01	AD_VP	17000	(null)	100	90
13	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06	IT_PROG	9000	(null)	102	60
14	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07	IT_PROG	6000	(null)	103	60
15	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-05	IT_PROG	4800	(null)	103	60
16	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-06	IT_PROG	4800	(null)	103	60
17	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07	IT_PROG	4200	(null)	103	60
18	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-02	FI_MGR	12008	(null)	101	100
19	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-02	FI_ACCO...	9000	(null)	108	100
20	110	John	Chen	JCHEN	515.124.4269	28-SEP-05	FI_ACCO...	8200	(null)	108	100
21	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-05	FI_ACCO...	7700	(null)	108	100
22	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-06	FI_ACCO...	7800	(null)	108	100
23	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-07	FI_ACCO...	6900	(null)	108	100
24	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-02	PU_MAN	11000	(null)	100	30

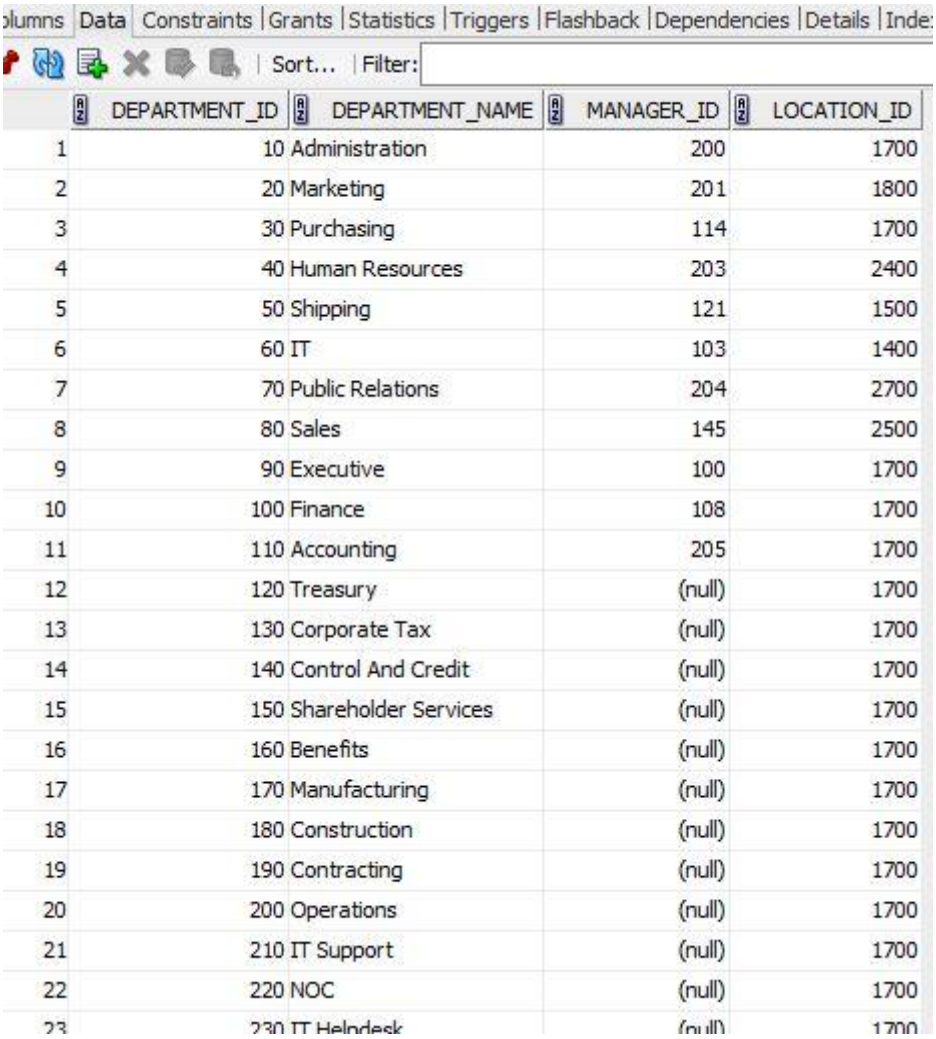
DEPARTMENT TABLE with Column name

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

Actions...

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
DEPARTMENT_ID	NUMBER(4,0)	No	(null)	1	1	Primary key c...
DEPARTMENT_NAME	VARCHAR2(30 BYTE)	No	(null)	2	(null)	A not null colu...
MANAGER_ID	NUMBER(6,0)	Yes	(null)	3	(null)	Manager_id o...
LOCATION_ID	NUMBER(4,0)	Yes	(null)	4	(null)	Location id wh...

DEPARTMENT TABLE with Data



olumns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Inde

Sort... | Filter:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700
8	80	Sales	145	2500
9	90	Executive	100	1700
10	100	Finance	108	1700
11	110	Accounting	205	1700
12	120	Treasury	(null)	1700
13	130	Corporate Tax	(null)	1700
14	140	Control And Credit	(null)	1700
15	150	Shareholder Services	(null)	1700
16	160	Benefits	(null)	1700
17	170	Manufacturing	(null)	1700
18	180	Construction	(null)	1700
19	190	Contracting	(null)	1700
20	200	Operations	(null)	1700
21	210	IT Support	(null)	1700
22	220	NOC	(null)	1700
23	230	IT Helpdesk	(null)	1700

Now, let me share all the types one by one.

1. Single Row Sub Query

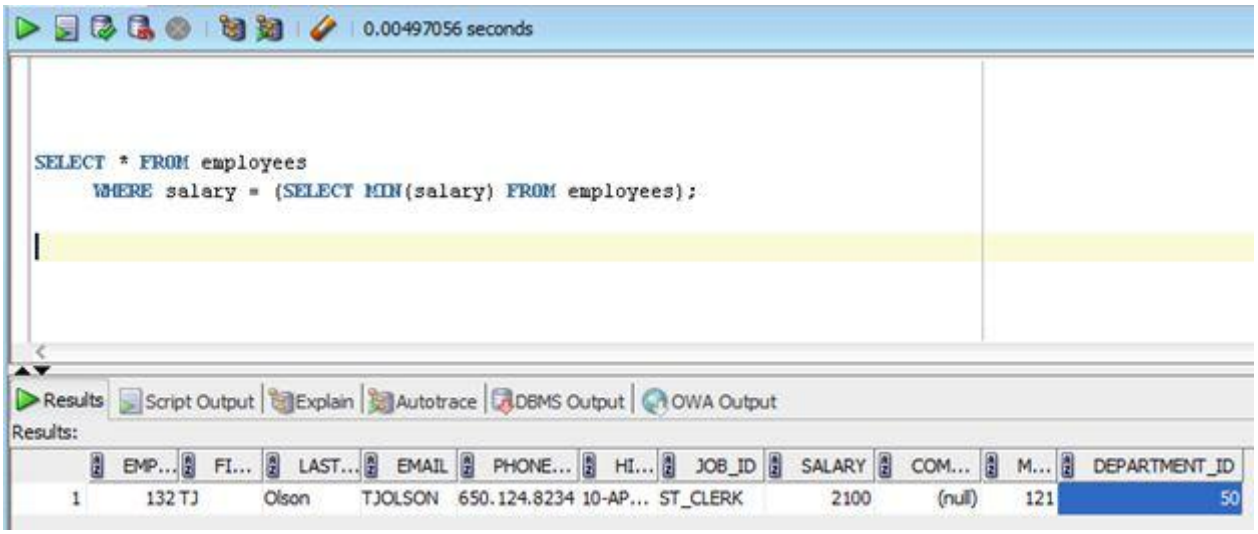
In a Single Row Sub Query the queries return a single/one row of results to the parent/main Query. It can include any of the following operators:

- = Equals to
- > Greater than
- < Less than
- >= Greater than Equals to
- <= Less than Equals to
- <> Not Equals to

Example

```
1. SELECT * FROM employees
2. WHERE salary = (SELECT MIN(salary) FROM employees);
```

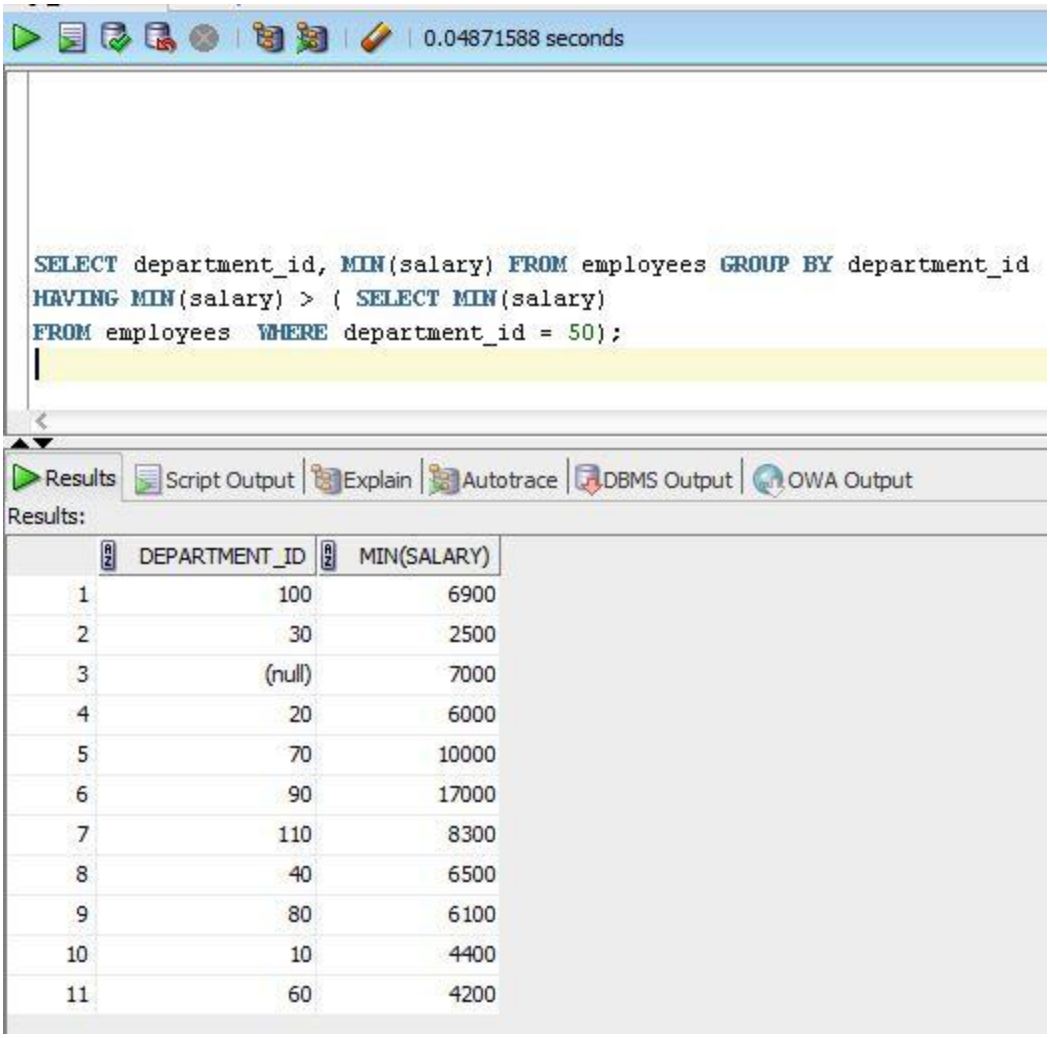
Execute the Query. The result will be as in the following:



Single Row Sub Query using HAVING Clause

```
1. SELECT department_id, MIN(salary)
2. FROM employees
3. GROUP BY department_id
4. HAVING MIN(salary) > ( SELECT MIN(salary)
5. FROM employees
6. WHERE department_id = 50);
```

Execute the Query, the result will be as in the following:



Multiple Row Sub Query

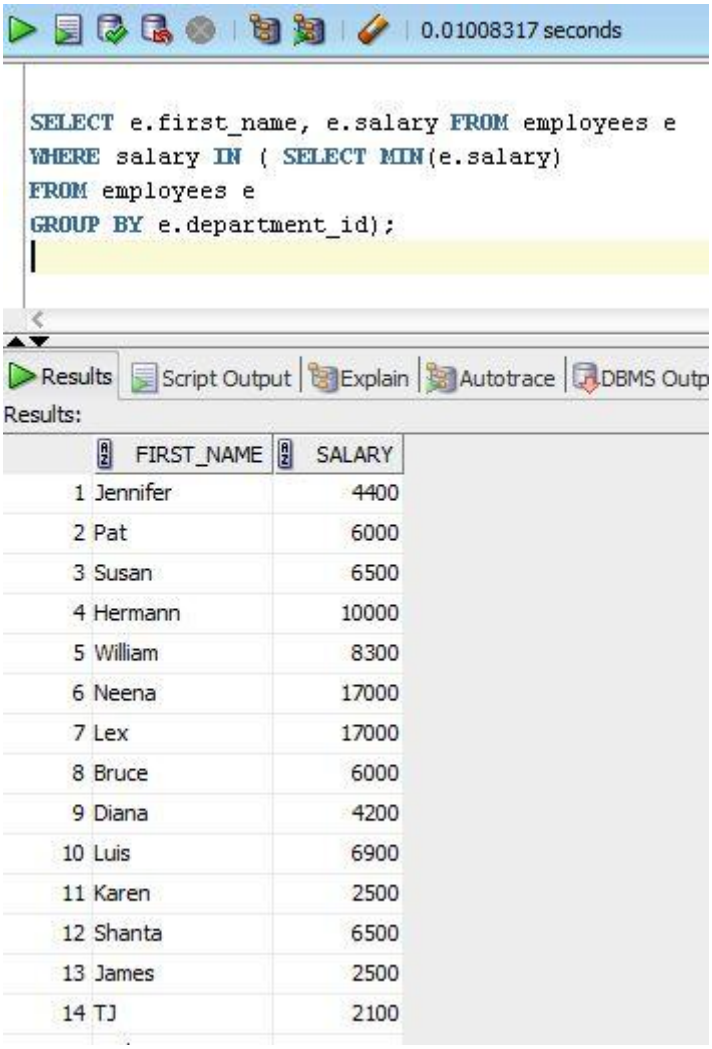
A Multiple Row Sub Query returns a result of multiple rows to the outer/main/parent query. It includes the following operators:

1. IN
2. ANY
3. ALL or EXISTS

Example

```
1. SELECT e.first_name, e.salary
2. FROM employees e
3. WHERE salary IN ( SELECT MIN(e.salary)
4. FROM employees e
5. GROUP BY e.department_id);
```

Execute the Query, then the result will be as in the following:

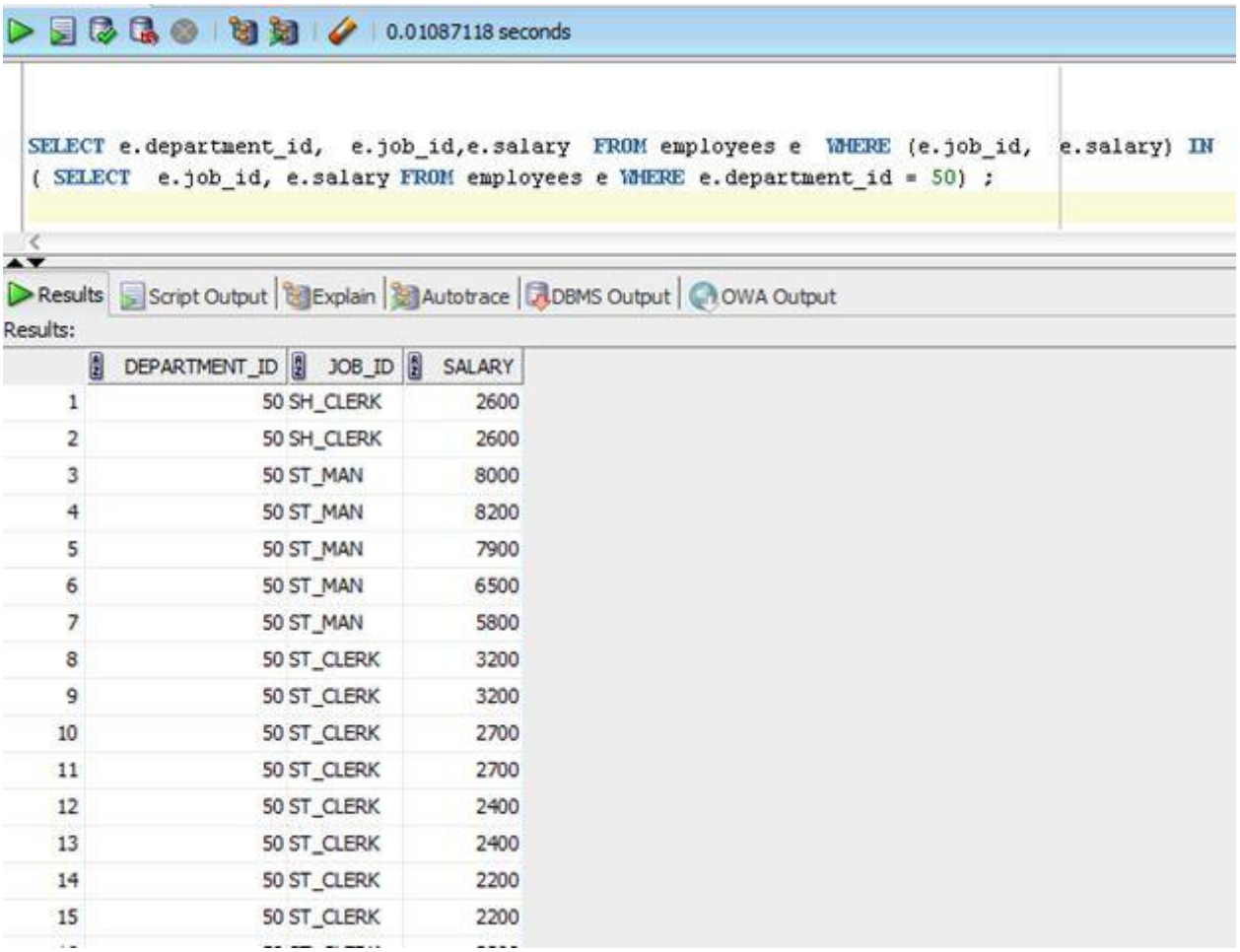


Multiple Column Sub Query

Multiple Column Sub Queries are queries that return multiple columns to the outer SQL query. It uses the IN operator for the WHERE and HAVING clause.

```
1. SELECT e.department_id,  e.job_id,e.salary
2. FROM employees e
3. WHERE (e.job_id,  e.salary) IN ( SELECT  e.job_id, e.salary
4. FROM employees e
5. WHERE e.department_id = 50) ;
```

Execute the Query, then the result will be as in the following:

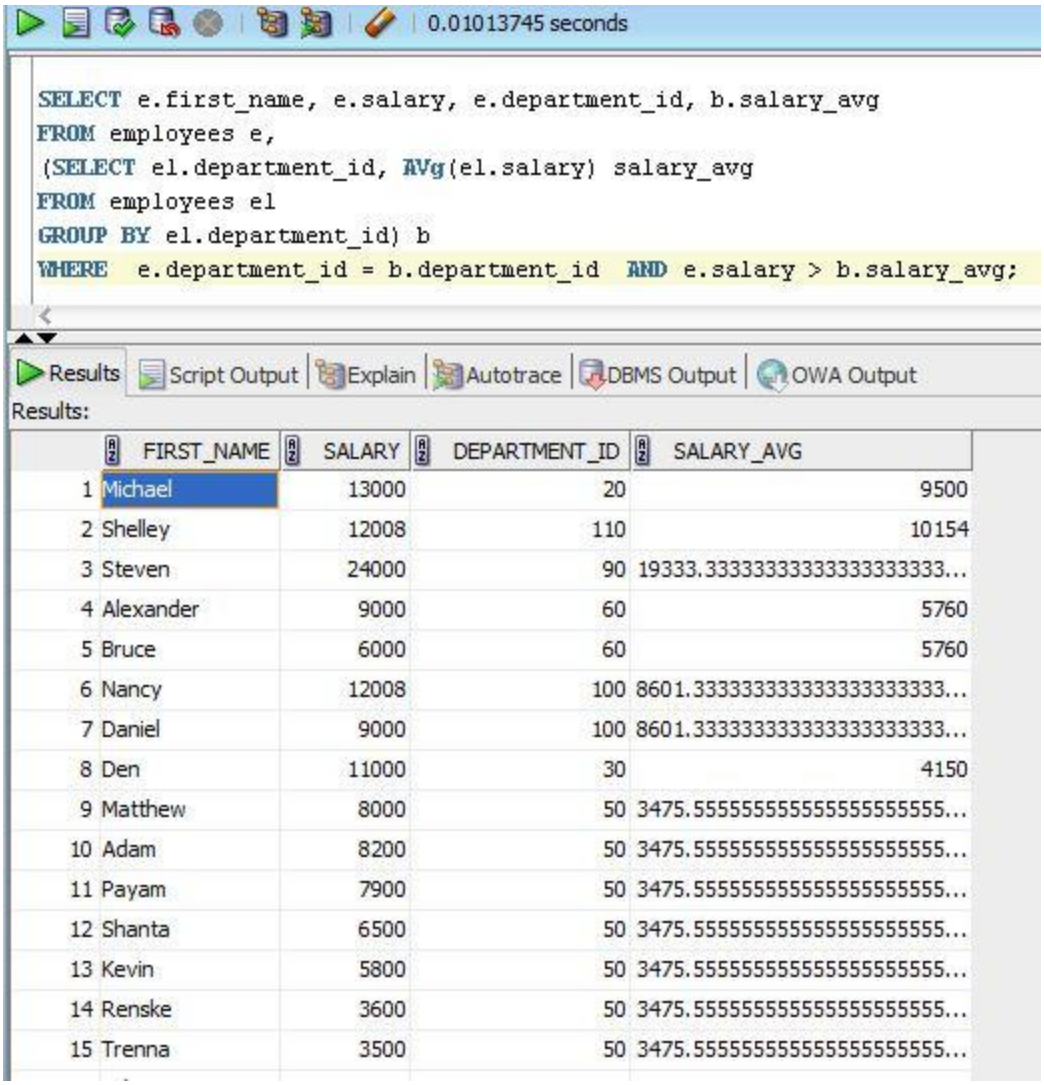


	DEPARTMENT_ID	JOB_ID	SALARY
1	50	SH_CLERK	2600
2	50	SH_CLERK	2600
3	50	ST_MAN	8000
4	50	ST_MAN	8200
5	50	ST_MAN	7900
6	50	ST_MAN	6500
7	50	ST_MAN	5800
8	50	ST_CLERK	3200
9	50	ST_CLERK	3200
10	50	ST_CLERK	2700
11	50	ST_CLERK	2700
12	50	ST_CLERK	2400
13	50	ST_CLERK	2400
14	50	ST_CLERK	2200
15	50	ST_CLERK	2200

Note: We can use a Sub Query using a FROM clause in the main query.

```
1. SELECT e.first_name, e.salary, e.department_id, b.salary_avg
2. FROM employees e,
3. (SELECT e1.department_id, AVg(e1.salary) salary_avg
4. FROM employees e1
5. GROUP BY e1.department_id) b
6. WHERE  e.department_id = b.department_id  AND e.salary > b.salary_avg;
```

Execute the Query, then the result will be as in the following:

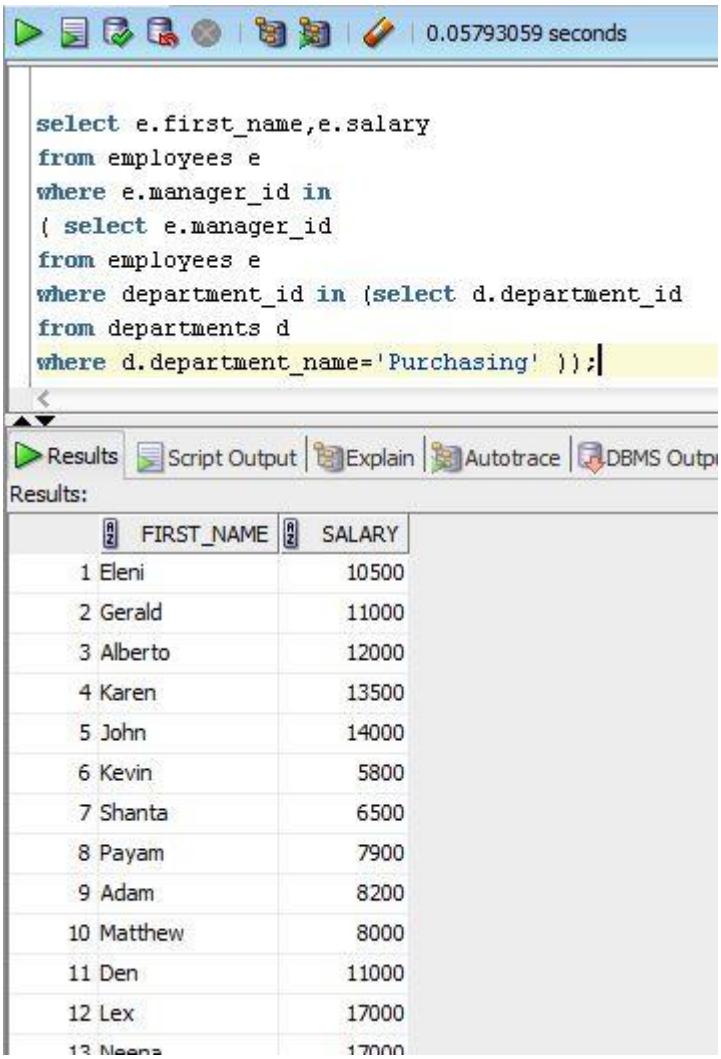


Nested Sub Query

When we write a Sub Query in a WHERE and HAVING clause of another Sub Query then it is called a nested Sub Query.

```
1. SELECT e.first_name,e.salary
2. FROM employees e
3. WHERE e.manager_id in
4. ( SELECT e.manager_id
5. FROM employees e
6. WHERE department_id in (select d.department_id
7. FROM departments d
8. WHERE d.department_name='Purchasing' ));
```

Execute the Query, then the result will be as in the following:

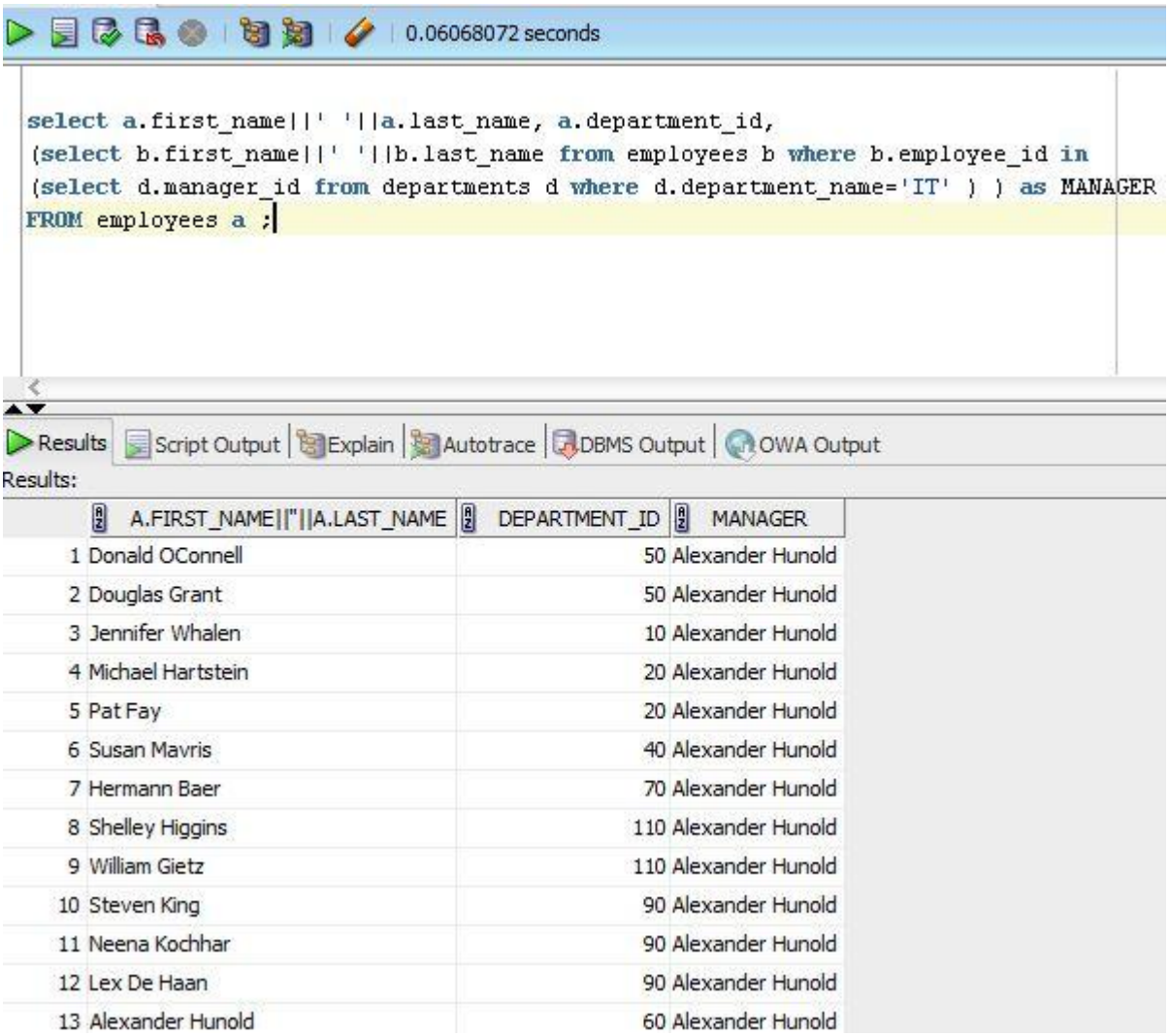


Correlated Sub Query

A Correlated Sub Query contains a reference to a table that appears in the outer query. It is used for row by row processing, in other words the Sub Query will execute row by row for the parent query.

```
1. SELECT a.first_name||' '||a.last_name, a.department_id,
2. (SELECT b.first_name||' '||b.last_name
3. FROM employees b
4. WHERE b.employee_id in
5. (SELECT d.manager_id
6. FROM departments d
7. WHERE d.department_name='IT' ) ) as MANAGER
8. FROM employees a ;
```

Execute the Query, then the result will be as in the following:



DBMS | Nested Queries in SQL

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. We will use **STUDENT, COURSE, STUDENT_COURSE** tables for understanding nested queries.

STUDENT

S_ID	S_NAME	S_ADDRESS	S_PHONE	S_AGE
S1	RAM	DELHI	9455123451	18
S2	RAMESH	GURGAON	9652431543	18
S3	SUJIT	ROHTAK	9156253131	20
S4	SURESH	DELHI	9156768971	18

COURSE

C_ID	C_NAME
C1	DSA
C2	Programming
C3	DBMS

STUDENT_COURSE

S_ID	C_ID
S1	C1
S1	C3
S2	C1
S3	C2
S4	C2
S4	C3

There are mainly two types of nested queries:

- **Independent Nested Queries:** In independent nested queries, query execution starts from innermost query to outermost queries. The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.

IN: If we want to find out **S_ID** who are enrolled in **C_NAME** 'DSA' or 'DBMS', we can write it with the help of independent nested query and IN operator. From **COURSE** table, we can find out **C_ID** for **C_NAME** 'DSA' or DBMS' and we can use these **C_ID**s for finding **S_ID**s from **STUDENT_COURSE** TABLE.

STEP 1: Finding **C_ID** for **C_NAME** ='DSA' or 'DBMS'

Select **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME** = 'DBMS'

STEP 2: Using **C_ID** of step 1 for finding **S_ID**

Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME** = 'DSA' or **C_NAME**='DBMS');

The inner query will return a set with members C1 and C3 and outer query will return those **S_ID**s for which **C_ID** is equal to any member of set (C1 and C3 in this case). So, it will return S1, S2 and S4.

Note: If we want to find out names of **STUDENT**s who have either enrolled in 'DSA' or 'DBMS', it can be done as:

Select S_NAME from **STUDENT** where **S_ID** IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

NOT IN: If we want to find out **S_ID**s of **STUDENT**s who have neither enrolled in 'DSA' nor in 'DBMS', it can be done as:

Select **S_ID** from **STUDENT** where **S_ID** NOT IN

(Select **S_ID** from **STUDENT_COURSE** where **C_ID** IN

(SELECT **C_ID** from **COURSE** where **C_NAME**='DSA' or **C_NAME**='DBMS'));

The innermost query will return a set with members C1 and C3. Second inner query will return those **S_ID**s for which **C_ID** is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those **S_ID**s where **S_ID** is not a member of set (S1, S2 and S4). So it will return S3.

Co-related Nested Queries: In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out **S_NAME** of **STUDENT**s who are enrolled in **C_ID** 'C1', it can be done with the help of co-related nested query as:

Select S_NAME from **STUDENT** S where EXISTS (select * from **STUDENT_COURSE** SC where S.**S_ID**=SC.**S_ID** and SC.**C_ID**='C1');

For each row of **STUDENT** S, it will find the rows from **STUDENT_COURSE** where S.**S_ID** = SC.**S_ID** and SC.**C_ID**='C1'. If for a **S_ID** from **STUDENT** S, atleast a row exists in **STUDENT_COURSE** SC with **C_ID**='C1', then inner query will return true and corresponding **S_ID** will be returned as output.

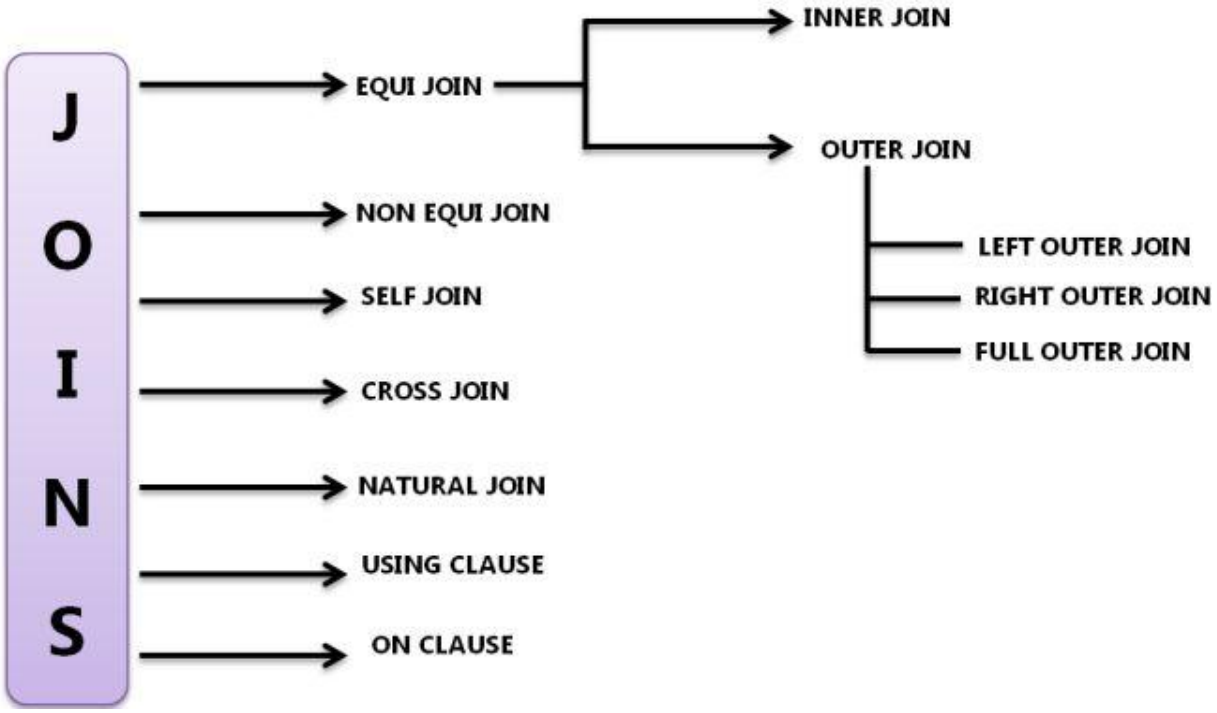
JOINS IN ORACLE

In Oracle, a join is the most powerful operation for merging information from multiple tables based on a common field. There are various types of joins but an INNER JOIN is the common of them.

Syntax

```
SELECT col1, col2, col3...
FROM table_name1, table_name2
WHERE table_name1.col2 = table_name2.col1;
```

Types Of Joins



To understand each of the preceding joins clearly we are assuming the following "CUSTOMER" and "ORDERS" tables:

```
CREATE TABLE Customer
      (
       Cust_id Number(10) NOT NULL,
       Cust_name varchar2(20),
            Country varchar2(20),
       Receipt_no Number(10),
       Order_id Number(10) NOT NULL,
      );

CREATE TABLE Orders
      (
       Order_id Number(10),
       Item_ordered varchar2(20),
       Order_date date
      );
```

Table: CUSTOMER

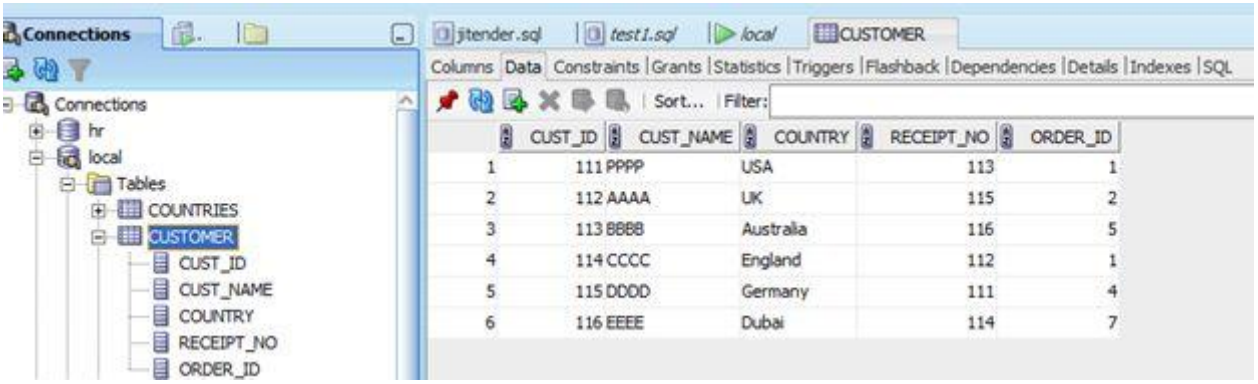
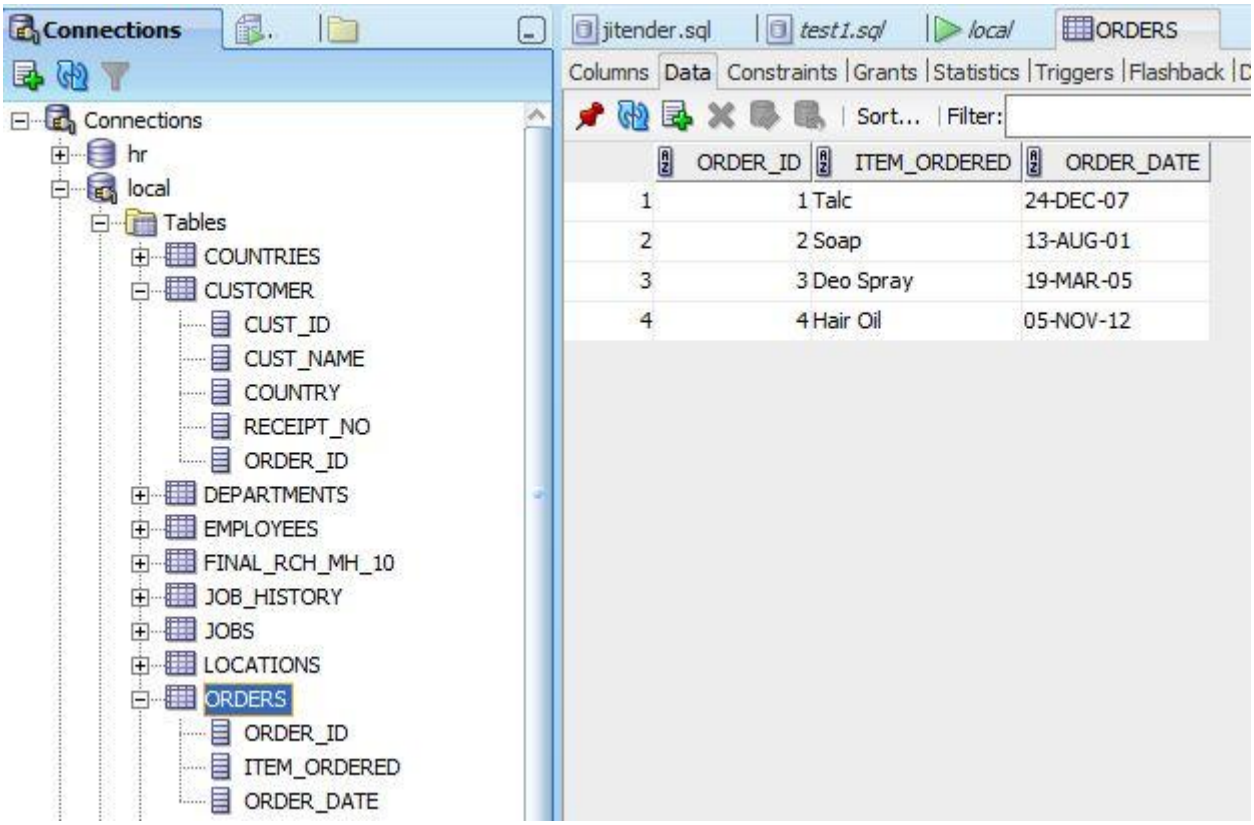


Table: ORDERS



First of all we will explain the "USING" clause and the "ON" clause.

1. Using Clause

To join a table using the USING Clause we write the following command.

Query

```
SELECT Cust_id, Cust_name, Country, item_Ordered, Order_date
FROM Customer C JOIN Orders O
USING (Order_id);
```

Execution of the query with result

```
SELECT cust_id, cust_name, country, Item_ordered, order_date
FROM customer c join orders o
USING (order_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DAT
1	111	PPPP	USA	Talc	24-DEC-07
2	114	CCCC	England	Talc	24-DEC-07
3	112	AAAA	UK	Soap	13-AUG-01
4	115	DDDD	Germany	Hair Oil	05-NOV-12

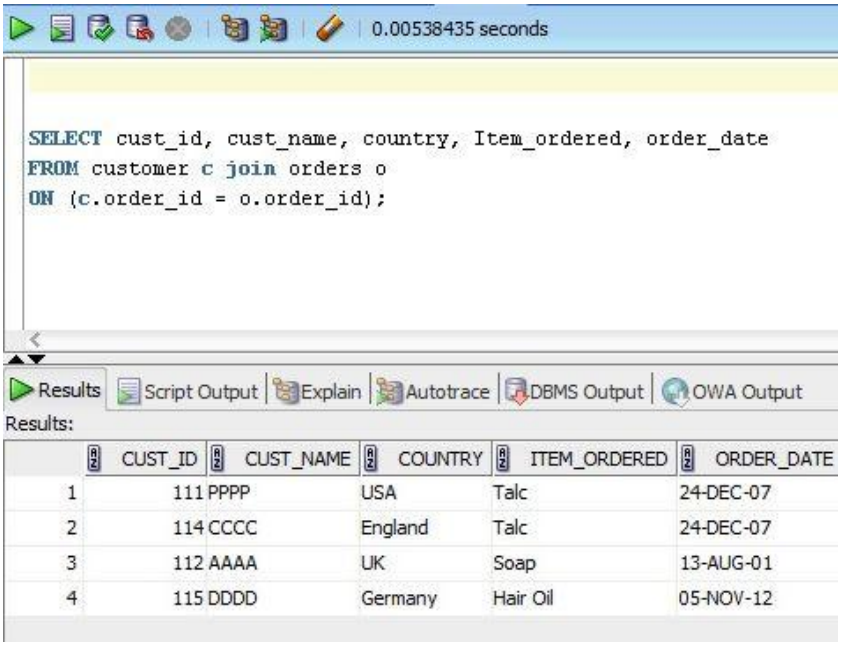
2. On Clause

To join a table using an ON Clause we write the following command:

Query

```
SELECT Cust_id, Cust_name, Country, item_Ordered, Order_date
FROM Customer C JOIN Orders O
USING (C.Order_id = O.Order_id);
```

Execution of the query with result



0.00538435 seconds

```
SELECT cust_id, cust_name, country, Item_ordered, order_date
FROM customer c join orders o
ON (c.order_id = o.order_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	114	CCCC	England	Talc	24-DEC-07
3	112	AAAA	UK	Soap	13-AUG-01
4	115	DDDD	Germany	Hair Oil	05-NOV-12

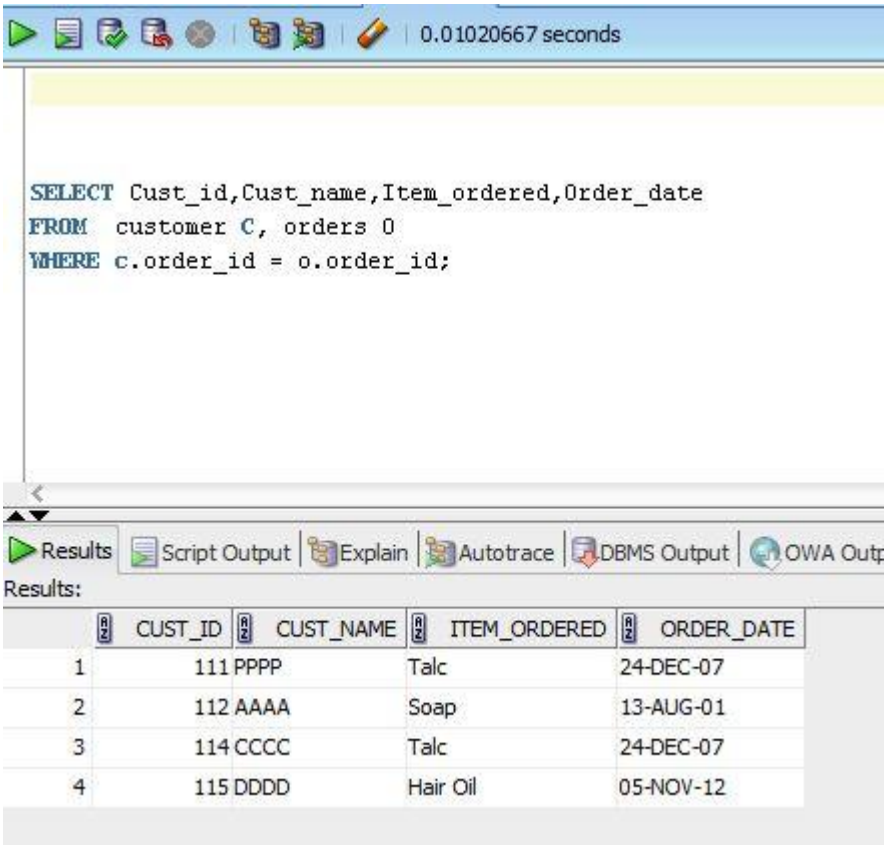
Equi Join

An Equi join is used to get the data from multiple tables where the names are common and the columns are specified. It includes the equal ("=") operator.

Example

```
SELECT Cust_id, Cust_name, item_Ordered, Order_date
FROM Customer C, Orders O
WHERE C.Order_id = O.Order_id;
```

Execution of the query with result



1. Inner Join

An Inner Join retrieves the matching records, in other words it retrieves all the rows where there is at least one match in the tables.

Example

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM Customer INNER JOIN Orders
USING (Order_id);
```

Execution of the query with result

```
SELECT cust_id, cust_name, country, Item_ordered, order_date
FROM customer INNER JOIN orders
USING (Order_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	114	CCCC	England	Talc	24-DEC-07
3	112	AAAA	UK	Soap	13-AUG-01
4	115	DDDD	Germany	Hair Oil	05-NOV-12

2. Outer Join

The records that don't match will be retrieved by the Outer join. It is of the following three types:

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

1. Left Outer Join

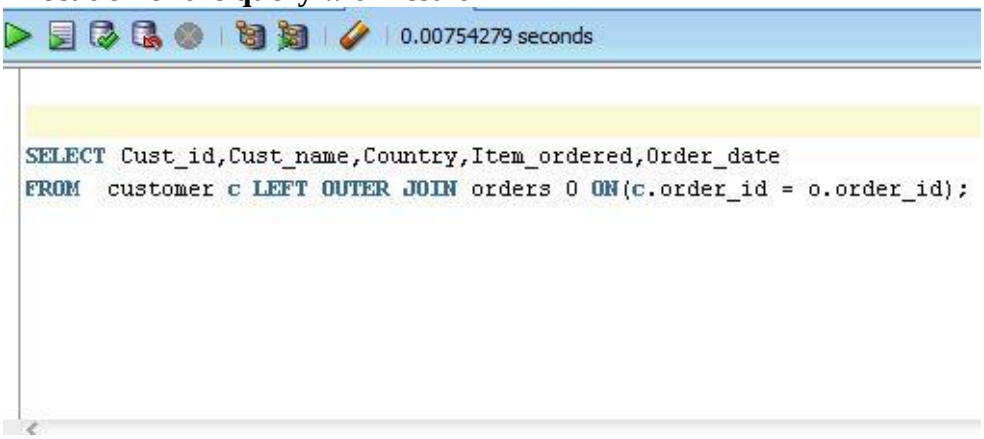
A Left outer join retrieves all records from the left hand side of the table with all the matched records. This query can be written in one of the following two ways.

Example

Method 1

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM customer C, LEFT OUTER JOIN Orders O
ON (C. Order_id = O.Order_id)
```

Execution of the query with result



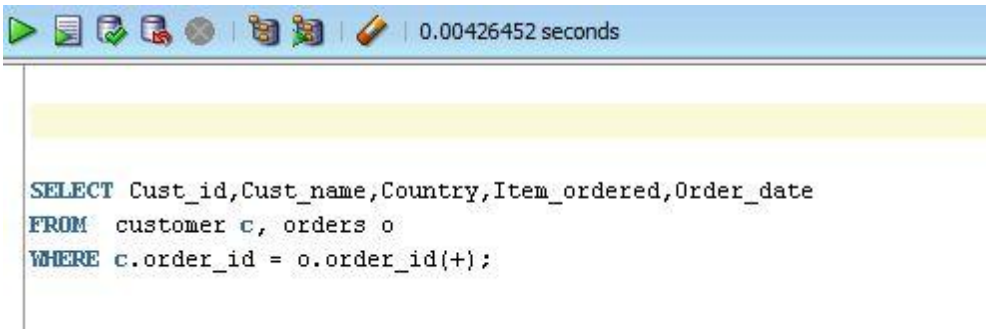
	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	112	AAAA	UK	Soap	13-AUG-01
3	113	BBBB	Australia	(null)	(null)
4	114	CCCC	England	Talc	24-DEC-07
5	115	DDDD	Germany	Hair Oil	05-NOV-12
6	116	EEEE	Dubai	(null)	(null)

Or:

Method 2

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM customer C, Orders O
WHERE C.Order_id = O.Order_id(+);
```

Execution of the query with result



	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	112	AAAA	UK	Soap	13-AUG-01
3	113	BBBB	Australia	(null)	(null)
4	114	CCCC	England	Talc	24-DEC-07
5	115	DDDD	Germany	Hair Oil	05-NOV-12
6	116	EEEE	Dubai	(null)	(null)

2. Right Outer Join

A Right Outer Join retrieves the records from the right hand side columns.

Example

Method 1

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM customer C, RIGHT OUTER JOIN Orders O
ON (C. Order_id = O.Order_id)
```

Execution of the query with result

```
SELECT Cust_id,Cust_name,Country,Item_ordered,Order_date
FROM  customer c RIGHT OUTER JOIN orders O ON(c.order_id = o.order_id);
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	114	CCCC	England	Talc	24-DEC-07
3	112	AAAA	UK	Soap	13-AUG-01
4	(null)	(null)	(null)	Deo Spray	19-MAR-05
5	115	DDDD	Germany	Hair Oil	05-NOV-12

Or: Method 2

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM customer C, Orders O
WHERE C.Order_id(+)= O.Order_id;
```

Execution of the query with result

0.00778223 seconds

```
SELECT Cust_id,Cust_name,Country,Item_ordered,Order_date
FROM  customer c, orders o
WHERE c.order_id(+) = o.order_id;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	114	CCCC	England	Talc	24-DEC-07
3	112	AAAA	UK	Soap	13-AUG-01
4	(null)	(null)	(null)	Deo Spray	19-MAR-05
5	115	DDDD	Germany	Hair Oil	05-NOV-12

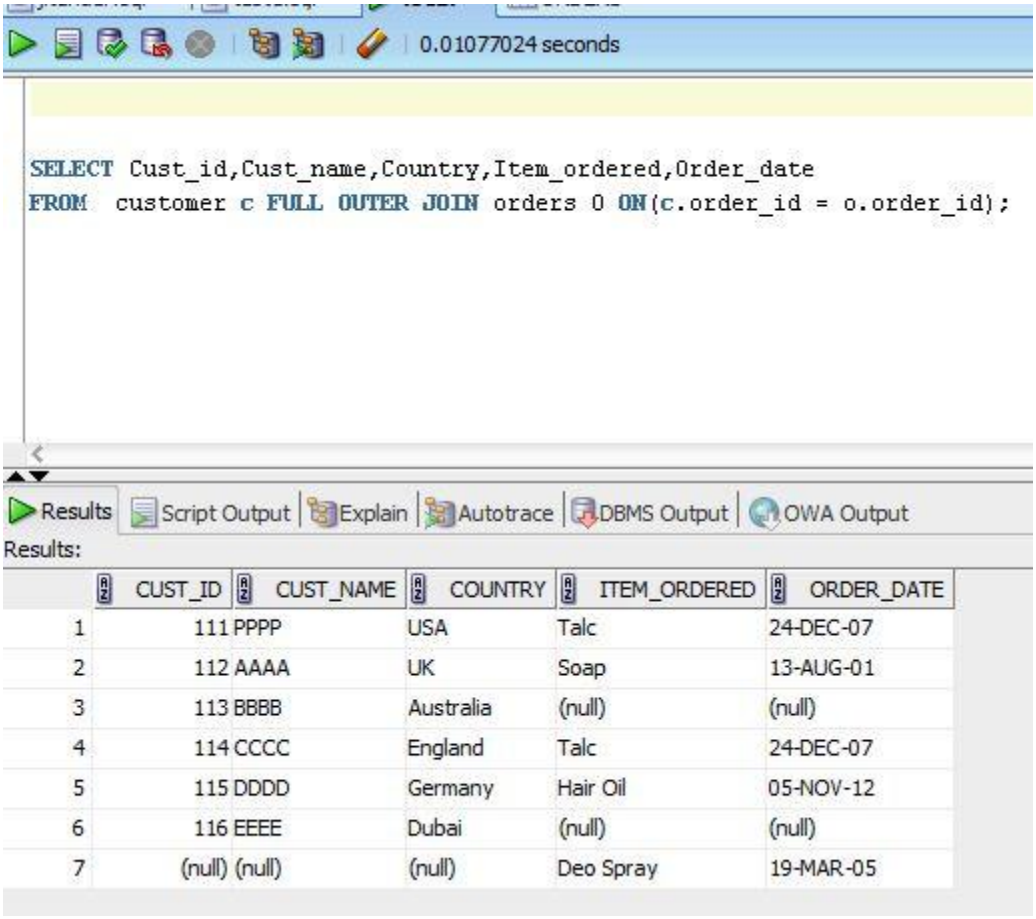
3. Full Outer Join

To retrieve all the records, both matching and unmatched from all the tables then use the FULL OUTER JOIN.

Example

```
SELECT Cust_id, Cust_name, Country, item_ordered, Order_date
FROM customer C, FULL OUTER JOIN Orders OON (C. Order_id = O.Order_id)
```

Execution of the query with result



Joins in Oracle: Part 2

2. Non-Equi Join

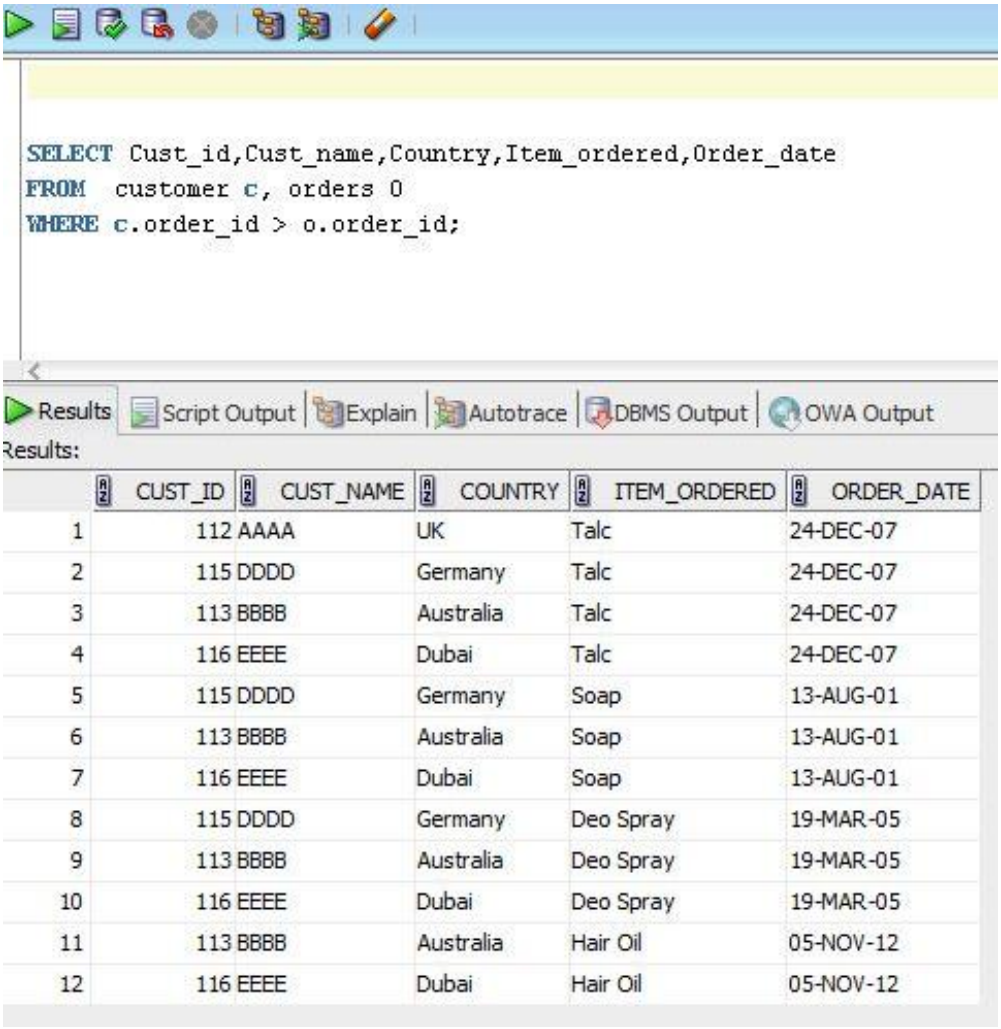
A Non-Equi join is based on a condition using an operator other than equal to "=".

Example

```
SELECT Cust_id, Cust_name, Country, Item_ordered, Order_date
```

```
FROM Customer C, Oredrs O
WHERE C. Order_id > O.Order_id;
```

Execution of the query with result:



```
SELECT Cust_id,Cust_name,Country,Item_ordered,Order_date
FROM  customer c, orders O
WHERE c.order_id > o.order_id;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	112	AAAA	UK	Talc	24-DEC-07
2	115	DDDD	Germany	Talc	24-DEC-07
3	113	BBBB	Australia	Talc	24-DEC-07
4	116	EEEE	Dubai	Talc	24-DEC-07
5	115	DDDD	Germany	Soap	13-AUG-01
6	113	BBBB	Australia	Soap	13-AUG-01
7	116	EEEE	Dubai	Soap	13-AUG-01
8	115	DDDD	Germany	Deo Spray	19-MAR-05
9	113	BBBB	Australia	Deo Spray	19-MAR-05
10	116	EEEE	Dubai	Deo Spray	19-MAR-05
11	113	BBBB	Australia	Hair Oil	05-NOV-12
12	116	EEEE	Dubai	Hair Oil	05-NOV-12

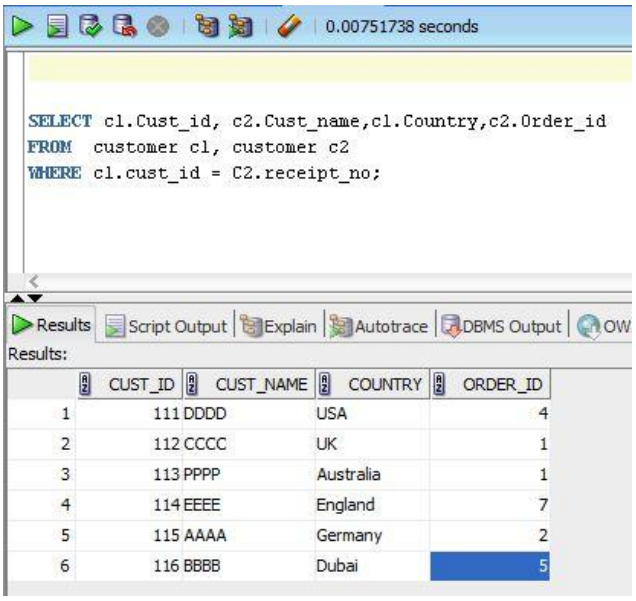
3. Self-join

When a table is joined to itself only then that condition is called a self-join.

Example

```
SELECT C1.Cust_id, C2.Cust_name, C1.Country, C2.Order_id
FROM Customer C1, Customer C2
WHERE C. Cust_id > O.Order_id;
```

Execution of the query with result:



	CUST_ID	CUST_NAME	COUNTRY	ORDER_ID
1	111	DDDD	USA	4
2	112	CCCC	UK	1
3	113	PPPP	Australia	1
4	114	EEEE	England	7
5	115	AAAA	Germany	2
6	116	BBBB	Dubai	5

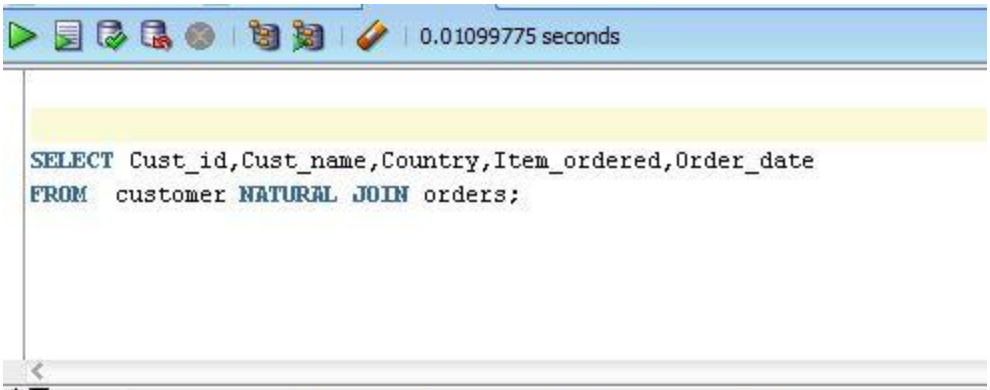
4. Natural Join

A natural join is just like an equi-join since it compares the common columns of both tables.

Example

```
SELECT Cust_id, Cust_name, Country, Item_ordered, Order_date
FROM Customer, NATURAL JOIN Orders;
```

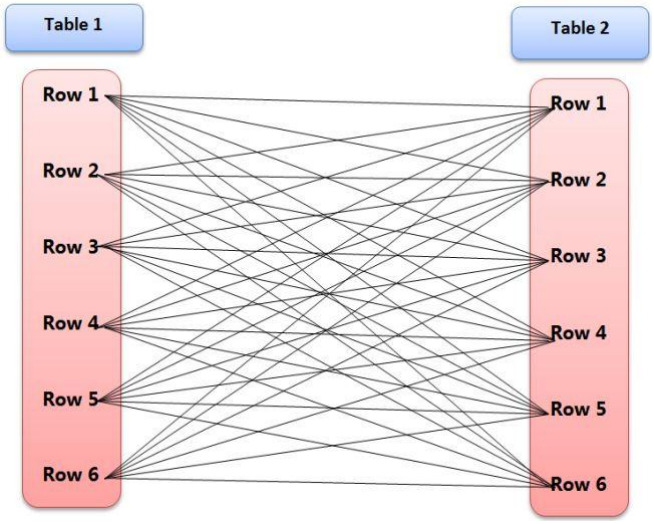
Execution of the query with result:



	CUST_ID	CUST_NAME	COUNTRY	ITEM_ORDERED	ORDER_DATE
1	111	PPPP	USA	Talc	24-DEC-07
2	112	AAAA	UK	Soap	13-AUG-01
3	114	CCCC	England	Talc	24-DEC-07
4	115	DDDD	Germany	Hair Oil	05-NOV-12

5. Cross Join

This join is a little bit different from the other joins since it generates the Cartesian product of two tables as in the following:



Syntax
SELECT * FROM table_name1 CROSS JOIN table_name2;

Example
SELECT Cust_id, Cust_name, Country, Item_ordered, Order_date FROM Customer,
CROSS JOIN Orders;

SQL - Using Joins

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables −

Table 1 − CUSTOMERS Table

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
   FROM CUSTOMERS, ORDERS
   WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.
- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

Let us now discuss each of these joins in detail.

SQL - INNER JOINS

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

The basic syntax of the **INNER JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the INNER JOIN as follows −

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

SQL - LEFT JOINS

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **LEFT JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2 − Orders Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
```

```
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the LEFT JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

SQL - RIGHT JOINS

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **RIGHT JOIN** is as follow.

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the RIGHT JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

SQL - FULL JOINS

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax

The basic syntax of a **FULL JOIN** is as follows −

```
SELECT table1.column1, table2.column2...
```

```
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using FULL JOIN as follows.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   FULL JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
```

```
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use **UNION ALL** clause to combine these two JOINS as shown below.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

SQL - SELF JOINS

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax

The basic syntax of SELF JOIN is as follows −

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Here, the WHERE clause could be any given expression based on your requirement.

Example Consider the following table.

CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| 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, let us join this table using SELF JOIN as follows −

```
SQL> SELECT  a.ID, b.NAME, a.SALARY
   FROM CUSTOMERS a, CUSTOMERS b
   WHERE a.SALARY < b.SALARY;
```

This would produce the following result −

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

SQL - CARTESIAN or CROSS JOINS

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows −

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

Example

Consider the following two tables.

Table 1 − CUSTOMERS table is as follows.

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

Table 2: ORDERS Table is as follows −

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using CARTESIAN JOIN as follows −

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS, ORDERS;
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

Result: The Database Querying – Simple Queries, Nested Queries, Sub Queries and Joins are executed successfully.

VIEWS, SEQUENCES, SYNONYMS

Aim:

To create database and apply Views, Sequences, Synonyms.

SQL - Using Views

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 a type 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 in such a way 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 a similar way as you use them in a 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 |
+----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

Sql > create view customers_view as select name, age from customers;

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

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 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

Create view customers_view as select name, age from customers Where age is not null with check option;

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- 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 that view. The following code block has 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 the same would reflect in the view itself. Now, try to query the base table and the 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 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

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 the same would reflect in the view itself. Now, try to query the base table and the 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. The syntax is very simple and is given below −

drop view view_name;

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

drop view customers_view;

SQL | SEQUENCES

Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.

- A sequence is a user defined schema bound object that generates a sequence of numeric values.
- Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when exceeds max_value.

Syntax:

CREATE SEQUENCE sequence_name START WITH initial_value INCREMENT BY increment_value MINVALUE minimum value MAXVALUE maximum value CYCLE|NOCYCLE ;

sequence_name: Name of the sequence.

initial_value: starting value from where the sequence starts.
Initial_value should be greater than or equal
to minimum value and less than equal to maximum value.

increment_value: Value by which sequence will increment itself.
Increment_value can be positive or negative.
minimum_value: Minimum value of the sequence.
maximum_value: Maximum value of the sequence.
cycle: When sequence reaches its set_limit it starts from beginning.
nocycle: An exception will be thrown if sequence exceeds its max_value.

Example

Following is the sequence query creating sequence in ascending order.

- **Example 1:**
- CREATE SEQUENCE sequence_1
- start with 1
- increment by 1
- minvalue 0
- maxvalue 100
- cycle;

Above query will create a sequence named *sequence_1*.Sequence will start from 1 and will be incremented by 1 having maximum value 100. Sequence will repeat itself from start value after exceeding 100.

Example 2:
Following is the sequence query creating sequence in descending order.

```
CREATE SEQUENCE sequence_2

start with 100 increment by -1 minvalue 1 maxvalue 100 cycle;
```

Above query will create a sequence named *sequence_2*.Sequence will start from 100 and should be less than or equal to maximum value and will be incremented by -1 having minimum value 1.

Example to use sequence : create a table named students with columns as id and name.

```
CREATE TABLE students( ID number(10),NAME char(20));
```

Now insert values into table

INSERT into students VALUES(sequence_1.nextval,'Ramesh');
INSERT into students VALUES(sequence_1.nextval,'Suresh');

where *sequence_1.nextval* will insert id's in id column in a sequence as defined in sequence_1.
Output:

```
| ID  |      NAME      |
------------------------
|  1  |     Ramesh     |
|  2  |     Suresh     |
```

Oracle / PLSQL: Synonyms

This Oracle tutorial explains how to **create and drop synonyms** in Oracle with syntax and examples.

Description

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

Create Synonym (or Replace)

You may wish to create a synonym so that users do not have to prefix the table name with the schema name when using the table in a query.

Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name
  FOR [schema .] object_name [@ dblink];
```

OR REPLACE
: Allows you to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC
: It means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema
: The appropriate schema. If this phrase is omitted, Oracle assumes that you are referring to your own schema.

object_name
: The name of the object for which you are creating the synonym. It can be one of the following:

- table
- view
- sequence
- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object

- synonym

Example

Let's look at an example of how to create a synonym in Oracle.

For example:

create public synonym suppliers for app.suppliers;

This first CREATE SYNONYM example demonstrates how to create a synonym called *suppliers*. Now, users of other schemas can reference the table called *suppliers* without having to prefix the table name with the schema named *app*. For example:

SELECT * FROM suppliers;

If this synonym already existed and you wanted to redefine it, you could always use the *OR REPLACE* phrase as follows:

CREATE OR REPLACE PUBLIC SYNONYM suppliers FOR app.suppliers;

Drop synonym

Once a synonym has been created in Oracle, you might at some point need to drop the synonym.

Syntax

The syntax to drop a synonym in Oracle is:

DROP [PUBLIC] SYNONYM [schema .] synonym_name [force];

PUBLIC
: Allows you to drop a public synonym. If you have specified *PUBLIC*, then you don't specify a *schema*.

force
: It will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use *force* as it can cause invalidation of Oracle objects.

Example

Let's look at an example of how to drop a synonym in Oracle.

For example:

drop public synonym suppliers;

This DROP statement would drop the synonym called *suppliers* that we defined earlier.

Result:

The database created and applied Views, Sequences, Synonyms in a database.

DATABASE PROGRAMMING: IMPLICIT AND EXPLICIT CURSORS

Aim:

To create a database and apply Implicit and Explicit Cursors

PL/SQL - Cursors

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

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 will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
```

```
   END IF;
END;
/
```

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

CURSOR c_customers IS SELECT id, name, address FROM customers;

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
```

6 Komal MP

PL/SQL procedure successfully completed.

CURSORS FOR LOOP IN ORACLE

In continuation of my previous article about Oracle Cursors, here I will share the use of cursors for loops with an appropriate example.

The process of opening, fetching, and closing is handled implicitly by a cursors FOR LOOP. If there is a need to FETCH and PROCESS each and every record from a cursor , the cursor FOR LOOP is helpful for that.

Let's learn about the implicit cursor for loop using the following table ORDERS and example:

Supp_id	Supp_name	Items	Customer_id
111	AAA	DEO	#128
222	BBB	Perfume	#32
333	CCC	Perfume	#121
444	DDD	DEO	#88
555	EEE	DEO	#199
666	FFF	Perfume	#02
777	GGG	DEO	#105

Implicit Cursor for Loop

Note: Here, an implicit cursor FOR LOOP statement prints the name of the supplier and supplier id of the entire item named as DEO whose customer has an ID greater than 100.

```
1.  BEGIN
2.  //Beginning of FOR LOOP//
3.    FOR item IN (
4.        SELECT ,supp_id,supp_name
5.      FROM Orders
6.      WHERE Supp_id LIKE '%DEO%'
7.      AND Customer_id > 120
8.      ORDER BY supp_name
9.    )
10.  LOOP
11.    DBMS_OUTPUT.PUT_LINE
12.      ('Supplier Name = ' || item.supp_name || ', Supplier ID = ' || item.Supp_id);
13.  END LOOP;
14. //End of FOR LOOP//
15. END;
```

Result

Supp_id	Supp_name
Supplier ID = 111	Supplier Name = AAA
Supplier ID = 555	Supplier Name = EEE
Supplier ID = 777	Supplier Name = GGG

Explicit Cursor for Loop

Note: In the following example, an explicit cursor FOR LOOP statement prints the name of the supplier and supplier id of the entire item named PERFUME whose customer has an ID lesser than 100.

```
1.  DECLARE
2.   CURSOR C1 IS
3.     SELECT ,supp_id,supp_name
4.     FROM Orders
5.     WHERE Supp_id LIKE '%PERFUME%'
6.     AND Customer_id < 100
7.     ORDER BY supp_name;
8.  BEGIN
9.
10. //Beginning of FOR LOOP//
11.   FOR item IN C1
12.   LOOP
13.     DBMS_OUTPUT.PUT_LINE
14.      ('Supplier Name = ' || item.supp_name || ', Supplier ID = ' || item.Supp_id);
15.   END LOOP;
16. //End of FOR LOOP//
17.
18. END;
```

Result

Supp_id	Supp_name
Supplier ID = 222	Supplier Name = BBB
Supplier ID = 666	Supplier Name = FFF

Nested Cursor for Loop

Cursors can be nested, in other words a cursor can have another cursor inside it.

If there is one main cursor called a parent cursor and two small/child cursors then each time the main cursor makes a single loop, it will loop through each small cursor once and then begin a second round.

Here is an example of a nested cursor with an assumed table customer:

Cust_id	First_name	Last_name	Zip Code	City	State
111	Rahul	Tondon	456246	Bareilly	Uttar Pradesh
222	Karan	Siddhu	455633	Mumbai	Maharashtra
333	Sandhiya	Rathi	345345	Ahemdabad	Gujarat
444	Meenakshi	Gautam	574567	Dehradun	Uttrakhand
555	Saras	Katyal	345335	Dharamshala	Himachal Pradesh

```
1.
    DECLARE
2.   cur_zip zipcode.zip%TYPE;
3.  //Cursor one cur_zip//
4.   CURSOR a_zip IS
5.  // variable a_zip initialised//
6.   SELECT zip, city, state
7.   FROM zipcode
8.   WHERE state = 'CT';
9.   CURSOR c_Customer IS
10.  SELECT first_name, last_name
11.  FROM Customer
12.  WHERE zip = cur_zip;
13.  BEGIN
14.  FOR b_zip IN a_zip
15.  LOOP
16.  cur_zip := b_zip.zip;
17.  DBMS_OUTPUT.PUT_LINE (CHR(10));
18.  DBMS_OUTPUT.PUT_LINE ('Customers living in '||
19.  b_zip.city);
20.  FOR b_customer in cur1_customer
21.  LOOP
22.  DBMS_OUTPUT.PUT_LINE (b_customer.first_name||
23.  ' '||b_customer.last_name);
24.  END LOOP;
25.  END LOOP;
26.  END;
```

Result:

Data base created and applied Implicit and Explicit Cursors in a database.

PROCEDURES AND FUNCTIONS

Aim:

To create a database and apply Procedures and Functions

PL/SQL - Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter '**PL/SQL - Packages**'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named '**greetings**' can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
   greetings;
END;
/
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

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

```
Minimum of (23, 45) : 23
```

```
PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

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

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following −

```
findMin(x => a, y => b, z => c, m => d);
```

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

PL/SQL - Functions

In this chapter, we will discuss the functions in PL/SQL. A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

use the CUSTOMERS table

```
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 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM customers;

  RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

Function created.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

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

Total no. of Customers: 6

PL/SQL procedure successfully completed.

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
  a number;
  b number;
  c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;
  RETURN z;
END;
BEGIN
  a:= 23;
  b:= 45;
  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

n! = n*(n-1)!

= n*(n-1)*(n-2)!
...
= n*(n-1)*(n-2)*(n-3)... 1

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
  num number;
  factorial number;

FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

Step 1

The following will create a table in Oracle. For example I will create a table for customer details.

```
1. CREATE TABLE CUSTOMER (
2. NAME VARCHAR2(20),
3. GENDER VARCHAR2(7),
4. ADDRESS VARCHAR2(100));
```

Step 2

After creating the table, write a Stored Procedure for an **insert**:

```
CREATE OR REPLACE PROCEDURE INSERTcustomer (
p_name CUSTOMER.NAME%TYPE,
p_gender CUSTOMER.GENDER%TYPE,
p_address CUSTOMER.ADDRESS%TYPE)
IS
BEGIN
INSERT INTO CUSTOMER (NAME, GENDER, ADDRESS)
VALUES (p_name, p_gender, p_address);
COMMIT;
END;
/
```

Step 3
Stored Procedure for an **update**:

```
CREATE OR REPLACE PROCEDURE UPDATEcustomer (
p_name IN CUSTOMER.NAME%TYPE,
p_gender IN CUSTOMER.GENDER%TYPE,
p_address IN CUSTOMER.ADDRESS%TYPE)
IS
BEGIN
UPDATE CUSTOMER SET NAME=p_name, GENDER=p_gender, ADDRESS=p_address WHERE NAME=p_name;
COMMIT;
END;
/
```

Step 4
Stored Procedure for a **select**:

```
CREATE OR REPLACE PROCEDURE SELECTcustomer (
p_name IN CUSTOMER.NAME%TYPE,
p_customer_display OUT SYS_REFCURSOR)
IS
BEGIN
OPEN p_customer_display for SELECT NAME, GENDER, ADDRESS FROM CUSTOMER WHERE NAME=p_name;
END;
/
```

Step 5
Stored Procedure for a **delete:**

```
CREATE OR REPLACE PROCEDURE DELETEcustomer (
p_name IN CUSTOMER3.NAME%TYPE)
IS
BEGIN
DELETE FROM CUSTOMER WHERE NAME=p_name;
END;
/
```

Result:

Database created successfully. Procedures and Functions are executed successfully.

TRIGGERS

Aim:

To build database and apply triggers.

PL/SQL - Triggers

In this chapter, we will discuss Triggers in PL/SQL. 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).

Learn About Different Type Of Triggers In Oracle

Overview

Database triggers are specialized stored programs. Oracle engine allows the definition of the procedures, which are implicitly executed when an insert, update, or delete is issued in a table from SQL or through the Application, and the trigger automatically executes a DML statement. They are not called directly, but are triggered by the events in the database. They run between the time, when you issue a command and the time, you perform the database management system action. You can write the triggers in PL/SQL.

Introduction

PL/SQL Type of Triggers are based on how they are triggered.

Before Triggers: These triggers are fired before the SQL statement trigger (INSERT, UPDATE, DELETE) is executed. The execution of the triggering SQL statement is stopped, depending on the various conditions to be fulfilled in the BEFORE trigger.

After Triggers: These triggers are fired after the triggering SQL statement (INSERT, UPDATE, DELETE) is executed. The triggering SQL statement is executed first, followed by the code of the trigger.

ROW Trigger: The triggers are fired for each and every record, which is inserted or updated or deleted from a table.

Statement Trigger: The trigger is fired for each row of the DML operation, being performed on a table. We cannot access the column values for the records being inserted, updated, deleted on the table nor the individual records.

PL/SQL Triggers Syntax Description

CREATE or REPLACE TRIGGER trigger_name: Creates a trigger with the given name, else overwrites an existing trigger with the same name.

{BEFORE , AFTER }: Indicates where should trigger be fired. BEFORE trigger executes before when statement executes before time or AFTER trigger executes, after when statement executes after time.

{INSERT , UPDATE , DELETE}: Determines the performing trigger event. More than one triggering events can be used together, separated by OR keyword.

ON Table Name: Determines the performed trigger event in the selected table.

[Referencing {old AS old, new AS new}]: Reference the old and new values of the data, being changed. : old is used for existing row to perform and : new is used to execute a new row to perform. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference the old values, when inserting a record or new values, or when deleting a record, because they do not exist.

Note

Insert has no :OLD value (before execution) and have : NEW value (After execution).
Delete has no : OLD value but it has :NEW value.
Update has both : OLD and : NEW value.
for each row: Trigger must fire, when each row gets affected (Row Level Trigger) or just once, when the entire SQL statement is executed (Statement Level trigger).
WHEN (condition): Valid only for row level triggers. The trigger is fired only for the rows, which satisfy the specified condition.

There are various events on which a trigger can be written, such as:

1. System events
    - Database startup and shutdown.
    - Server error message events.
2. User events
    - User login and logoff.
    - DDL statements (CREATE, ALTER, and DROP).
    - DML statements (INSERT, DELETE, and UPDATE).

Based on the above condition, we can classify the trigger into five categories: DML trigger, DDL trigger, Compound triggers, Instead-Of triggers and System or database event triggers. Out of which, here I am discussing mainly DDL and DML triggers.

DDL Trigger

DDL triggers fire, when you create, change or remove objects in a database. They support both before and after event triggers and work at the database or schema level.

DDL event supported

alter, analyze, associate statistics, audit, comment, create, DDL, disassociate statistics, drop, grant, noaudit, rename, revoke, truncate .

There are a number of event attribute functions, which can be used to get user, client or system information, commonly used ones are given below:

Triggers can 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 will 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 will 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, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will 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 the SQL prompt, it produces the following result −

Trigger created.

The following points need to be considered here −

- 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.
- The 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 the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the 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
```

Example.1

The given DDL trigger prevents truncating the table on the schema level.

```
1. SQL> create or replace trigger prevent_truncates
2.  before truncate on schema
3.  begin
4.  raise_application_error(-20001,'TRUNCATE not permitted');
5.  end;
6.  /
```

Trigger created.

```
1. SQL> create table salary_bk as select * from salary;
```

Table created.

```
1. SQL> select * from salary_bk;
```

ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
--- --- --- --- --- --- --- ---
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 19500 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 19500 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 19500 1500 1200 1800 22000

```
1. SQL> truncate table salary_bk;
2. truncate table salary_bk
3. *
```

ERROR at line 1

ORA-00604: error occurred at recursive SQL level 1
ORA-20001: TRUNCATE not permitted
ORA-06512: at line 2
SQL>

Example.2

The below given trigger updates every create statement, which happens in the schema level into the log_table.

```
1. SQL> CREATE TABLE log_table(
2.  user_name VARCHAR2(100),
3.  event_date DATE,
4.  detail VARCHAR2(400));
```

Table created.

```
1. SQL>
2. CREATE OR REPLACE TRIGGER log_create_trigg
3.  AFTER CREATE ON SCHEMA
4.  BEGIN
5.  INSERT INTO log_table
6.  (user_name, event_date, detail)
7.  VALUES
8.  (USER, SYSDATE, 'created object is: ' || ora_dict_obj_name);
9.  END;
10. /
```

Trigger created.

```
1. SQL> select * from log_table;
```

No rows are selected.

```
1. SQL> create table abc as select * from dba_users;
```

Table created.

```
1. SQL> col user_name for a12
2. SQL> col detail for a25
3. SQL> select * from log_table;
```

USER_NAME EVENT_DAT DETAIL
---------- --------- ------------------------
MAHI 19-OCT-12 created object is: ABC

Database event trigger

These triggers fire, when a system activity occurs in the database like the login and logoff event triggers. They are useful for auditing the information of the system access. These triggers, allow you to track the system events and map them to the users.

Example

Below given trigger logs the logging information into log_trigger_table.

```
1. SQL> CREATE TABLE log_trigger_table (
2.  user_name VARCHAR2(30),
3.  event_date DATE,
4.  action VARCHAR2(300));
```

Table created.

```
1. SQL> CREATE OR REPLACE TRIGGER logon_trigger
2.  AFTER LOGON ON SCHEMA
3.  BEGIN
4.  INSERT INTO log_trigger_table
5.  (user_name, event_date, action )
6.  VALUES
7.  (USER, SYSDATE, 'Logging On');
8.  END;
9.  /
```

Trigger created.

SQL> exit
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit
Production With the Partitioning, OLAP, Data Mining and Real Application Testing options.

C:\Users\DELL\node1>sqlplus

SQL*Plus: Release 11.2.0.1.0 Production on Fri Oct 19 17:39:19 2012

Enter user-name: mahi
Enter password:

Connected to:

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production With the Partitioning, OLAP, Data Mining and Real Application Testing options:

```
1. SQL> select * from log_trigger_table;
```

USER_NAME EVENT_DAT ACTION
---------- --------- ----------
MAHI 19-OCT-12 Logging On

DML Trigger

These triggers fire, when you insert, update or delete the data from a table. You can fire them once for all the changes on a table or for each row change, using statement- or row-level trigger types, respectively. DML triggers are useful to control DML statements. You can use these triggers to audit, check, save and replace values before they are changed.

Example.1

Below given example inserts each record, which will be deleted from salary table into sal_deleted table.

```
1. SQL> select * from salary;
```

ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 20000 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 20000 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 20000 1500 1200 1800 22000
10005 Arunkumar.K Programmer 12000 1440 1320 1080 13800

```
1. SQL> create table sal_deleted(
2. rollno number(5), name varchar(15),
3. del_date date);
```

Table created.

SQL>

Now create the trigger.

SQL>ed sal_delete_trig

Create or replace trigger sal_delete before deleting on salary. For each row, begin:

```
1. insert into sal_deleted values
2. (:old.rollno, :old.empname,sysdate);
3. end;
4. /
5. SQL> @sal_delete_trig
```

Trigger created.

```
1. SQL> delete from salary where rollno = 10005;
```

1 row deleted.

```
1. SQL> select * from salary;
```

```
ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 20000 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 20000 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 20000 1500 1200 1800 22000

SQL>
```

```
1. select * from sal_deleted;
```

```
ROLLNO NAME DEL_DATE
---------- ---------- ---------
10005 Arunkumar.K 19-OCT-12
```

Example.2

The following trigger will insert the system time automatically into DOJ field, while inserting the records into student_details table.
SQL>

```
1. create table student_details
2.
3. 2 (rollno number(5), name varchar(15),
4. 3 dob date, doj date, dop date );
```

Table created.

```
1. SQL> ed student_details_trig;
```

```
2.
3.  create trigger student_details_trig before insert
4.  on student_details for each row
5.  begin
6.  :new.doj := sysdate;
7.  end;
8.  /
9.  SQL> @student_details_trig
```

Trigger created.

```
1.  SQL> select * from student_details;
```

No rows selected

```
1.  SQL> select sysdate from dual;
```

SYSDATE

19-OCT-12

```
1.  SQL> insert into student_details (rollno,name,dob) values (1001,'MAHESH','30-OCT-86');
```

1 row created.

```
1.  SQL> select * from student_details;
```

ROLLNO NAME DOB DOJ DOP
---------- --------------- --------- --------- ---------
1001 MAHESH 30-OCT-86 19-OCT-12
SQL>

Here, you can see DOJ is automatically inserted by the trigger.

Example.3

Following trigger will insert each record into salupdated table before the update happens in salary table,

```
1.  SQL> select * from salary;
```

ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
---------- --------------- --------------- ---------- ---------- ---------- ---------- ----------
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 20000 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 20000 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 20000 1500 1200 1800 22000

```
SQL> create table salupdated(
 rollno number(5),
 empname varchar(15),
 design varchar(15),
 bpay number(8,2),
 da number(6,2),
 total number(8,2),
 ta number(6,2));
```

Table created.

```
SQL> ed salupdate_trig
```

create or replace trigger salupdate_trig before update on salary for each row,

```
insert into salupdated values (:old.rollno, :old.empname, :old.design, :old.bpay, :old.da, :old.
netsal, :old.ta);
end;
/
SQL> @salupdate_trig
```

Trigger created.

```
SQL> select * from salupdated;
```

no rows selected

```
SQL> update salary set BPAY=21000 where DESIGN='Asst.Manager';
```

3 rows updated.

```
SQL> select * from salary;
```

ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
--- --- --- --- --- --- --- ---
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 21000 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 21000 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 21000 1500 1200 1800 22000

```
SQL> select * from salupdated;
```

ROLLNO EMPNAME DESIGN BPAY DA TOTAL TA
--- --- --- --- --- --- ---
10002 K.K.Omana Asst.Manager 20000 1500 22000 1200
10003 Anishkumar.K Asst.Manager 20000 1500 22000 1200
10004 Girishkumar.K Asst.Manager 20000 1500 22000 1200

SQL>

Example.4

Following DML trigger will raise an Application error, while trying to delete the records belonging to Asst.Manager.

```
1. SQL> select * from salary;
```

```
ROLLNO EMPNAME DESIGN BPAY DA TA PF NETSAL
---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
10001 S.Krishnan HOD 25000 1500 1200 2250 27000
10002 K.K.Omana Asst.Manager 19500 1500 1200 1800 22000
10003 Anishkumar.K Asst.Manager 19500 1500 1200 1800 22000
10004 Girishkumar.K Asst.Manager 19500 1500 1200 1800 22000
```

```
1.  SQL> CREATE or REPLACE TRIGGER not_del
2.   AFTER
3.   DELETE ON salary
4.   for each row
5.
6.   BEGIN
7.   IF :old.DESIGN = 'Asst.Manager' THEN
8.   raise_application_error(-20015, 'Not Delete this Row');
9.   END IF;
10.  END;
11.  /
```

```
Trigger created.
```

```
1.  SQL> delete from salary where rollno=10004;
2.  delete from salary where rollno=10004
3.  *
```

```
ERROR at line 1:
ORA-20015: Not Delete this Row
ORA-06512: at "MAHI.NOT_DEL", line 3
ORA-04088: error during execution of trigger 'MAHI.NOT_DEL
```

Result:

Trigger program executed successfully.

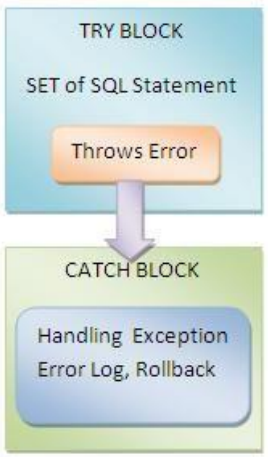
EXCEPTION HANDLING

Aim:

To construct database and apply Exception Handling in PL/SQL Program.

An error condition during a program execution is called an exception and the mechanism for resolving such an exception is known as an exception handler. SQL Server provides TRY, CATCH blocks for exception handling. We can put all T-SQL statements into a TRY BLOCK and the code for exception handling can be put into a CATCH block. We can also generate user-defined errors using a THROW block.

Syntax of Exception Handling



BEGIN TRY

/ T-SQL Statements */*

END TRY
BEGIN CATCH

- Print Error OR
- Rollback Transaction

END CATCH

In exception handling all T-SQL statements are put into a try block. If all statements execute without any error then everything is OK else control will go to the catch block.

PL/SQL - Exceptions

In this chapter, we will discuss Exceptions in PL/SQL. An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION**

block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** −

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
    exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
  c_id customers.id%type := 8;
  c_name customerS.Name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
```

```
    dbms_output.put_line('Error!');
END;
/
```

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION block**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
  my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
  c_id customers.id%type := &cc_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

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

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

Exception	Oracle	SQLCODE	Description

	Error		
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

System Defined Exception

In a System Defined Exception the exceptions (errors) are generated by the system.

Example

```
Declare @val1 int;
Declare @val2 int;
BEGIN TRY
Set @val1=8;
Set @val2=@val1/0; /* Error Occur Here */
END TRY
BEGIN CATCH
Print 'Error Occur that is:'
Print Error_Message()
END CATCH
```

Output

Error Occur that is:
Divide by zero error encountered

User Defined Exception

This type of exception is user generated, not system generated.

```
Declare @val1 int;
Declare @val2 int;
BEGIN TRY
Set @val1=8;
Set @val2=@val1%2;
if @val1=1
Print ' Error Not Occur'
else
Begin
Print 'Error Occur';
Throw 60000,'Number Is Even',5
End

END TRY
BEGIN CATCH
Print 'Error Occur that is:'
Print Error_Message()
END CATCH
```

Output

Error Occur
Error Occur that is:
Number Is Even

Here 60000 denotes the error number and 5 denotes the state to associate with the message.

The following are system functions and the keyword used within a catch block:

1. @@ERROR
2. ERROR_NUMBER()
3. ERROR_STATE()
4. ERROR_LINE()
5. ERROR_MESSAGE()
6. ERROR_PROCEDURE()
7. ERROR_SEVERITY()
8. RAISERROR()
9. GOTO()

Now we will see some examples to help understand all these functions and keywords.

First create a table and enter some value into the table as in the following:

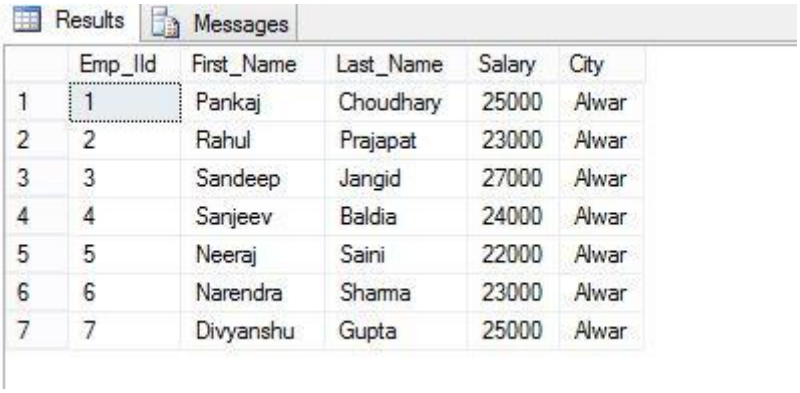
```
Create TABLE Employee
(
Emp_IId Int identity(1,1),
First_Name Nvarchar(MAX) Not NUll,
Last_Name Nvarchar(MAX) Not Null,
Salary Int Not Null check(Salary>20000),
City Nvarchar(Max) Not Null
)
```

Insert data into Employee.

```
Select 'Pankaj','Choudhary',25000,'Alwar' Union All
Select 'Rahul','Prajapat',23000,'Alwar' Union All
Select 'Sandeep','Jangid',27000,'Alwar' Union All
Select 'Sanjeev','Baldia',24000,'Alwar' Union All
Select 'Neeraj','Saini',22000,'Alwar' Union All
Select 'Narendra','Sharma',23000,'Alwar' Union All
Select 'Divyanshu','Gupta',25000,'Alwar'
```

Now execute a select command.

```
Select * From Employee
```



Results | Messages

	Emp_IId	First_Name	Last_Name	Salary	City
1	1	Pankaj	Choudhary	25000	Alwar
2	2	Rahul	Prajapat	23000	Alwar
3	3	Sandeep	Jangid	27000	Alwar
4	4	Sanjeev	Baldia	24000	Alwar
5	5	Neeraj	Saini	22000	Alwar
6	6	Narendra	Sharma	23000	Alwar
7	7	Divyanshu	Gupta	25000	Alwar

Example 1: (@@ERROR)

@@ERROR return the error number for last executed T-SQL statements. It returns 0 if the previous Transact-SQL statement encountered no errors else return an error number.

```
1. Update Employee set Salary=19000 Where Emp_IID=5
2. IF @@ERROR = 547
3. PRINT 'A check constraint violation occurred.';
```

Output:

Msg 547, Level 16, State 0, Line 1

The UPDATE statement conflicted with the CHECK constraint "CK__Employee__Salary__68487DD7". The conflict occurred in database "Home_Management", table "dbo.Employee", column 'Salary'.

The statement has been terminated.

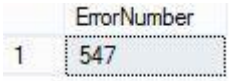
A check constraint violation occurred.

Example 2 (ERROR_NUMBER)

ERROR_NUMBER() returns the error number that caused the error. It returns zero if called outside the catch block.

```
1. BEGIN TRY
2.
3. Update Employee set Salary=19000 Where Emp_IID=5
4. END TRY
5. BEGIN CATCH
6. SELECT ERROR_NUMBER() AS ErrorNumber;
7. END CATCH;
8. GO
```

Output



	ErrorNumber
1	547

Now a question develops of what is diff @@ERROR and ERROR_NUMBER. Let me explain.

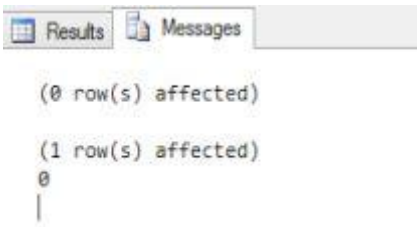
1. ERROR_NUMBER can only be used in a catch block, outside a catch block it returns Null but @@ERROR can be used inside or outside the catch block (see Example 1).

2. ERROR_NUMBER is a contrast to @@ERROR, that only returns the error number in the statement immediately after the one that causes an error, or the first statement of a CATCH block.

Now we will see an example and observe the differences between them.

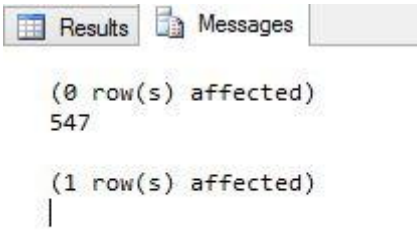
```
1. BEGIN TRY
2.
3. Update Employee set Salary=19000 Where Emp_IID=5
4. END TRY
5. BEGIN CATCH
6.
7. SELECT ERROR_NUMBER() AS ErrorNumber;
8. print @@ERROR
9. END CATCH;
10. GO
```

Output




```
1. BEGIN TRY
2.
3. Update Employee set Salary=19000 Where Emp_IID=5
4. END TRY
5. BEGIN CATCH
6. print @@ERROR
7. SELECT ERROR_NUMBER() AS ErrorNumber;
8.
9. END CATCH;
10. GO
```

Output



Example 3 (ERROR_MESSAGE)

ERROR_MESSAGE returns the message text of the error that caused the error. The return type of ERROR_MESSAGE is **nvarchar(4000)**.

```
1. BEGIN TRY
2.
3. Update Employee set Salary=19000 Where Emp_IID=5
4. END TRY
5. BEGIN CATCH
6. SELECT ERROR_MESSAGE() AS ErrorMsg;
7.
8. END CATCH;
9. GO
```

Output

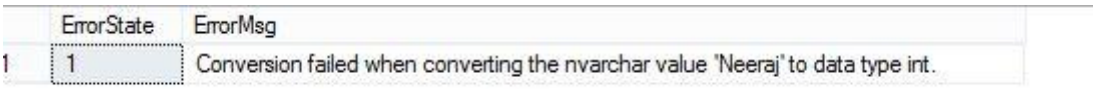
The UPDATE statement conflicted with the CHECK constraint "CK__Employee__Salary__68487DD7". The conflict occurred in database "Home_Management", table "dbo.Employee", column 'Salary'.

Example 4 (ERROR_STATE)

ERROR_STATE returns the state number of the error. The return type of ERROR_STATE is **INT**.

```
1. BEGIN TRY
2.
3. SELECT SALARY + First_Name From Employee Where Emp_IID=5
4. END TRY
5. BEGIN CATCH
6. SELECT ERROR_STATE() AS ErrorState , ERROR_MESSAGE() ErrorMsg ;
7. END CATCH;
8. GO
```

Output



	ErrorState	ErrorMsg
1	1	Conversion failed when converting the nvarchar value 'Neeraj' to data type int.

Example 5 (ERROR_LINE)

ERROR_LINE returns the line number at which an error occurred. The return type of ERROR_LINE is **INT**.

```
1. BEGIN TRY
2. SELECT SALARY + First_Name From Employee Where Emp_IID=5
3. END TRY
4. BEGIN CATCH
5. SELECT ERROR_STATE() AS ErrorLine;
6. END CATCH;
7. GO
```

Output

	ErrorLine
1	1

Example 6 (ERROR_PROCEDURE)

ERROR_PROCEDURE returns the name of the Stored Procedure or trigger of where an error occurred. The return type of ERROR_PROCEDURE is **nvarchar(128)**.

Return value

Return value returns the Stored Procedure Name if an error occurs in a Stored Procedure or trigger and the catch block is called.

It returns **NULL** if the error did not occur within a Stored Procedure or trigger or it isb called outside the scope of a CATCH block.

First we create a Stored Procedure.

```
1. Create Procedure My_Proc
2. AS
3. begin
4. BEGIN TRY
5. SELECT SALARY + First_Name From Employee Where Emp_IID=5
6. END TRY
7. BEGIN CATCH
8. SELECT ERROR_PROCEDURE() AS ProcName;
9. END CATCH;
10. END
```

Now execute this Stored Procedure.

```
1. Exec My_Proc
```

Output

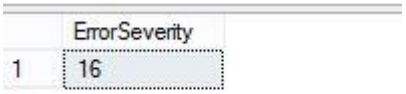
	ProcName
1	My_Proc

Example 7 (ERROR_SEVERITY)

ERROR_SEVERITY returns the severity of the error. The return type of ERROR_SEVERITY is **INT**.

```
1. BEGIN TRY
2. SELECT SALARY + First_Name From Employee Where Emp_IID=5
3. END TRY
4. BEGIN CATCH
5. SELECT ERROR_SEVERITY() AS ErrorSeverity;
6. END CATCH;
```

Output



	ErrorSeverity
1	16

The severity level of an error message provides an indication of the type of problem that Microsoft® SQL Server encountered. In the preceding example the Severity Level is 16. That means that the error can be removed by the user.

Some important severity levels are:

13	Indicates transaction deadlock errors.
14	Indicates security-related errors, such as permission denied.
15	Indicates syntax errors in the Transact-SQL command.
16	Indicates general errors that can be corrected by the user.

Example 8 (RAISERROR)

RAISEERROR is used to generate an error message and initiates error processing for the session.

```
1. BEGIN TRY
2. SELECT SALARY + First_Name From Employee Where Emp_IID=5
3. END TRY
```

```
4. BEGIN CATCH
5. RAISERROR(N'An Error Is Occur',16,3);
6. END CATCH;
```

Output

```
(0 row(s) affected)
Msg 50000, Level 16, State 3, Line 8
An Error Is Occur
```

In RAISERROR(N'An Error Is Occur',16,3) the first argument represents the error messagethe , second argument represents the Severity Level and the last argument represents the Error State.

Example 9 (GOTO)

GOTO causes a jump to a specific step or statement. It alters the flow of execution to a label. We declare some labels in batch and alter we can move at a specific label. GOTO can exist within a conditional control-of-flow statements, statement blocks, or procedures, but it cannot go to a label outside the batch. GOTO cannot be used to jump into a TRY or CATCH scope.

```
1. Declare @Var Int;
2. Set @Var=1
3. Print 'Goto exercise'
4. If @Var%2=0
5. GOTO Label1;
6. else
7. GOTO Label2;
8. Set @Var=@Var+1;
9. Label1:
10. Print 'Var Is Odd'
11. Label2:
12. Print 'Var Is Even'
```

Output

Goto exercise
Var Is Even

Example 10

```
1. BEGIN TRY
2. SELECT SALARY + First_Name From Employee Where Emp_IID=5
3. END TRY
4. BEGIN CATCH
5. SELECT ERROR_STATE() AS Error_Stat,ERROR_SEVERITY() AS ErrorSeverity, ERRO
   R_LINE() as ErrorLine, ERROR_NUMBER() AS ErrorNumber, ERROR_MESSAGE() AS
   ErrorMsg;
6. END CATCH;
```

Output

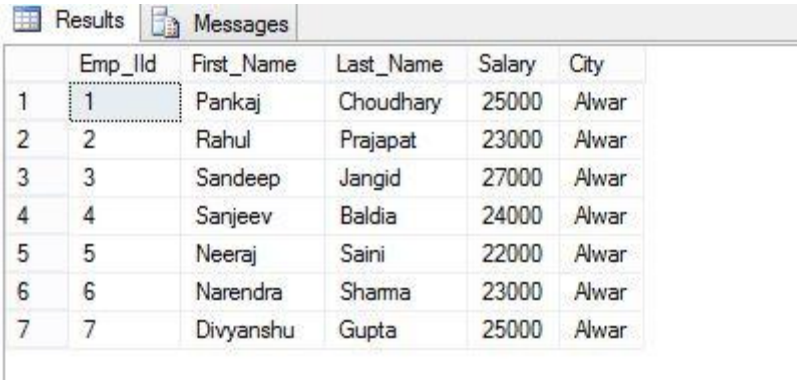
	Error_Stat	ErrorSeverity	ErrorLine	ErrorNumber	ErrorMsg
1	1	16	2	245	Conversion failed when converting the nvarchar v...

Exercise 11 (Transaction Management)

Exception handling is mainly used for Transaction Management. Let us see an example.

```
1.  Begin Transaction Trans
2.    Begin Try
3.      Delete From Employee Where Employee.Emp_IID<3
4.      Update Employee Set Employee.First_Name='Pankaj kumar' Where Employee.Emp_IID
    ='6th' /* Error Will Occur Here */
5.      If @@TranCount>0
6.      begin Commit Transaction Trans
7.    End
8.  End Try
9.  Begin Catch
10.   if @@TranCount>0
11.   Print 'Error Is Occur in Transaction'
12.   begin Rollback Transaction Trans
13. End
14. End Catch
15.
16. Select * From Employee
```

Output



Results | Messages

	Emp_IId	First_Name	Last_Name	Salary	City
1	1	Pankaj	Choudhary	25000	Alwar
2	2	Rahul	Prajapat	23000	Alwar
3	3	Sandeep	Jangid	27000	Alwar
4	4	Sanjeev	Baldia	24000	Alwar
5	5	Neeraj	Saini	22000	Alwar
6	6	Narendra	Sharma	23000	Alwar
7	7	Divyanshu	Gupta	25000	Alwar

When to use Exception Handling:

1. In Transaction Management to Rollback the transaction.
2. While using cursors in SQL Server.

3. When implementing a DML Query (insert, update or delete) for for an checking the error and to handle it.

Result:

The exception handling program is executed successfully.

DATABASE DESIGN USING ER MODELING, NORMALIZATION AND IMPLEMENTATION FOR ANY APPLICATION

What is ER Modeling?

Entity Relationship Modeling (ER Modeling) is a graphical approach to database design. It uses Entity/Relationship to represent real world objects.

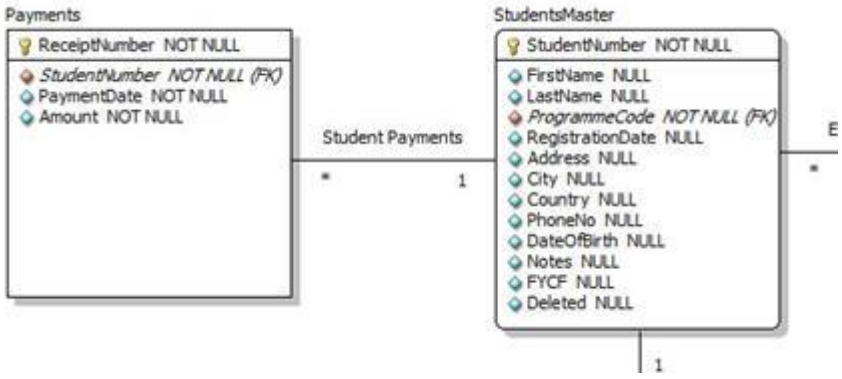
An **Entity** is a thing or object in real world that is distinguishable from surrounding environment. For example each employee of an organization is a separate entity. Following are some of major characteristics of entities.

- An entity has a set of properties.
- Entity properties can have values.

Enhanced Entity Relationship (EER) Model

Enhanced Entity Relationship (EER) Model is a high level data model which provides extensions to original **Entity Relationship** (ER) model. EER Models supports more details design. EER Modeling emerged as a solution for modeling highly complex databases.

EER uses UML notation. UML is the acronym for Unified Modeling Language; it is a general purpose modeling language used when designing object oriented systems. Entities are represented as class diagrams. Relationships are represented as associations between entities. The diagram shown below illustrates an ER diagram using the UML notation.



Why use ER Model?

Now you may think why use ER modeling when we can simply create the database and all of its objects without ER modeling? One of the challenges faced when designing database is the fact that designers, developers and end-users tend to view data and its usage differently. If this situation is left unchecked, we can end up producing a database system that does not meet the requirements of the users.

Communication tools understood by all stakeholders(technical as well non-technical users) are critical in producing database systems that meet the requirements of the users. ER models are examples of such tools.

ER diagrams also increase user productivity as they can be easily translated into relational tables.

Case Study: ER diagram for "MyFlix" Video Library

Let's now work with the MyFlix Video Library database system to help understand the concept of ER diagrams. We will using this database for all hand-on in the remainder of this tutorials

MyFlix is a business entity that rents out movies to its members. MyFlix has been storing its records manually. The management now wants to move to a DBMS

Let's look at the steps to develop EER diagram for this database-

1. Identify the entities and determine the relationships that exist among them.
2. Each entity, attribute and relationship, should have appropriate names that can be easily understood by the non-technical people as well.
3. Relationships should not be connected directly to eachother. Relationships should connect entities.
4. Each attribute in a given entity should have a unique name.

Entities in the "MyFlix" library

The entities to be included in our ER diagram are;

- **Members** - this entity will hold member information.
- **Movies** - this entity will hold information regarding movies
- **Categories** - this entity will hold information that places movies into different categories such as "Drama", "Action", and "Epic" etc.
- **Movie Rentals** - this entity will hold information that about movies rented out to members.
- **Payments** - this entity will hold information about the payments made by members.

Defining the relationships among entities

Members and movies

The following holds true regarding the interactions between the two entities.

- A member can rent a more than movie in a given period.
- A movie can be rented by more than one member in a given period.

From the above scenario, we can see that the nature of the relationship is many-to-many. **Relational databases do not support many-to-many relationships. We need to introduce a junction entity**. This is the role that the MovieRentals entity plays. It has a one-to-many relationship with the members table and another one-to-many relationship with movies table.

Movies and categories entities

The following holds true about movies and categories.

- A movie can only belong to one category but a category can have more than one movie.

We can deduce from this that the nature of the relation between categories and movies table is one-to-many.

Members and payments entities

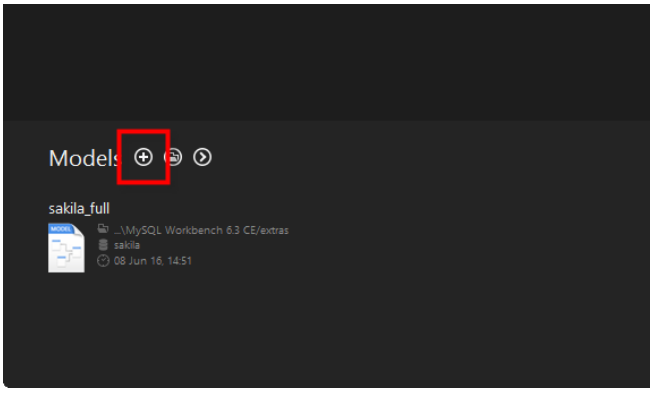
The following holds true about members and payments

- A member can only have one account but can make a number of payments.

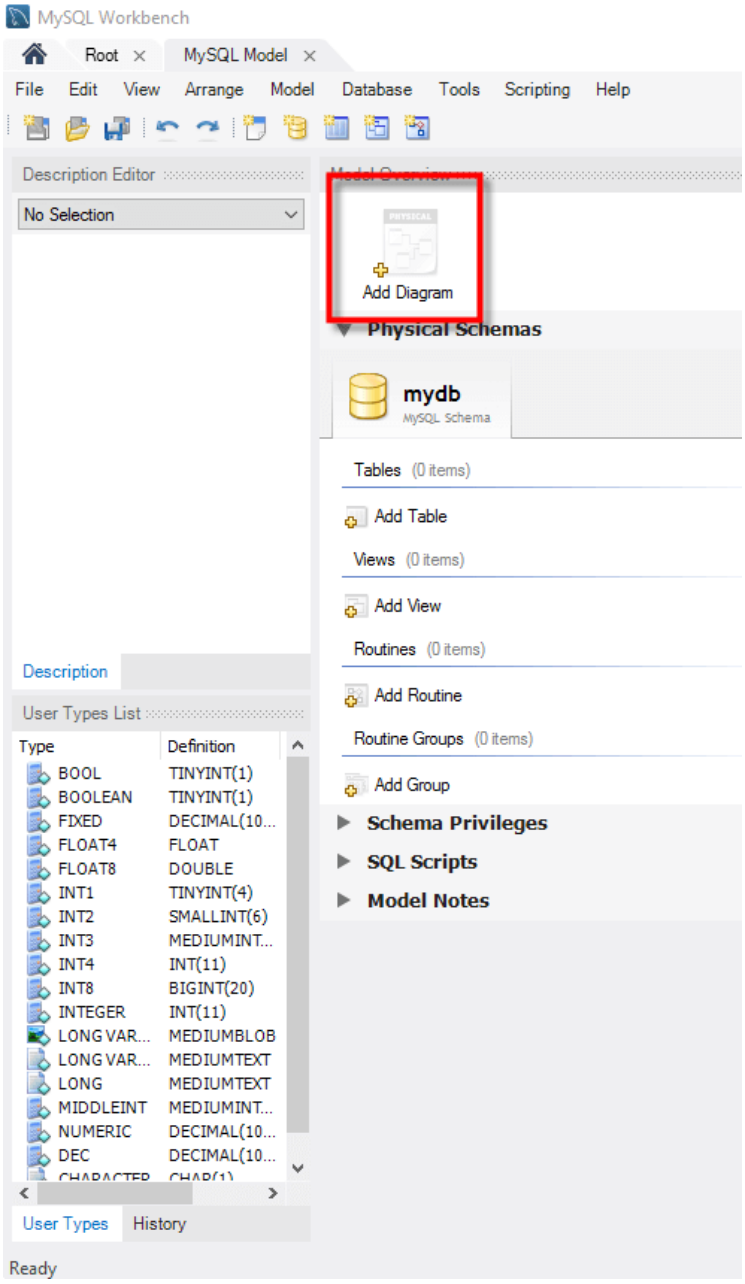
We can deduce from this that the nature of the relationship between members and payments entities is one-to-many.

Now lets create EER model using MySQL Workbench

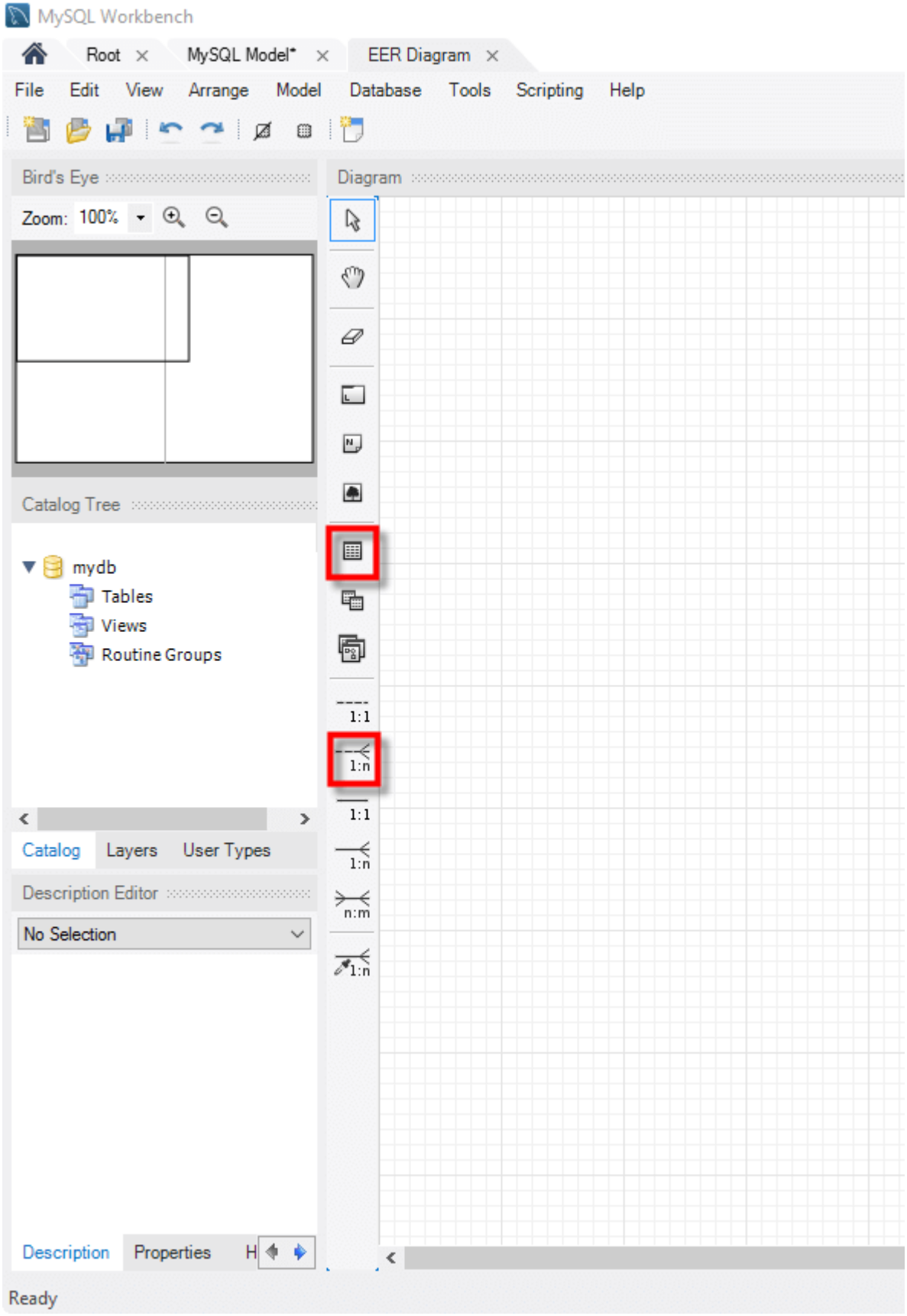
In the MySQL workbench , Click - "+" Button



Double click on Add Diagram button to open the workspace for ER diagrams.


MySQL Workbench
Root
MySQL Model
File Edit View Arrange Model Database Tools Scripting Help
Description Editor
No Selection
Add Diagram
Physical Schemas
mydb
MySQL Schema
Tables (0 items)
Add Table
Views (0 items)
Add View
Routines (0 items)
Add Routine
Routine Groups (0 items)
Add Group
Schema Privileges
SQL Scripts
Model Notes
Description
User Types List
Type
Definition
BOOL TINYINT(1)
BOOLEAN TINYINT(1)
FIXED DECIMAL(10...
FLOAT4 FLOAT
FLOAT8 DOUBLE
INT1 TINYINT(4)
INT2 SMALLINT(6)
INT3 MEDIUMINT...
INT4 INT(11)
INT8 BIGINT(20)
INTEGER INT(11)
LONG VAR... MEDIUMBLOB
LONG VAR... MEDIUMTEXT
LONG MEDIUMTEXT
MIDDLEINT MEDIUMINT...
NUMERIC DECIMAL(10...
DEC DECIMAL(10...
User Types
History
Ready

Following window appears



Let's look at the two objects that we will work with.

- The table object allows us to create entities and define the attributes associated with the particular entity.
- The place relationship button allows us to define relationships between entities.

The **members'** entity will have the following attributes

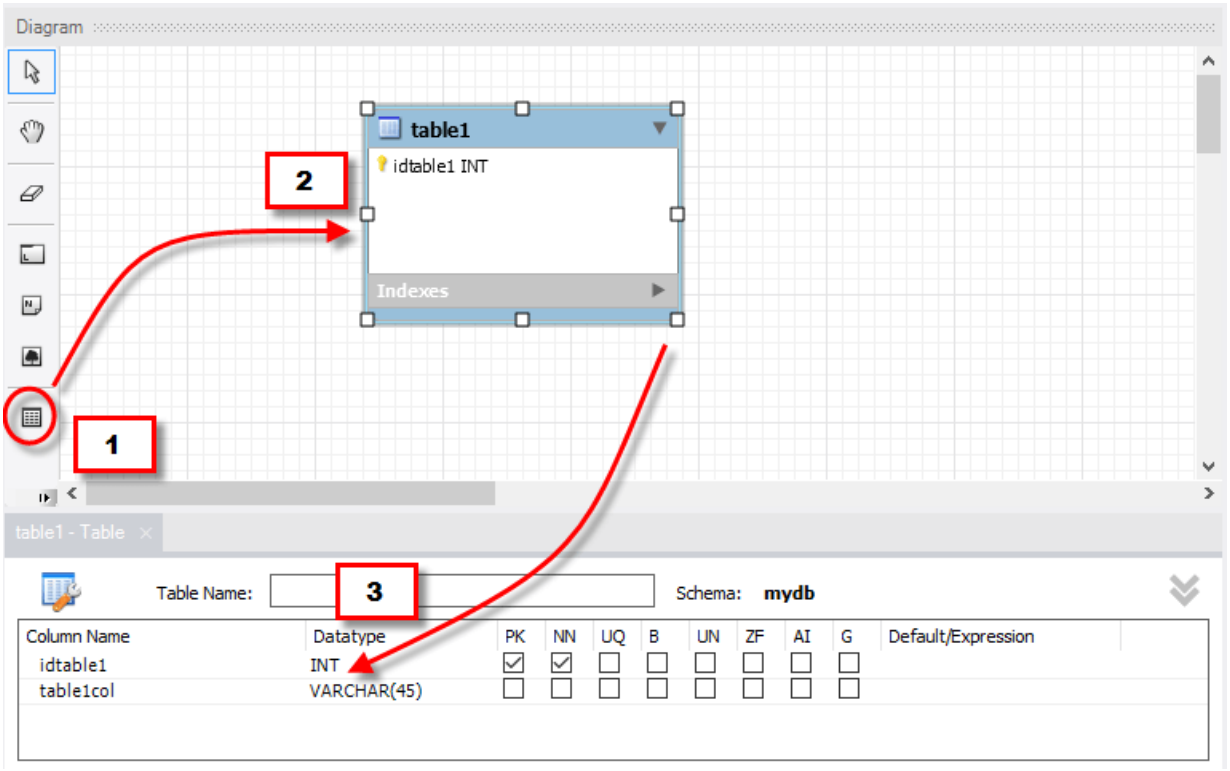
- Membership number
- Full names
- Gender
- Date of birth
- Physical address
- Postal address

Let's now create the members table

1.Drag the table object from the tools panel

2.Drop it in the workspace area. An entity named table 1 appears

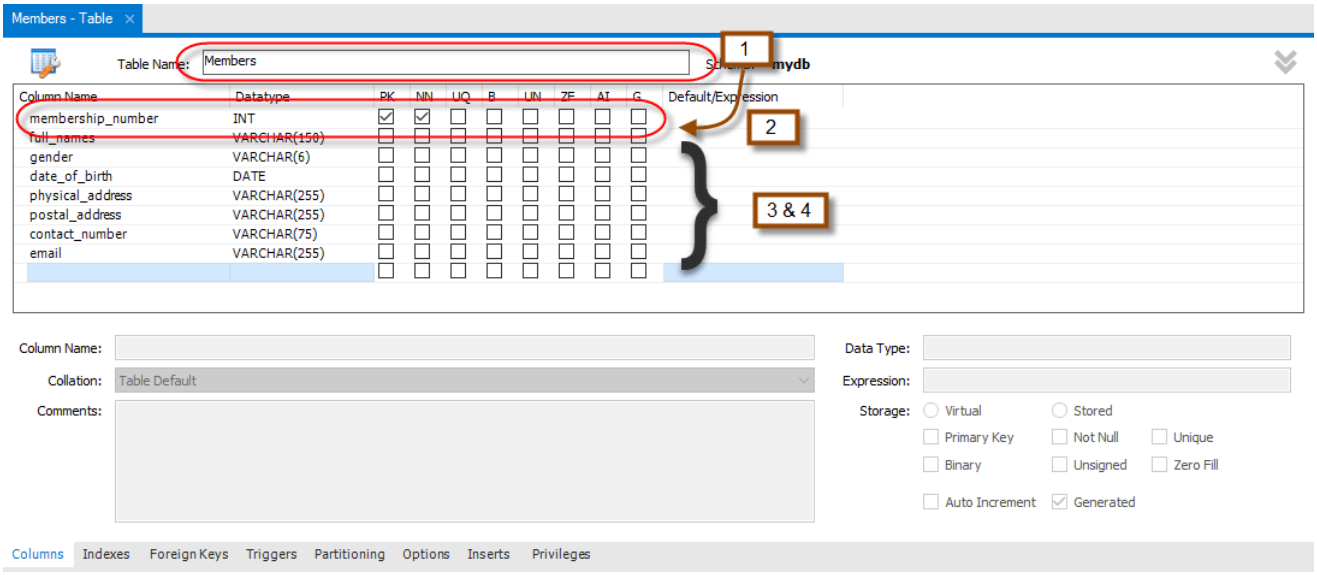
3.Double click on it. The properties window shown below appears



Next ,

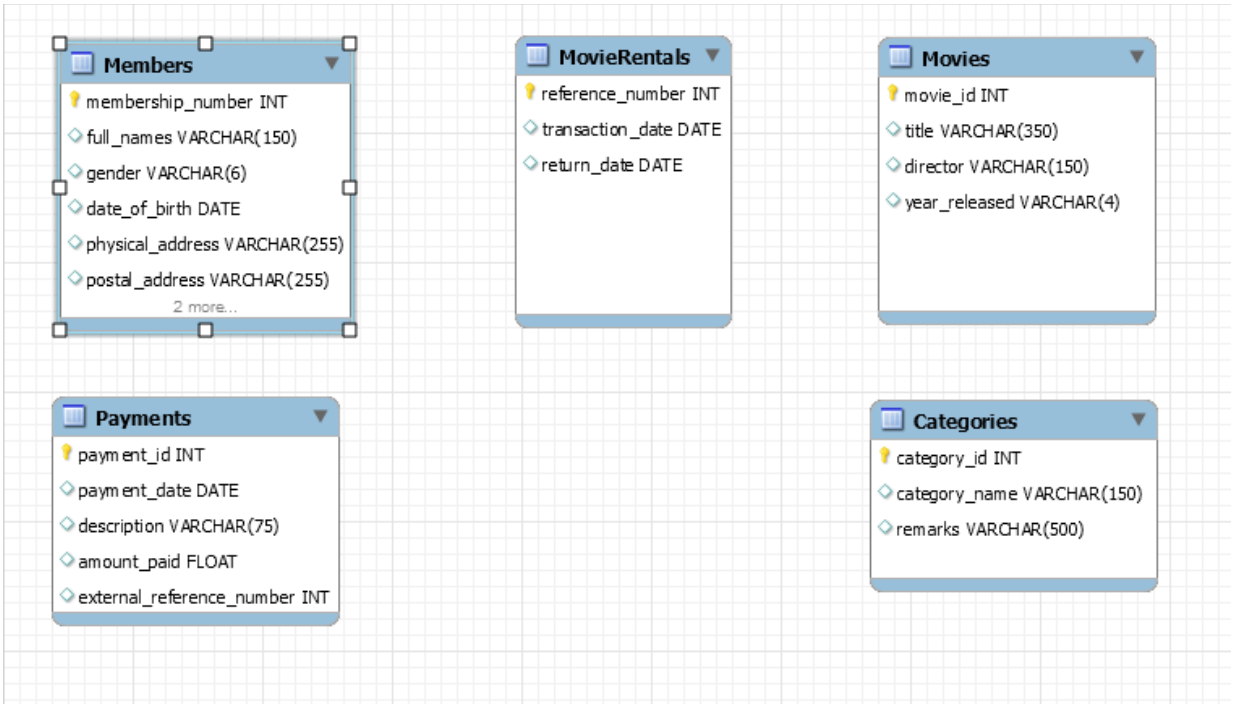
1. Change table 1 to Members
2. Edit the default idtable1 to membership_number
3. Click on the next line to add the next field
4. Do the same for all the attributes identified in members' entity.

Your properties window should now look like this.



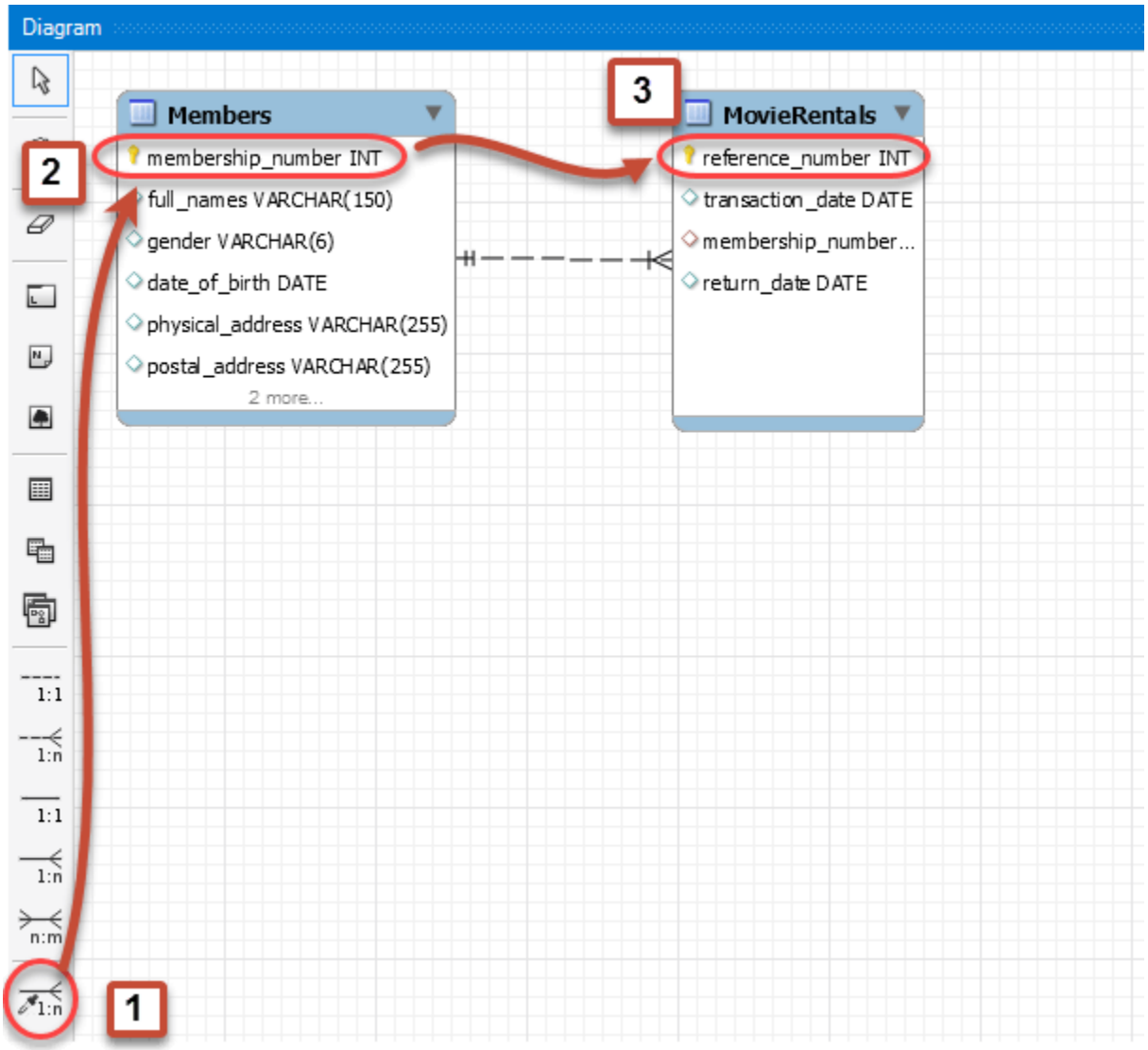
Repeat the above steps for all the identified entities.

Your diagram workspace should now look like the one shown below.

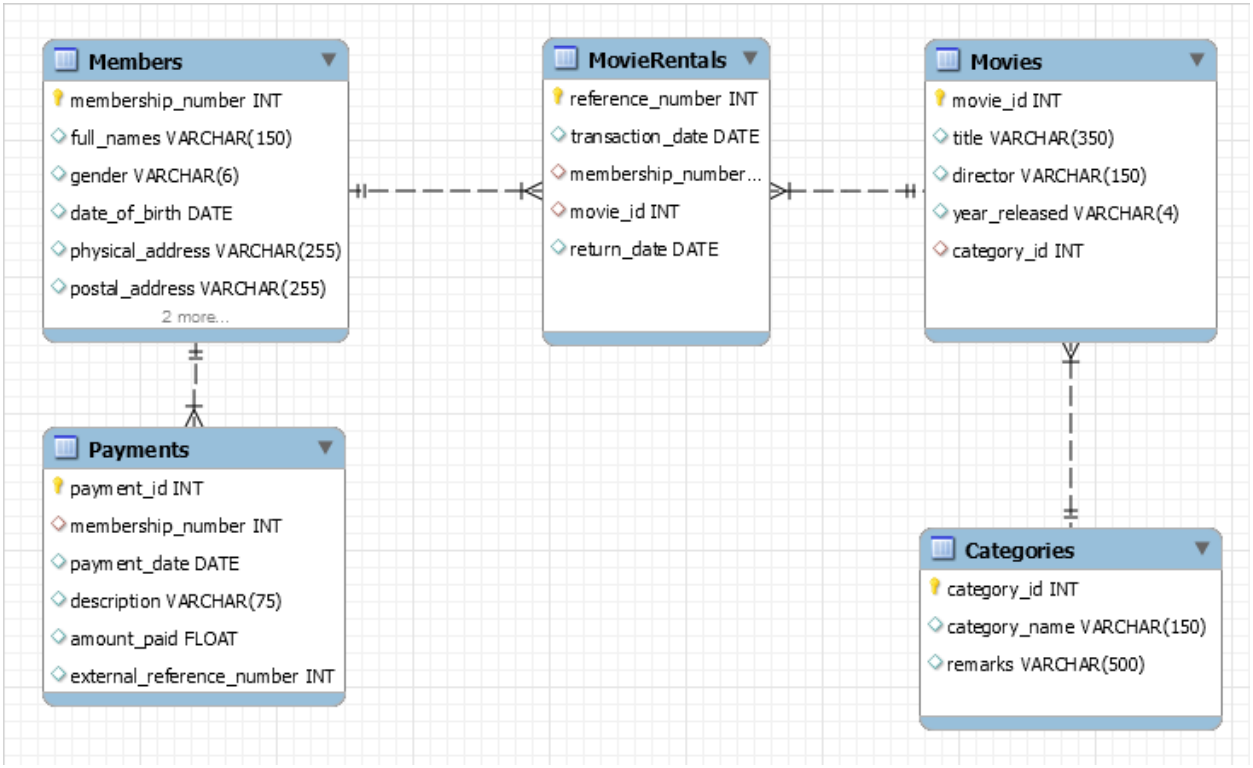


Lets create relationship between Members and Movie Rentals

1. Select the place relationship using existing columns too
2. Click on membership_number in the Members table
3. Click on reference_number in the MovieRentals table



Repeat above steps for other relationships. Your ER diagram should now look like this -



Summary

- ER Diagrams play a very important role in the database designing process. They serve as a non-technical communication tool for technical and non-technical people.
- Entities represent real world things; they can be conceptual as a sales order or physical such as a customer.
- All entities must be given unique names.
- ER models also allow the database designers to identify and define the relations that exist among entities.

What is Normalization? 1NF, 2NF, 3NF & BCNF with Examples

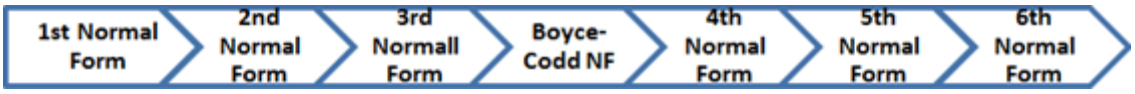
What is Normalization?

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

It divides larger tables to smaller tables and links them using relationships.

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined with Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

Theory of Data Normalization in SQL is still being developed further. For example, there are discussions even on 6th Normal Form. **However, in most practical applications, normalization achieves its best in 3rd Normal Form**. The evolution of Normalization theories is illustrated below-



Database Normalization Examples -

Assume a video library maintains a database of movies rented out. Without any normalization, all information is stored in one table as shown below.

Full Names	Physical Address	Movies rented	Salutation	Category
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.	Action, Action
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.	Romance, Romance
Robert Phil	5th Avenue	Clash of the Titans	Mr.	Action

Table 1

Here you see **Movies Rented column has multiple values**.

Database Normal Forms

Now let's move into 1st Normal Forms

1NF (First Normal Form) Rules

- Each table cell should contain a single value.
- Each record needs to be unique.

The above table in 1NF-

1NF Example

Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Table 1: In 1NF Form

Before we proceed let's understand a few things --

What is a KEY?

A KEY is a value used to identify a record in a table uniquely. A KEY could be a single column or combination of multiple columns

Note: Columns in a table that are NOT used to identify a record uniquely are called non-key columns.

What is a Primary Key?

A primary is a single column value used to identify a database record uniquely.

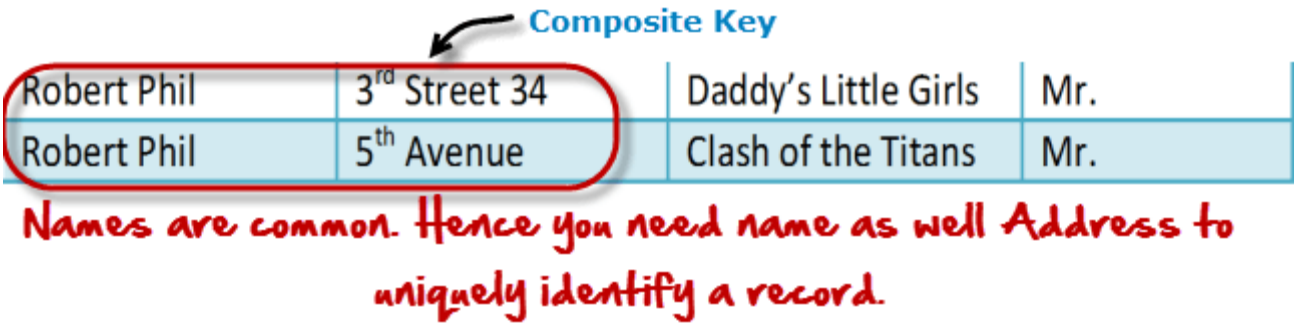
It has following attributes

- A primary key cannot be NULL
- A primary key value must be unique
- The primary key values cannot be changed
- The primary key must be given a value when a new record is inserted.

What is Composite Key?

A composite key is a primary key composed of multiple columns used to identify a record uniquely

In our database, we have two people with the same name Robert Phil, but they live in different places.



Hence, we require both Full Name and Address to identify a record uniquely. That is a composite key.

Let's move into second normal form 2NF

2NF (Second Normal Form) Rules

- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Table 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

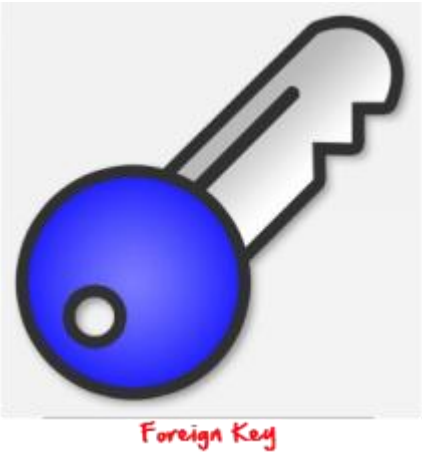
We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.

We have introduced a new column called Membership_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id

Database - Foreign Key

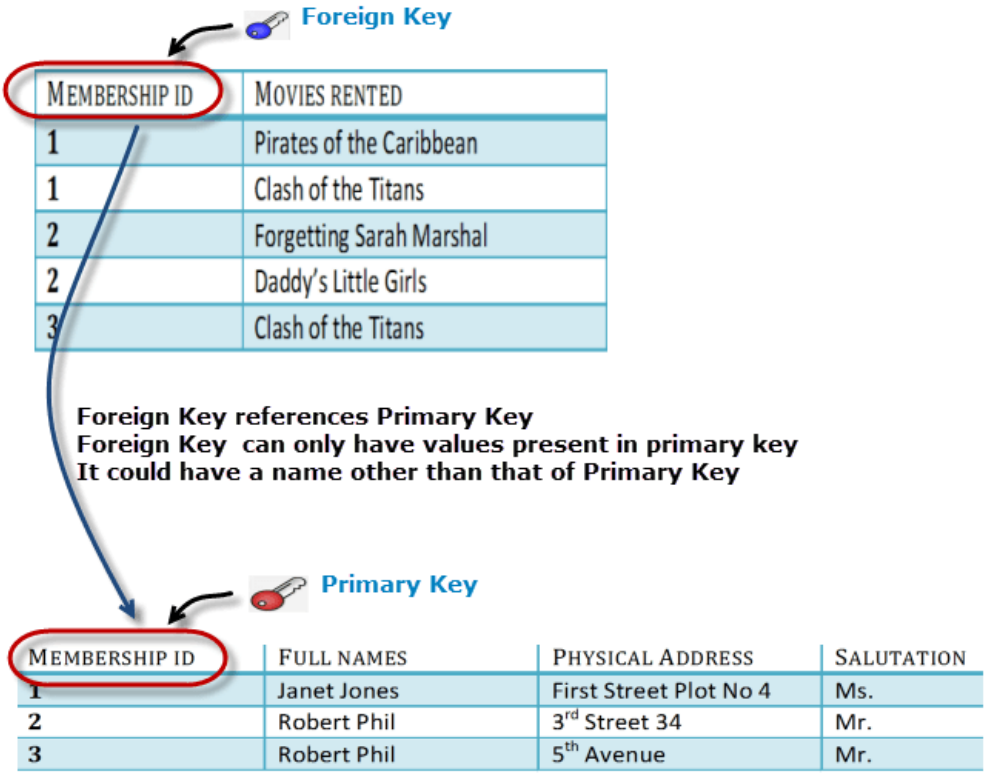
In Table 2, Membership_ID is the Foreign Key

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans



Foreign Key references the primary key of another Table! It helps connect your Tables

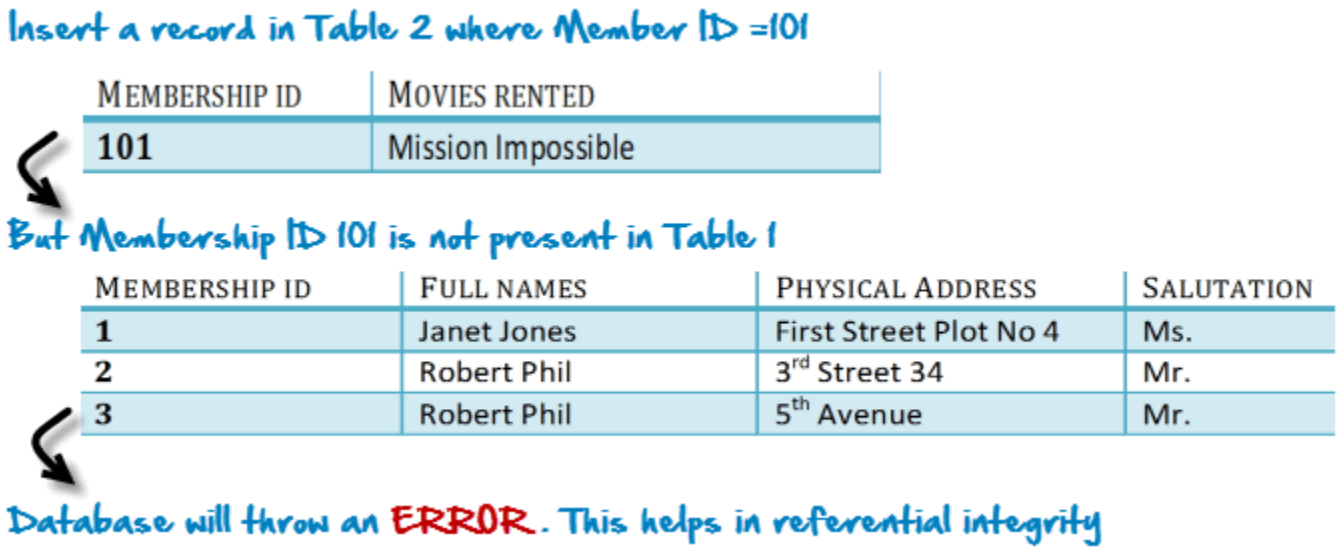
- A foreign key can have a different name from its primary key
- It ensures rows in one table have corresponding rows in another
- Unlike the Primary key, they do not have to be unique. Most often they aren't
- Foreign keys can be null even though primary keys can not



Why do you need a foreign key?

Suppose an idiot inserts a record in Table B such as

You will only be able to insert values into your foreign key that exist in the unique key in the parent table. This helps in referential integrity.



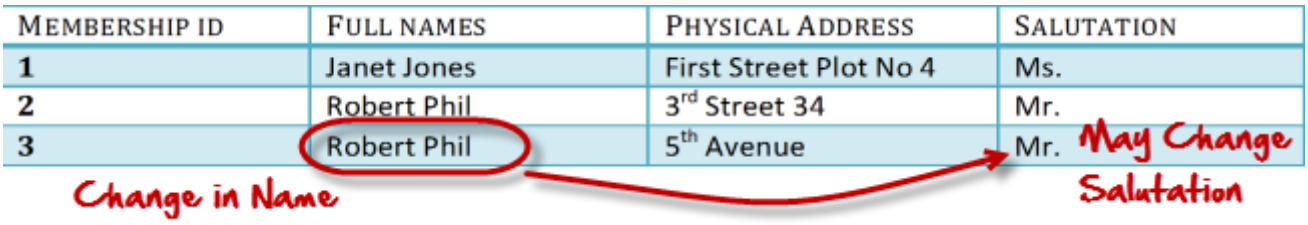
The above problem can be overcome by declaring membership id from Table2 as foreign key of membership id from Table1

Now, if somebody tries to insert a value in the membership id field that does not exist in the parent table, an error will be shown!

What are transitive functional dependencies?

A transitive functional dependency is when changing a non-key column, might cause any of the other non-key columns to change

Consider the table 1. Changing the non-key column Full Name may change Salutation.



Let's move into 3NF

3NF (Third Normal Form) Rules

- Rule 1- Be in 2NF
- Rule 2- Has no transitive functional dependencies

To move our 2NF table into 3NF, we again need to again divide our table.

3NF Example

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

TABLE 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

SALUTATION ID	SALUTATION
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

Table 3

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in Table 3

Now our little example is at a level that cannot further be decomposed to attain higher forms of normalization. In fact, it is already in higher normalization forms. Separate efforts for moving into next levels of normalizing data are normally needed in complex databases. However, we will be discussing next levels of normalizations in brief in the following.

Boyce-Codd Normal Form (BCNF)

Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has more than one **Candidate** Key.

Sometimes is BCNF is also referred as **3.5 Normal Form.**

4NF (Fourth Normal Form) Rules

If no database table instance contains two or more, independent and multivalued data describing the relevant entity, then it is in 4th Normal Form.

5NF (Fifth Normal Form) Rules

A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed into any number of smaller tables without loss of data.

6NF (Sixth Normal Form) Proposed

6th Normal Form is not standardized, yet however, it is being discussed by database experts for some time. Hopefully, we would have a clear & standardized definition for 6th Normal Form in the near future...

Project Title:-

Database system for Library Management system

Project Overview Statement

As You Know that a Library is collection of books in any institute .Librarian resposibilty is to manage all the records of books issued and also returned on Manualy.

Case study

Current system:

All the Transaction(books issues & books returned) are manualy recorded(registars.)

Students search books by racks it so time consuming

And there is no arrangement.

Also threat of losing recorde.

Project Aim and Objective

The project aim and objective are:

To eliminate the paper –work in library

-to record every transaction in computerized system so that problem such as record file missing won't happen again

Backgroud of Project

Library Management system is an application refer to other library system and is suitable to use by small and medium size libray .

It is use by librarian and libray admin to manage the library using a computerized system.

The system was designed to help librain record every book transcation so that the problem such as file missing will not happened again**.**

Design view

The library has the following tables in its database;

1. Books (book_id, ISBN,bookName, BOOKAUTHOR and bookedition)

2. student (student_id, studentname, student_email, student_address)

3. Staff (staff_id, staff_name, staff_address, staff_gender, staff_phone)

4. department (department_id, branch_name)

5. Issue (issue_id, issue_date, expiry_date, book_name, book_id)

6. Return (return_id, expiry_date, isuue_date, book_id)

NORAMALIZATION OF TABLE

Why Normalization:

Database normalization is the process of removing redundant data from your tables in order to improve storage efficiency, data integrity, and scalability.
Normalization generally involves splitting existing tables into multiple ones, which must be re-joined or linked each time a query is issued.

Given table is converted to its 1NF as follows.

• STEP NUMBER 1:

elimination of duplicative columns from table 1.

• Step number 2:

create separate table for each group of related data and identify each row with unique column (primary key).

2nd normal form

A table is in first normal form and each non-key field is functionally dependent upon primary key.
Now we'll take the table above and design new tables that will eliminate the repeted date in non key _field

To decide what fields belong together in a table, think about which field determines the
values in other fields.

Create a table for those fields and enter the sample data.

- Think about what the primary key for each table would be and about the relationship between the tables.
- Mark the primary key for each table and make sure that you do not have repeated data in non-key fields.

- Third normal form (3NF) requires that there are no functional dependencies of non-key attributes on something other than a candidate key.
- A table is in 3NF if all of the non primary-key attributes are mutually independent
- There should not be transitive depeinces.

Normalization of Tables in Database

ISSue_id	Book_id	Student_id
1122	110,120,320	bitE183

In the **ISSUE Table** there is repating book_id . A student has issued 3 books.

After first Normalization

ISSUE_ID	book_id	Student_id
1122	**110**	**bitE183**
1122	**320**	**bitE183**
1122	**120**	**bitE183**

Second normalized Form:

In the following Student relation all attributes are

dependent on the primary key StudID

Student_id Name Depid Issue_date Expairy_date Phone

BITf13E183	**Azhar**	**20**	**17-6-15**	**1-7-15**		**3127400**

We can create two other relations from Student Table one is Department fields are fully dependent on the primary keys DEp_id

DEp_id	Dep_name
11	**CS & IT Department**
22	**Education Department**
33	**Economics Department**
44	**Laaw Department**

Student_id	Name	Issue_date	Expairy_date	Phone
BITf13E183	Azhar	15-6-15	1-7-15	312-7400558

Before third normal form

Staff_id	Name	Gender	Designation	Address	City	state	cell
1101	Shaid	M	Librarian	House no 12 street 6	Sargodha	punjab	300-1234567
1345	Riaz	m	Data entery	Statilete town	Sargodha	Punjab	0346-1234567
2264	Arshad	m	Naib qaisd	Raza garden	Sargodha	Punjab	0333-1234567

After 3rd Normalization

Staff table

Staff_id **Name** **Gender**

Staff conatact

Staff_id	Address	City	State	Telephone	cell

STUDENT Table before Third normalized Form :

Std_id **Name** **Gender** **Address** **City** **State** **Phone** **Dep_id**

After third normal

Student_id	Dep_id	Department
IT-113	**C-26**	**Cs & IT**
Lm-456	**L-11**	**Law**
Eng-98	**E-41**	**ENGLISH**

Studentcontact table:

Student_id	Address	City	State	Phone
IT-111	Statlite twon	Sargodha	Punjab	312-1234567
Cs-786	Sahiwal	sargoda	punjab	300-1234567

Student table:

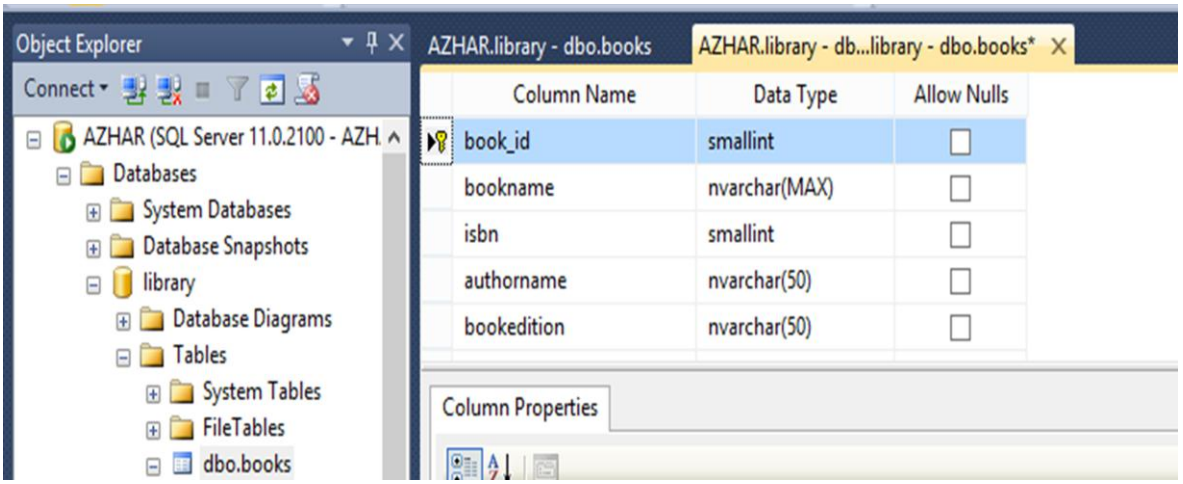
Student_id	Name	Gender	studentDepartment

Normalization End

ARCHITECTURE OF TABLES IN SQL SEERVER 2012 AND RECORD

FIRST TABLE IS BOOK

Design view



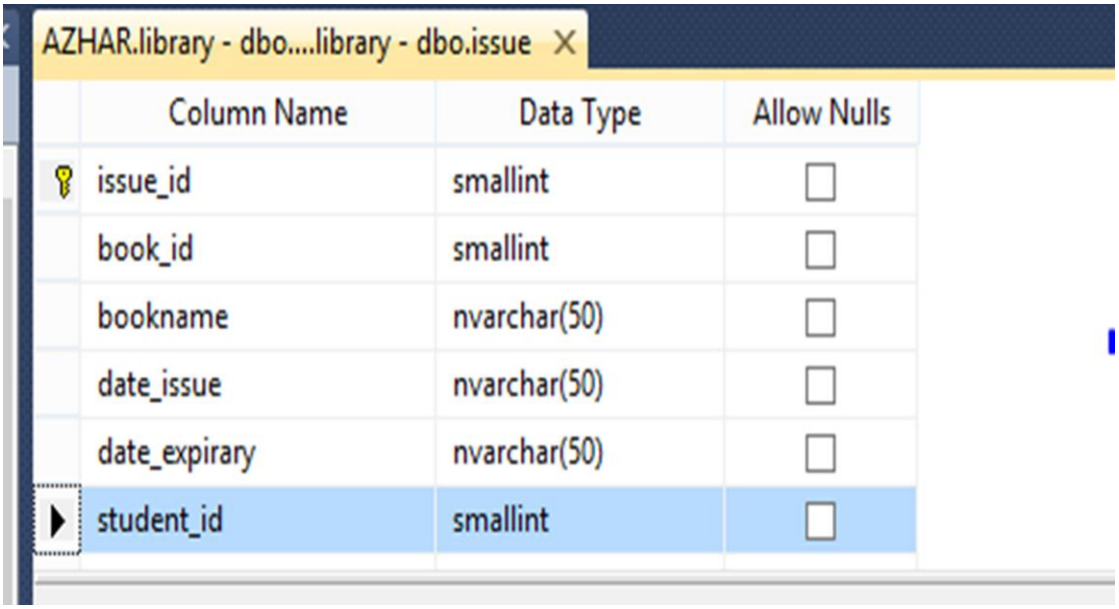
Records

AZHAR.library - dbo.books | AZHAR.library - db...library - dbo.books*

book_id	bookname	isbn	authorname	bookedition
1141	web pentration	4151	azhar	first
2567	Linux commond	5678	javed	second
4352	Ethical Hacking	5652	ramzan	first
5632	Hack the netwo...	9875	Mazhar	third
NULL	NULL	NULL	NULL	NULL

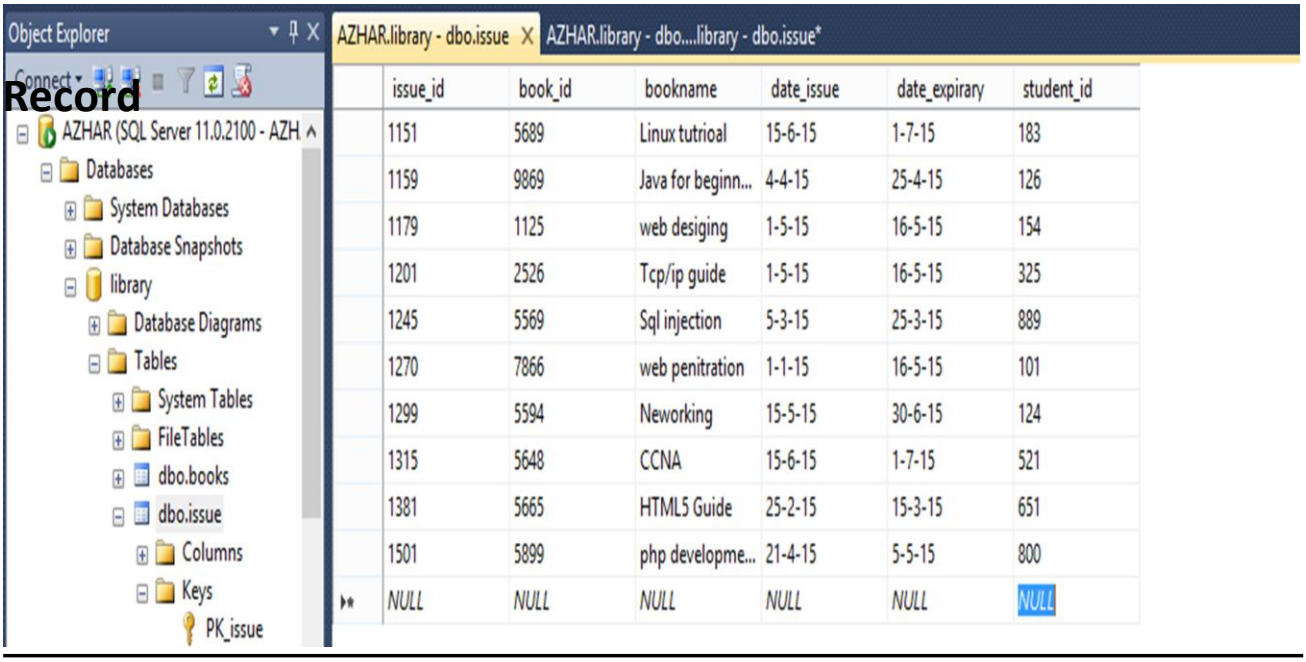
2nd table Issues

Design view



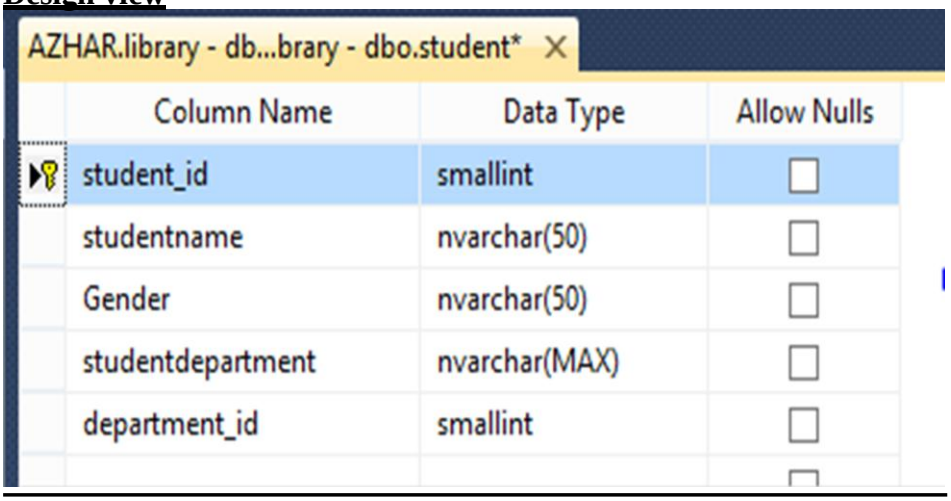
Student _id is forigen key in book table

Record



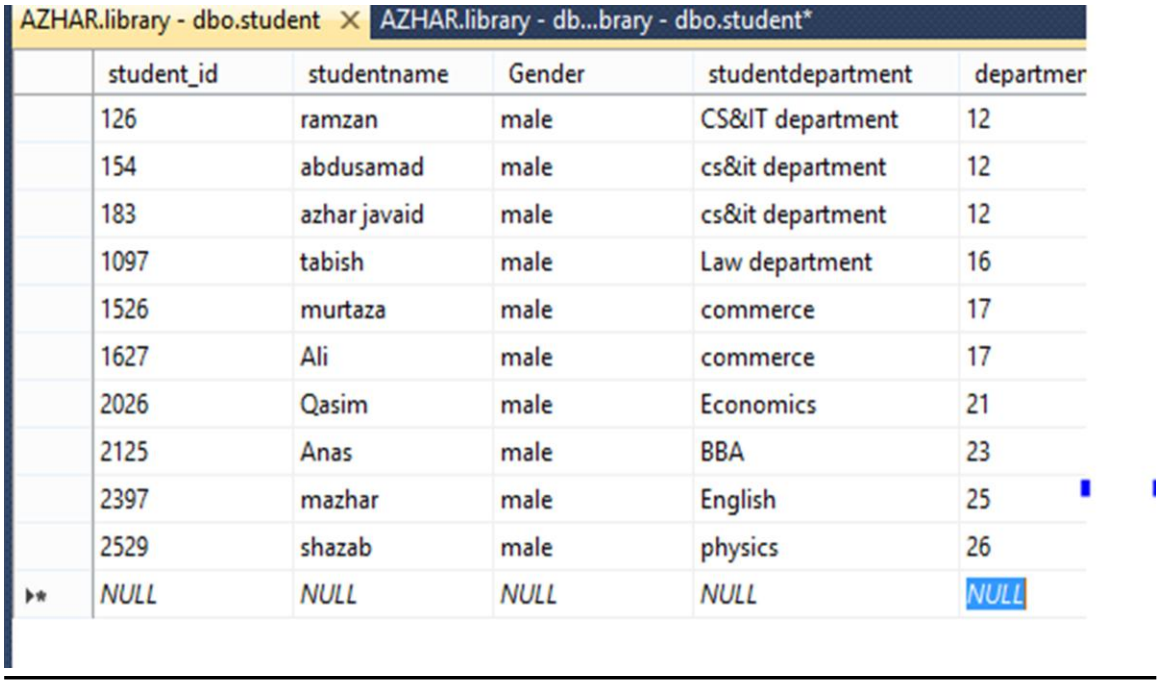
3rd table student

Design view



Dep_id is forign key in student table

Record view



student_id	studentname	Gender	studentdepartment	departmer
126	ramzan	male	CS&IT department	12
154	abdusamad	male	cs&it department	12
183	azhar javaid	male	cs&it department	12
1097	tabish	male	Law department	16
1526	murtaza	male	commerce	17
1627	Ali	male	commerce	17
2026	Qasim	male	Economics	21
2125	Anas	male	BBA	23
2397	mazhar	male	English	25
2529	shazab	male	physics	26
NULL	*NULL*	*NULL*	*NULL*	*NULL*

Student contact

Design view

AZHAR.library - db...bo.student contact*

Column Name	Data Type	Allow Nulls
student_id	smallint	☐
address	nvarchar(50)	☐
city	nvarchar(50)	☐
state	nvarchar(50)	☐
phone	smallint	☐

Record

AZHAR.library - dbo.student contact | AZHAR.library - db...bo.student contact

student_id	address	city	state	phone
126	chack 81	sargodha	punjab	3008136255
154	raza town	sargodha	punjab	3075408811
183	behria twon	sargodha	punjab	3127400558
196	block 12	sargodha	punjab	3001234567
224	block 4	sargodha	punjab	31212345678
231	block6	sargodha	punjab	3331234567
236	block 11	sargodha	punjab	3451234567
NULL	*NULL*	*NULL*	*NULL*	*NULL*

Department table - Design view

AZHAR.library - db...y - dbo.department

Column Name	Data Type	Allow Nulls
Dep_id	smallint	☐
Departmentname	nvarchar(50)	☐
student_id	smallint	☐
		☐

Here student_id is forigen key

Record view

AZHAR.library - dbo.department | AZHAR.library - db...y - dbo.department

Dep_id	Departmentna...	student_id
11	IT Department	183
13	physics	315
14	chemistry	501
17	commerce	751
18	business	901
19	economics	1171
21	food science	1401
22	english	1655
23	math	1831
24	islamic	2101
25	urdu	2405
26	education	2627
NULL	*NULL*	*NULL*

Return table

AZHAR.library - dbo.Table_1*

Column Name	Data Type	Allow Nulls
return_id	smallint	☐
book_id	smallint	☐
issue_date	date	☐
expairy_date	date	☐
return_date	date	☐
student_id	smallint	☐
staff_id	nvarchar(50)	☐
issue_id	smallint	☐
		☐

Here book_id,issue_id, staff_id ,student_id are forigen keys

Record view

AZHAR.library - dbo.return | AZHAR.library - dbo...ibrary - dbo.return

return_id	book_id	issue_date	expairy_date	return_date	student_id	staff_id	issue_id
5645	456	2015-01-03	2015-03-10	2015-09-03	115	sf12	2156
5699	1122	2015-01-05	2015-05-01	2015-07-04	526	sf15	556
5756	2654	2015-01-06	2015-01-15	2015-01-15	556	sf13	556
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

AZHAR.library - dbo.....library - dbo.staff*

Column Name	Data Type	Allow Nulls
staff_id	nvarchar(50)	☐
name	nchar(10)	☐
gender	nchar(10)	☐
Deg_id	nvarchar(50)	☐
		☐

Staff table

Design view

Deg_id is forigen key

Record view

AZHAR.library - dbo.staff | SQLQuery4.sql - A...hammad Azhar (52)) | AZHAR.library - dbo.....library - dbo.st

staff_id	name	qender	Deg_id
sf12	arshad	male	dg56
sf15	latif	male	dg60
sf16	irfan	male	dg65
sf17	faisal	male	dg66
sf18	roqia	femal	dg69
sf19	saira	female	dg70
sf20	muazamel	male	dg71
sf21	furqan	male	dg72
NULL	NULL	NULL	NULL

Staff deginations table

Design view

AZHAR.library - dbo.designation | AZHAR.library - db...y - dbo.designation

	Column Name	Data Type	Allow Nulls
🔑	Dg_id	nvarchar(50)	☐
	designation	nchar(10)	☐
	staff_id	nvarchar(50)	☐
			☐

Staff_id is forigen key Record view

AZHAR.library - dbo.designation | AZHAR.library - db...y - dbo.designation

	Dg_id	designation	staff_id
	dg3	naibqasid	sf15
	dg4	sweeper	sf10
	dg5	bookkeeper	sf11
	dg11	librairan	sf13
	dg12	clerk	sf16
▶*	NULL	NULL	NULL

Staff contact table Design view

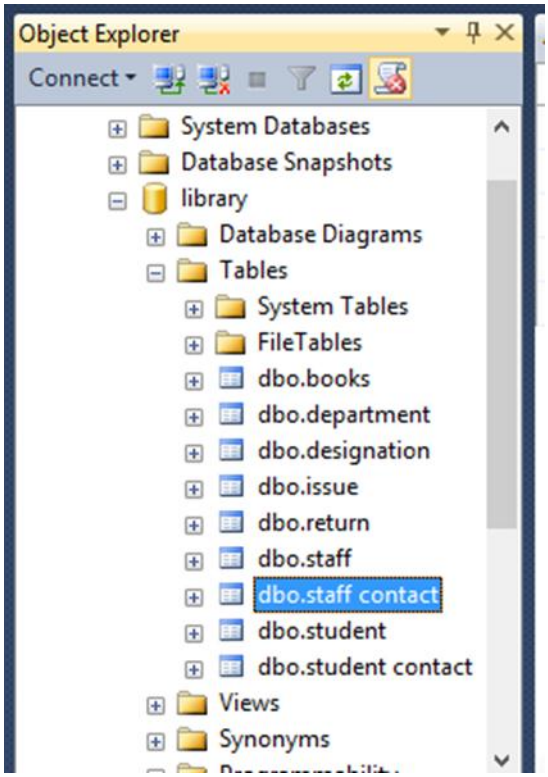
AZHAR.library - dbo...- dbo.staff contact*

	Column Name	Data Type	Allow Nulls
🔑	staff_id	nvarchar(50)	☐
	address	nvarchar(50)	☐
	city	nchar(10)	☐
	state	nchar(10)	☐
	phone	numeric(18, 0)	☑
			☐

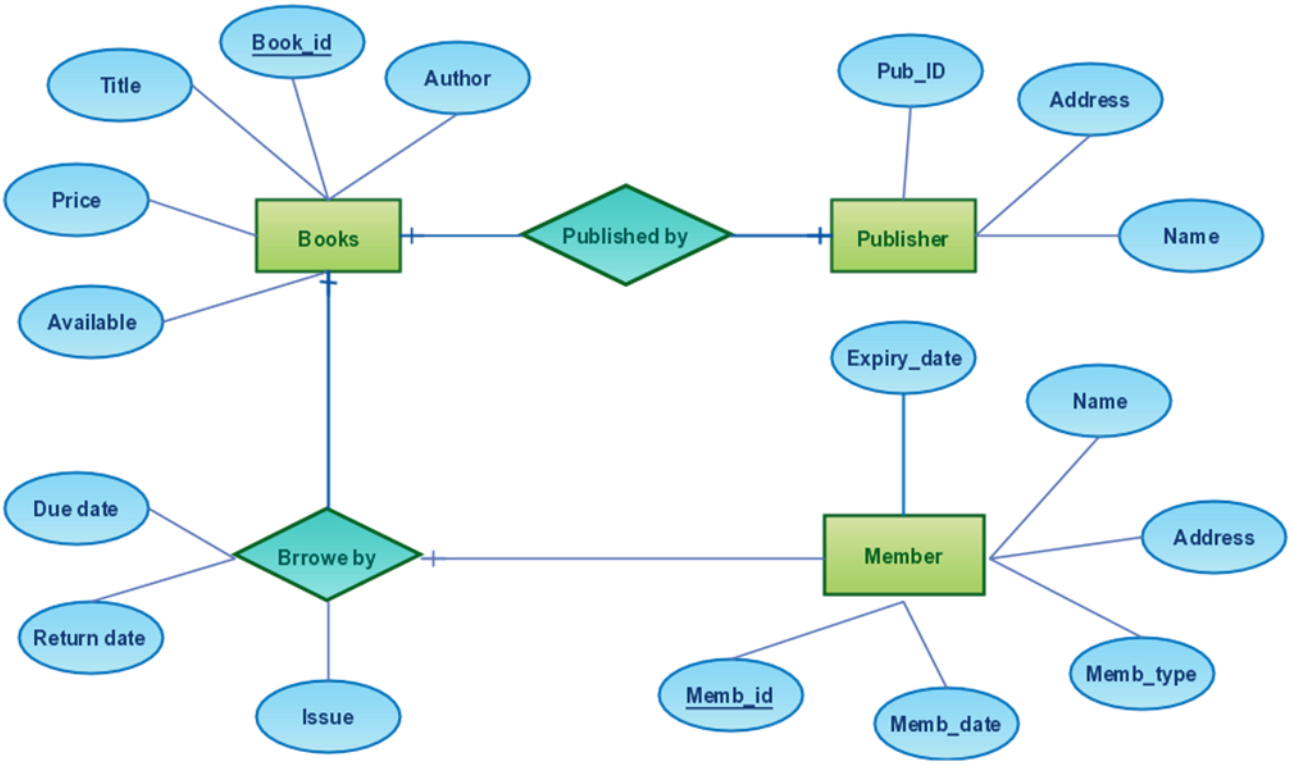
Record view

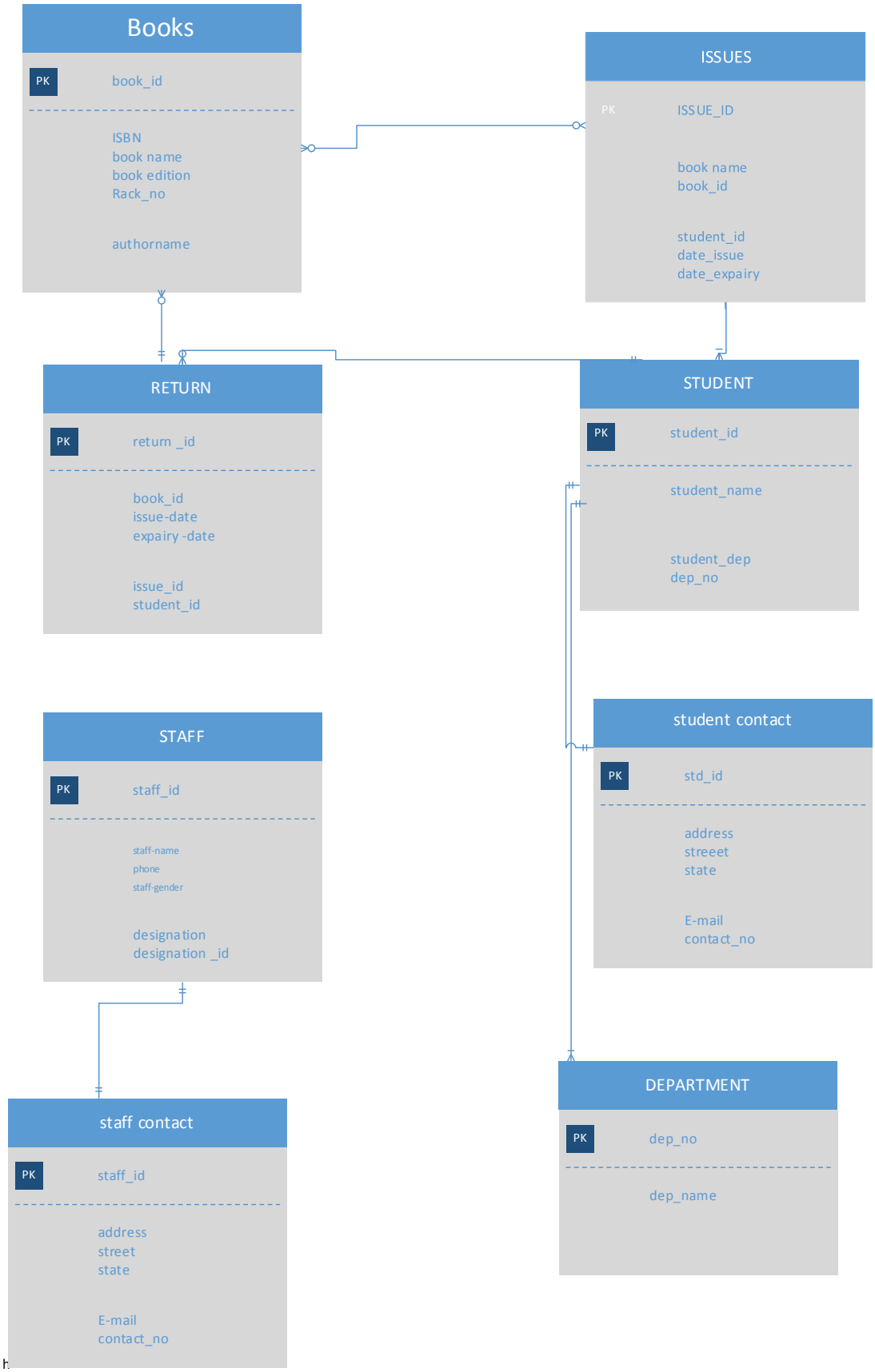
AZHAR.library - dbo.staff contact | AZHAR.library - dbo...- dbo.staff contact*

	staff_id	address	city	state	phone
	sf12	block 4	sargodha	punjab	3001234567
	sf15	main city	sargodha	punjab	3121234567
	sf17	green twon	sargodha	punjab	3331234567
	sf26	raza twon	sargodha	punjab	3461234567
▶*	NULL	NULL	NULL	NULL	NULL



E-R Diagram for Library Management System




Books
PK
book_id
ISBN
book name
book edition
Rack_no
authorname
ISSUES
PK
ISSUE_ID
book name
book_id
student_id
date_issue
date_expairy
RETURN
PK
return _id
book_id
issue-date
expairy -date
issue_id
student_id
STUDENT
PK
student_id
student_name
student_dep
dep_no
STAFF
PK
staff_id
staff-name
phone
staff-gender
designation
designation _id
student contact
PK
std_id
address
streeet
state
E-mail
contact_no
DEPARTMENT
PK
dep_no
dep_name
staff contact
PK
staff_id
address
street
state
E-mail
contact_no

Relational Model of ERD

Staff contact

Staff_id	Address	City	State	Phone

Staff

Staff_id	Name	Desgination	Gender

Designation

Designation_id **Designation**

Student contact

Student_id	Address	City		State	phone

Student

Sttudent_id	Name	Gender	Student_dep

Department

Dep_id **Department name**

Book

Book_id	Isbn	Book_name	Edition	Author	Rack_no
				name	

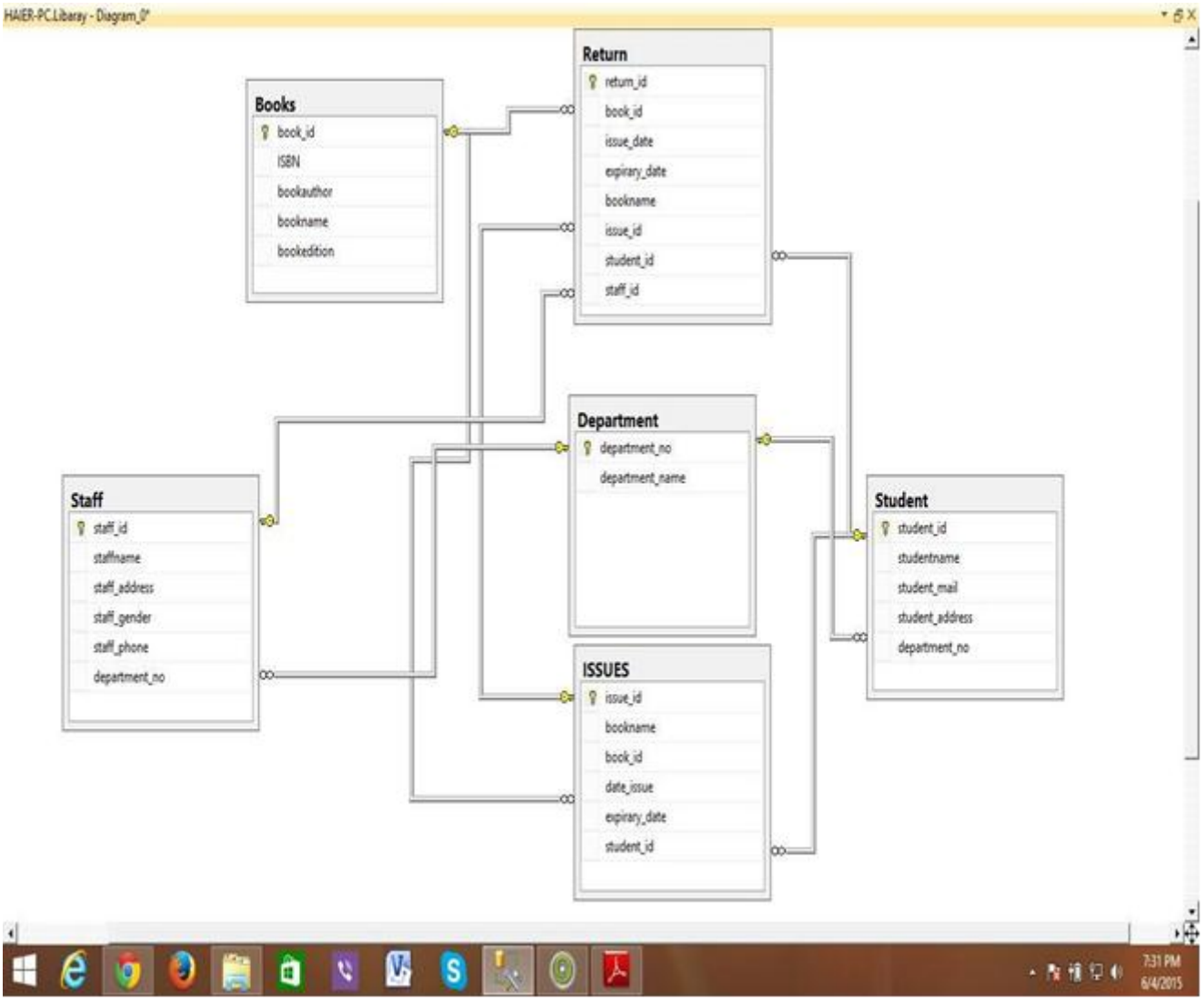
ISSUEs

Issue_id	Book_name	Book_id	Stud_id	Issue_date	Expairy_date

RETURN

Return_id	Book_id	Issues_date	Retuen_date

Entity Relationship Model in SQL SERVER 2012



CONNECTING TO ORACLE DATABASE USING C#

Introduction

Oracle is the most widely used database system available in the market. And on the other hand if we talk about the .Net framework, it's a software framework developed by the software giant Microsoft (we all know this), it includes a large class library known as the Framework Class Library (FCL) and provides language interoperability across several programming languages.

Prerequisites

1. Microsoft Visual Studio 2008 or higher (I'll be using Visual Studio 2012).
2. Oracle database 9.2 or higher or Oracle Database XE
3. Install Oracle Data Access Component (ODAC)

Use this link to download ODAC,

http://www.oracle.com/technetwork/topics/dotnet/utilsoft-086879.html

Moving Forward

Once you have installed ODAC, you must look for the following assembly files.

- Oracle.DataAccess.dll (this assembly file is required)
- Oracle.ManagedDataAccess.dll (optional)
- Oracle.Web.dll (optional)

Add the following references:

1. Go to Solution Explorer
2. Right-click and select Add Reference
3. Click on Extensions
4. Select the above mentioned files and click OK

Go to the web.config file and you can see the following code.

<assemblies>

<add assembly="Oracle.DataAccess,Version=4.121.1.0, Culture=neutral,

PublicKeyToken=*******"/>***

</ assemblies>

And you can also see a folder name, bin, that has already been created in the Solution Explorer.

ORA-12154: TNS: could not resolve the connect identifier specified
This error usually occurs when the code doesn't find the specified settings. This is quite confusing and irritating as well. The best thing you can do is declare the TNS setting in the code itself to prevent such errors.

The following is the sample of how to create TNS entries.

String oradb="Data Source=(DESCRIPTION=" + "address =(protocol=tcp) (host=your_host_name) (PORT =1521))" + "(CONNECT DATA=" + "(SERVER =DEDICATED)" + " (SERVICE_NAME=XE)));" + "User ID=your_user_id; Password=*****;";***

You must understand the following details before creating TNS entries.

1. Your host provider.

   This SQL query can help you determine the host name,

   ```
   1. SELECT SYS_CONTEXT ('USERENV', 'SERVER_HOST') FROM DUA
      L;
   ```

2. You need to know the Service name. You can find it in tnsnames.ora. This allows you to register an instance with the listener.
3. Third is the user id and password.
4. Don't forget to add "using Oracle.DataAccess.Client;" namespace.

The source code

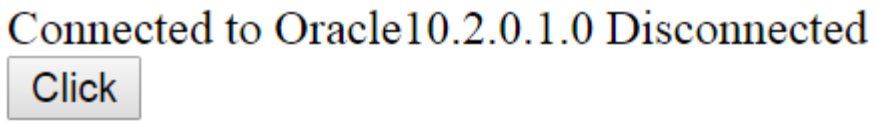
```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Web;
5.  using System.Web.UI;
6.  using System.Web.UI.WebControls;
7.  using Oracle.DataAccess.Client;
8.  using System.Data; & nbsp;
9.  public partial class _Default: System.Web.UI.Page {
10.    //creating TNS entries
11.    string oradb = "Data Source=(DESCRIPTION =" + "(ADDRESS = (PROTOCOL
    = TCP)(HOST = Your host name)(PORT = 1521))" + "(CONNECT_DATA =" + "(S
```

```
    ERVER = DEDICATED)" + "(SERVICE_NAME = XE)));" + "User Id= your user id
    ;Password=<strong>******</strong>;";
12.    protected void Page_Load(object sender, EventArgs e) {}
13.    protected void btn_Click(object sender, EventArgs e) {
14.        OracleConnection conn = new OracleConnection(oradb);
15.        conn.Open();
16.        Response.Write("Connected to Oracle" + conn.ServerVersion);
17.        // Close and Dispose OracleConnection object
18.        conn.Close();
19.        conn.Dispose();
20.        Response.Write("Disconnected");
21.    }
22. }
```

Remarks

- OracleConnection(): Initializes a new instance of the OracleConnection.
- OracleConnection(oradb): Initializes a new instance of the OracleConnection class with the specified connection string.
- OracleCommand(): Initializes a new instance of the OracleCommand.
- CommandText: Gets or sets the SQL statement or Stored Procedure to execute against the database. (Overrides DbCommand.CommandText.)
- Connection: Gets or sets the OracleConnection used by this instance of the OracleCommand.
- OracleDataReader: To create an OracleDataReader, you must call the ExecuteReader method of the OracleCommand object, rather than directly using a constructor. Changes made to a resultset by another process or thread when data is being read may be visible to the user of the OracleDataReader.

Output



FORM DESIGN

AIM:

To design a Single Document Interface and Multiple Document Interface forms using

Visual Basic.

PROCEDURE:

STEP 1: Start

STEP 2: Create the form with essential controls in tool box.

STEP 3: Write the code for doing the appropriate functions.

STEP 4: Save the forms and project.

STEP 5: Execute the form.

STEP 6: Stop

EXECUTION

Code for Dialog Menu:

```
Private Sub OKButton_Click()
If (Option1.Value = True) Then
SDI.Show
Unload Me
Else
MDIForm1.Show
Unload Me
End If
End Sub
```

Code for MDI Menu:

```
Private Sub ADD_Click()
MDIADD.Show
End Sub
Private Sub DIV_Click()
MDIDIV.Show
End Sub
Private Sub EXIT_Click()
End
End Sub
```

```
Private Sub MUL_Click()
MDIMUL.Show
End Sub
Private Sub SUB_Click()
MDISUB.Show
End Sub
```

Code for MDI ADD:

```
Private Sub Command1_Click()
Dim a As Integer
a = Val(Text1.Text) + Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
End Sub
Private Sub Command5_Click()
MDIForm1.Show
End Sub
```

Code for MDI DIV:

```
Private Sub Command1_Click()
Dim a As Integer
a = Val(Text1.Text) / Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
End Sub
Private Sub Command5_Click()
MDIForm1.Show
End Sub
```

Code for MDI MUL:

```
Private Sub Command1_Click()
Dim a As Integer
```

```
a = Val(Text1.Text) * Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
End Sub
Private Sub Command5_Click()
MDIForm1.Show
End Sub
```

Code for MDI SUB:

```
Private Sub Command1_Click()
Dim a As Integer
a = Val(Text1.Text) - Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
End Sub
Private Sub Command5_Click()
MDIForm1.Show
End Sub
```

Code for SDI MENU:

```
Private Sub Command1_Click()
SDIADD.Show
End Sub
Private Sub Command2_Click()
SDIMUL.Show
End Sub
Private Sub Command3_Click()
SDIDIV.Show
End Sub
Private Sub Command4_Click()
SDISUB.Show
```

```
End Sub

Private Sub Command5_Click()

Dialog.Show

Unload Me

End Sub
```

Code for SDI ADD:

```
Private Sub Command1_Click()

Dim a As Integer

a = Val(Text1.Text) + Val(Text2.Text)

MsgBox ("Addition of Two numbers is" + Str(a))

Unload Me

End Sub
Private Sub Command5_Click()

SDI.Show

Unload Me

End Sub
```

Code for SDI DIV:

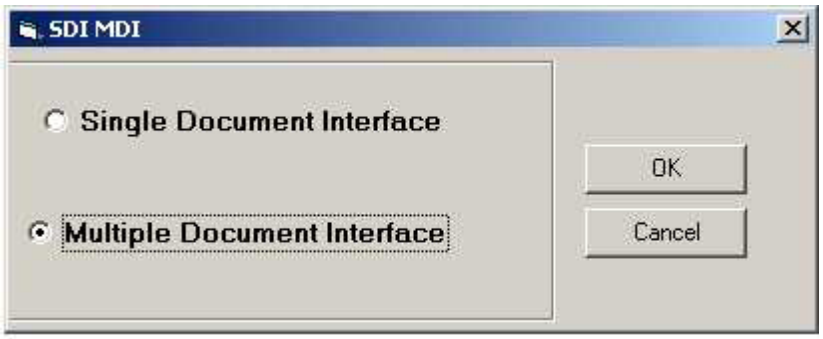
```
Private Sub Command2_Click()

a = Val(Text1.Text) / Val(Text2.Text)

MsgBox ("Addition of Two numbers is" + Str(a))

Unload Me

End Sub

Private Sub Command5_Click()

SDI.Show

Unload Me

End Sub
```

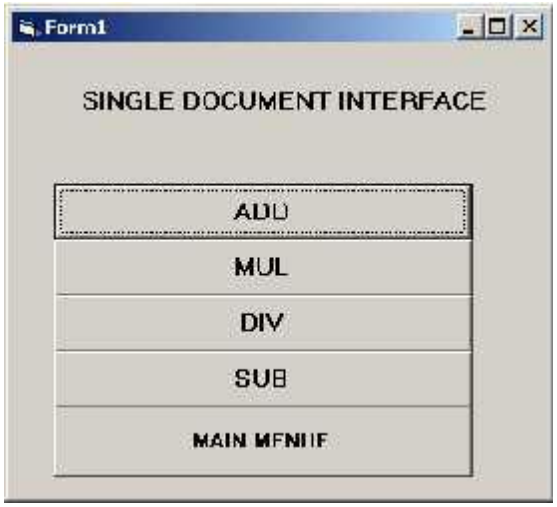
Code for SDI MUL:

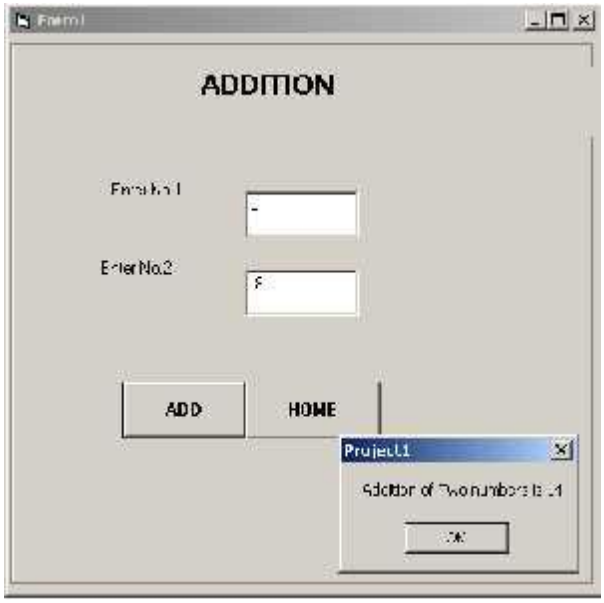
```
Private Sub Command2_Click()
a = Val(Text1.Text) * Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
Unload Me
End Sub
Private Sub Command5_Click()
SDI.Show
Unload Me
End Sub
```

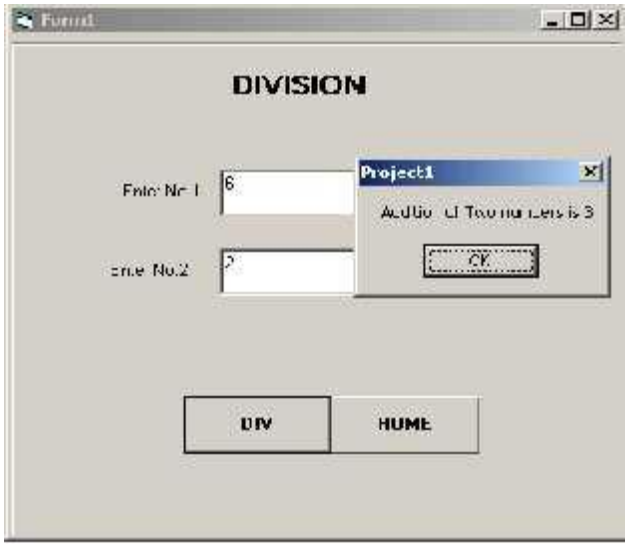
Code for SDI SUB:

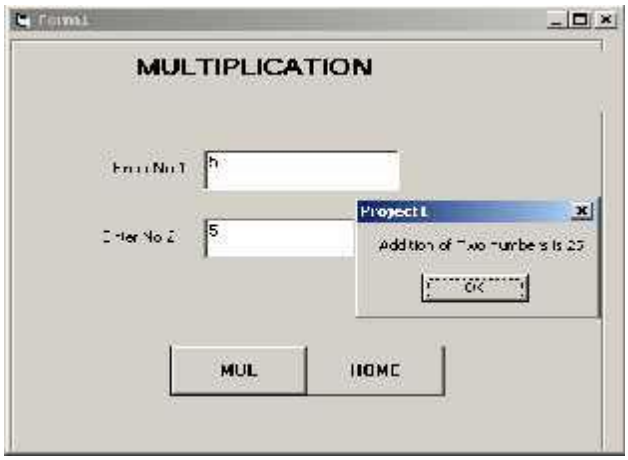
```
Private Sub Command2_Click()
a = Val(Text1.Text) - Val(Text2.Text)
MsgBox ("Addition of Two numbers is" + Str(a))
Unload Me
End Sub
Private Sub Command5_Click()
SDI.Show
Unload Me
End Sub
```

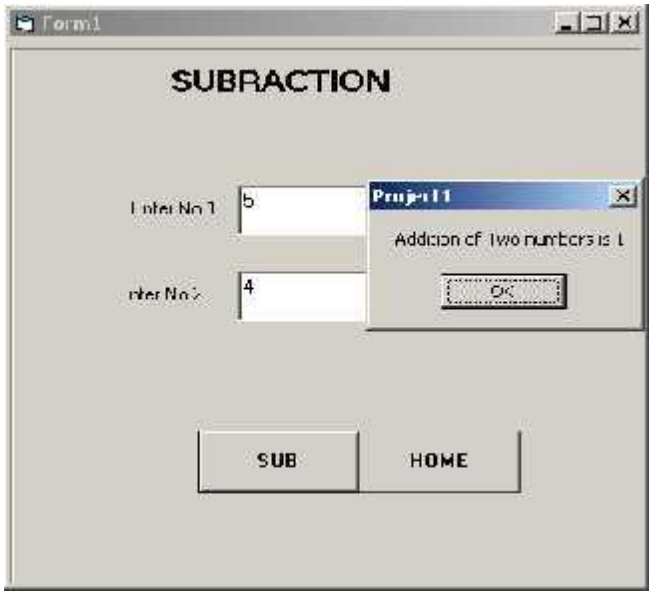


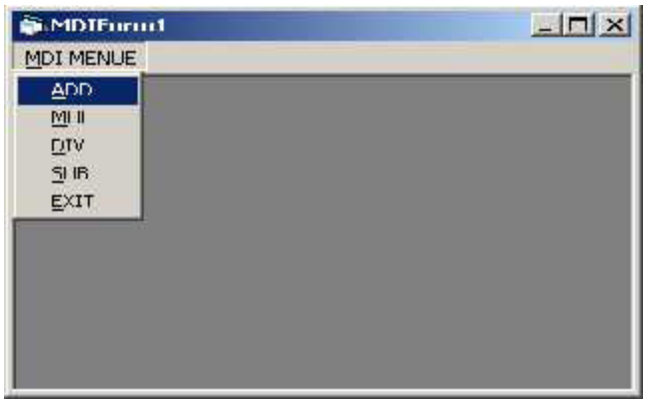

Form1
SINGLE DOCUMENT INTERFACE
ADD
MUL
DIV
SUB
MAIN MENUE

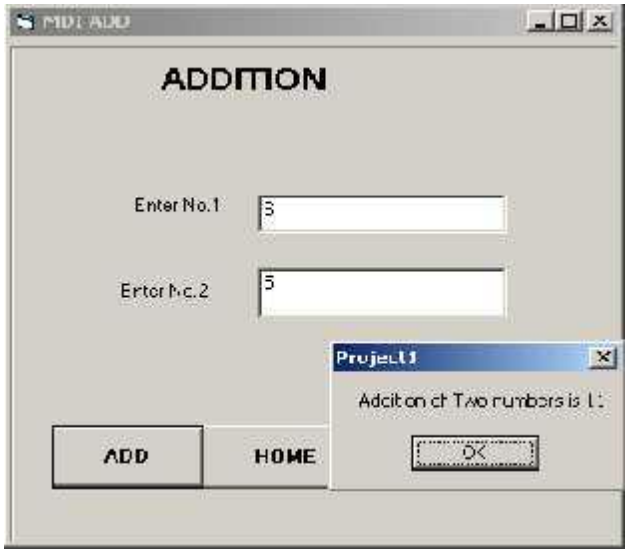

ADDITION
Enter No.2
ADD
HOME
Project1
OK

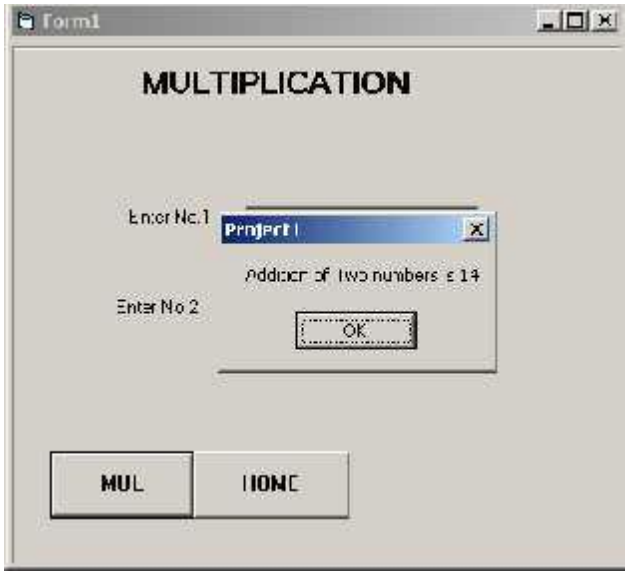

DIVISION
Project1
OK
DIV
HOME

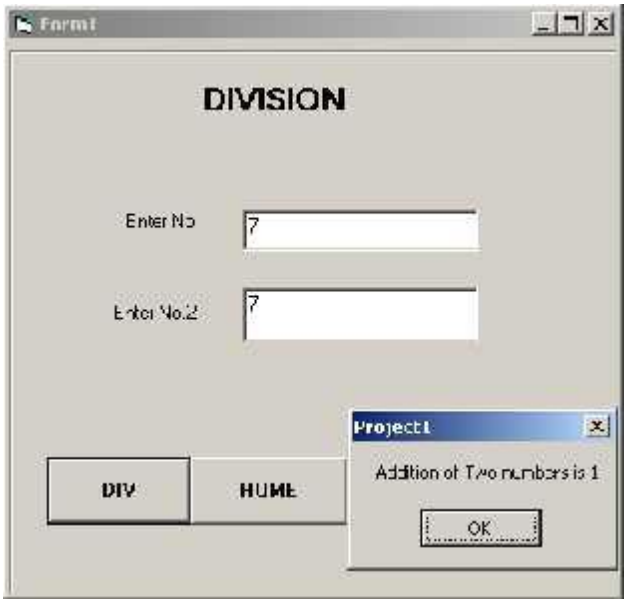

MULTIPLICATION
Project1
OK
MUL
HOME

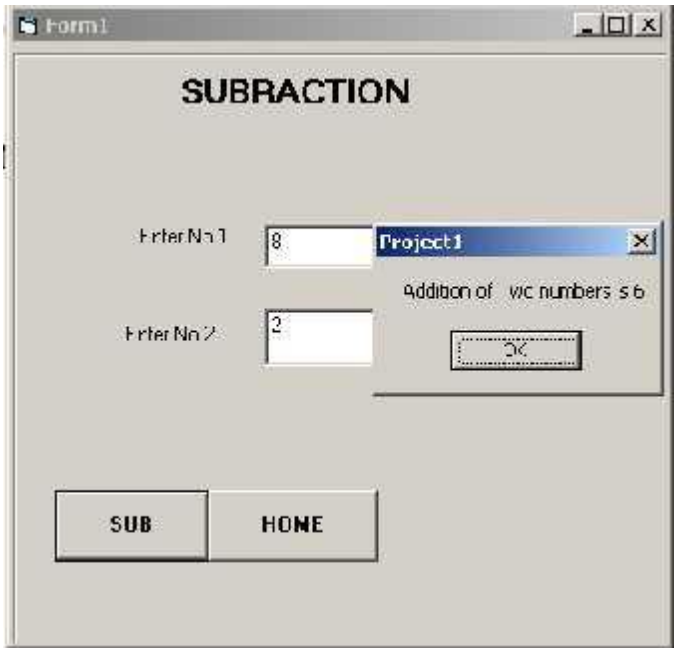

Form1
SUBRACTION
Project1
Addition of Two numbers is 1
OK
SUB
HOME


MDI MENUE
ADD
EXIT


MDI ADD
ADDITION
Enter No.1
5
Enter No.2
5
Project1
Addition of Two numbers is 1
OK
ADD
HOME


Form1
MULTIPLICATION
Enter No.1
Project1
Addition of Two numbers is 14
Enter No.2
OK
MUL
HOME





Result:

Thus the Single Document Interface and Multiple Document Interface forms in Visual Basic created and output verified successfully.

MENU DESIGN

AIM:

To design a Note Pad Application menu using Visual Basic.

PROCEDURE:

STEP 1: Start

STEP 2: Create the form with essential controls and insert the menu using menu editor.

STEP 3: Write the code for doing the appropriate functions.

STEP 4: Save the forms and project.

STEP 5: Execute the form.

STEP 6: Stop

EXECUTION:

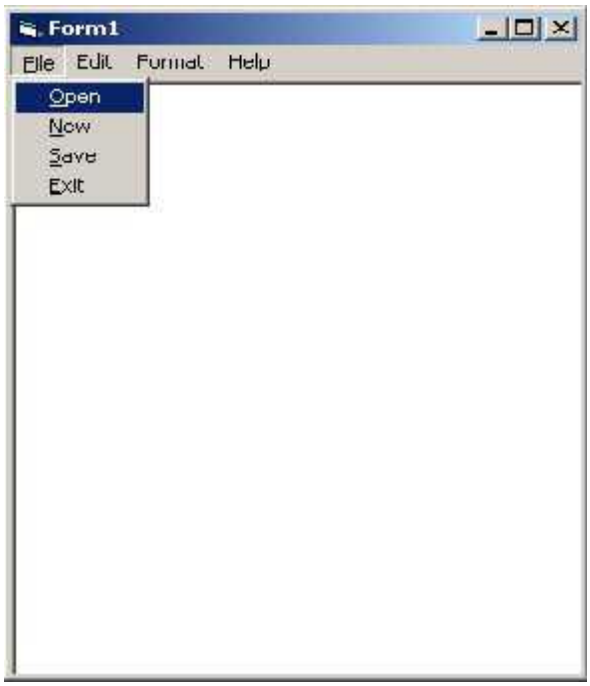
Coding:

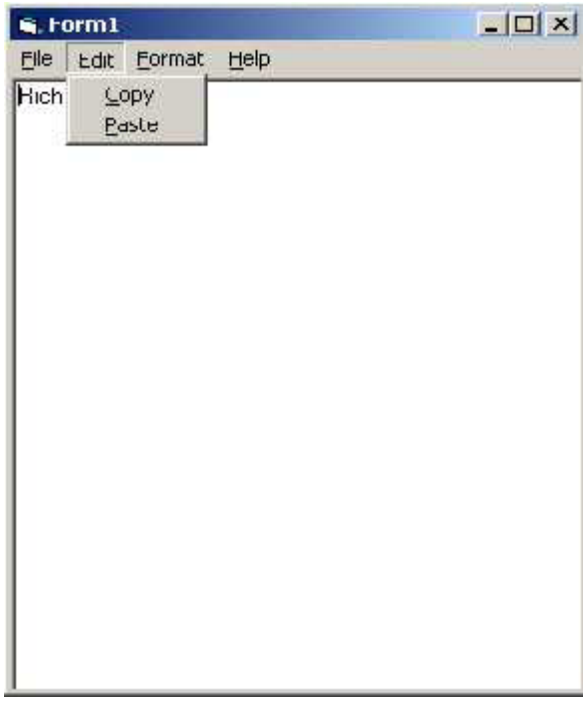
```
Private Sub ab_Click()
RichTextBox1.SelFontName = "Arial Black"
End Sub
Private Sub al_Click()
End Sub
Private Sub bold_Click()
RichTextBox1.SelBold = True
End Sub
Private Sub cb_Click()
RichTextBox1.SelColor = vbblue
End Sub
Private Sub cl_Click()
RichTextBox1.SelColor = vbred
End Sub
Private Sub copy_Click()
```

```
'Clipboard.SetText "richtextbox1.seltext", 1
'MsgBox Clipboard.GetText
Clipboard.SetText RichTextBox1.SelText, 1
RichTextBox1.SelText = Clipboard.GetText
MsgBox Clipboard.GetText
End Sub
Private Sub eighteen_Click()
RichTextBox1.SelFontSize = 18
End Sub
Private Sub exit_Click()
End
End Sub
Private Sub fcg_Click()
RichTextBox1.SelColor = vbgreen
End Sub
Private Sub fourteen_Click()
RichTextBox1.SelFontSize = 14
End Sub
Private Sub helpp_Click()
ans = MsgBox("visual basic sample notepad .....!", vbYes + vbinforamtion, "Help")
If ans = vbYes Then
Unload Me
End If
End Sub
Private Sub italic_Click()
RichTextBox1.SelItalic = True
End Sub
```

```
Private Sub MC_Click()
RichTextBox1.SelFontName = "Monotype Corsiva"
End Sub
Private Sub new_Click()
RichTextBox1 = ""
End Sub
Private Sub open_Click()
RichTextBox1.LoadFile ("C:\Notepad\Document.rtf")
End Sub
Private Sub paste_Click()
RichTextBox1.SelText = Clipboard.GetText
End Sub
Private Sub save_Click()
RichTextBox1.SaveFile ("C:\Notepad\Document.rtf")
End Sub
Private Sub sixteen_Click()
RichTextBox1.SelFontSize = 16
End Sub
Private Sub Th_Click()
RichTextBox1.SelFontName = "Tahoma"
End Sub
Private Sub tn_Click()
RichTextBox1.SelFontName = "Times New Roman"
End Sub
Private Sub twele_Click()
RichTextBox1.SelFontSize = 12
End Sub
```

```
Private Sub underline_Click()
RichTextBox1.SelUnderline = True
End Sub
Private Sub vbblue_Click()
RichTextBox1.SelColor = vbblue
End Sub
Private Sub vbgreen_Click()
RichTextBox1.SelColor = vbgreen
End Sub
Private Sub vbred_Click()
RichTextBox1.SelColor = vbred
End Sub
```




Form1
File
Edit
Format
Help
Rich
Copy
Paste


Form1
File
Edit
Format
Help
Bold
Italic
Underline
Font Face
font Size
Font Color Red
Font color Blue
Font Color Green
RichTextBox1

MINI PROJECT
INVENTORY CONTROL SYSTEM.

AIM:

To develop a Inventory Control System using Oracle as s back end(data base) and Microsoft Visual basic as a front end.

TABLE CREATION:

TABLE NAME:SUPPLIER

SQL> create table supplier(supno number(10),supname varchar2(20),supdate date,price number(20),quantity number(10),ITEM_NAME VARCHAR2(20));

Table created.

SQL> insert into supplier values(1,'pit','12-jan-2014',8000,2,'MONITOR');

1 row created.

SQL> insert into supplier values(2,'PEC','6-MAR-2014',4000,1,'KEYBOARD');

1 row created.

TABLE NAME:ITEM

SQL> CREATE TABLE ITEM(ITEMNO NUMBER(10),ITEM_NAME VARCHAR2(10),PRICE NUMBER(10),QUAT_AVAILABLE NUMBER(10));

Table created.

SQL> INSERT INTO ITEM VALUES(101,'MONITOR',80000,3);

1 row created.

SQL> insert into ITEM VALUES(102,'MOUSE',70000,10);

1 row created.

SQL> COMMIT;

Commit complete.

CODING

FORM1:

```
Private Sub Item_Click()

Form3.Show
```

```
End Sub
Private Sub Supplier_Click()
Form2.Show
End Sub
```

FORM2:

```
Private Sub Command1_Click()
Adodc1.Recordset.AddNew
Adodc1.Recordset.Fields("supno") = Text1.Text
Adodc1.Recordset.Fields("supname") = Text2.Text
Adodc1.Recordset.Fields("supdate") = Text3.Text
Adodc1.Recordset.Fields("price") = Text4.Text
Adodc1.Recordset.Fields("quantity") = Text5.Text
Adodc1.Recordset.Fields("item_name") = Text6Text
Adodc1.Recordset.Update
MsgBox "Data Added"
End Sub

Private Sub Command10_Click()
Form3.Show
End Sub
Private Sub Command3_Click()
Adodc1.Recordset.Delete
Adodc1.Recordset.Update
End Sub

Private Sub Command4_Click()
Unload Me
```

```
End Sub
Private Sub Command5_Click()
If Adodc1.Recordset.EOF = False Then
 Adodc1.Recordset.MoveNext
 Else
 MsgBox "END OF FILE!", vbOKOnly, "Warning"
 End If
End Sub

Private Sub Command6_Click()
Adodc1.Recordset.MoveFirst
End Sub

Private Sub Command7_Click()
Adodc1.Recordset.MoveLast
End Sub

Private Sub Command8_Click()
If Adodc1.Recordset.BOF = False Then
Adodc1.Recordset.MovePrevious
Else
MsgBox "BEGIN OF FILE!!", vbOKOnly, "Warning"
End If
End Sub

Private Sub Command9_Click()
Form1.Show
```

```
End Sub
```

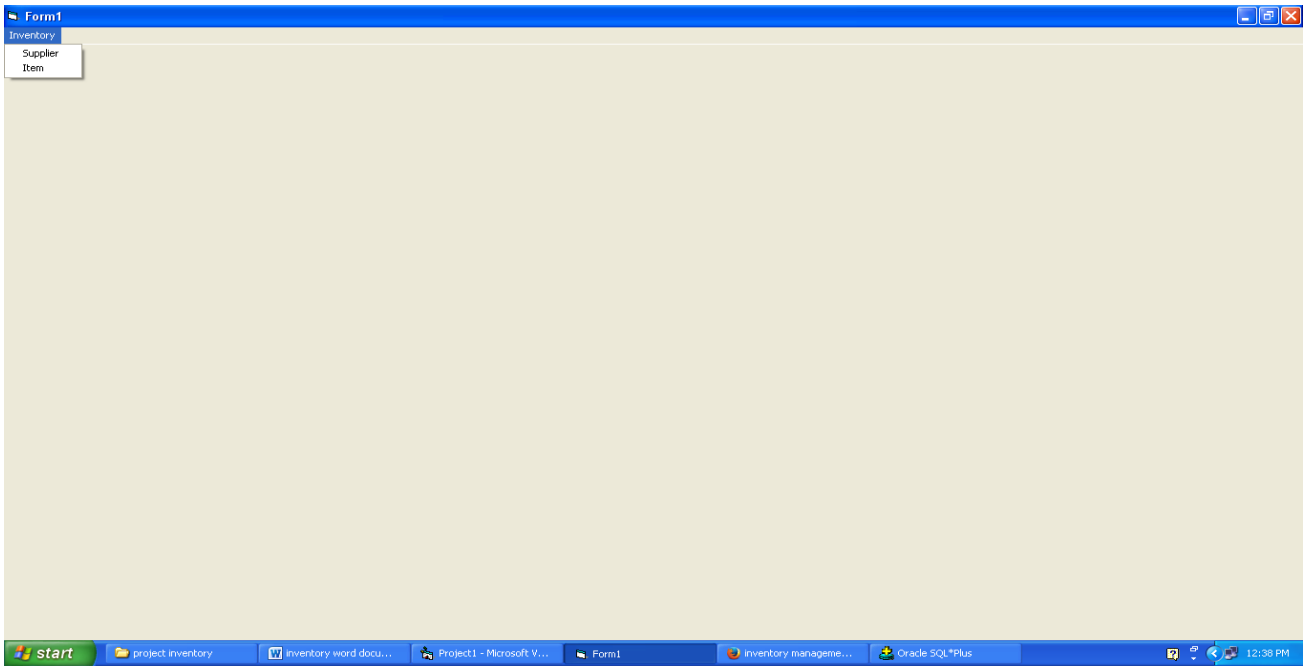
FORM3:

```
Private Sub Command1_Click()
Adodc1.Recordset.AddNew
Adodc1.Recordset.Fields("itemno") = Text1.Text
Adodc1.Recordset.Fields("item_name") = Text2.Text
Adodc1.Recordset.Fields("price") = Text4.Text
Adodc1.Recordset.Fields("quat_available") = Text5.Text
Adodc1.Recordset.Update
MsgBox "Data Added"
End Sub

Private Sub Command10_Click()
Form2.Show
End Sub

Private Sub Command3_Click()
Adodc1.Recordset.Delete
Adodc1.Recordset.Update
End Sub

Private Sub Command4_Click()
Unload Me
End Sub
```

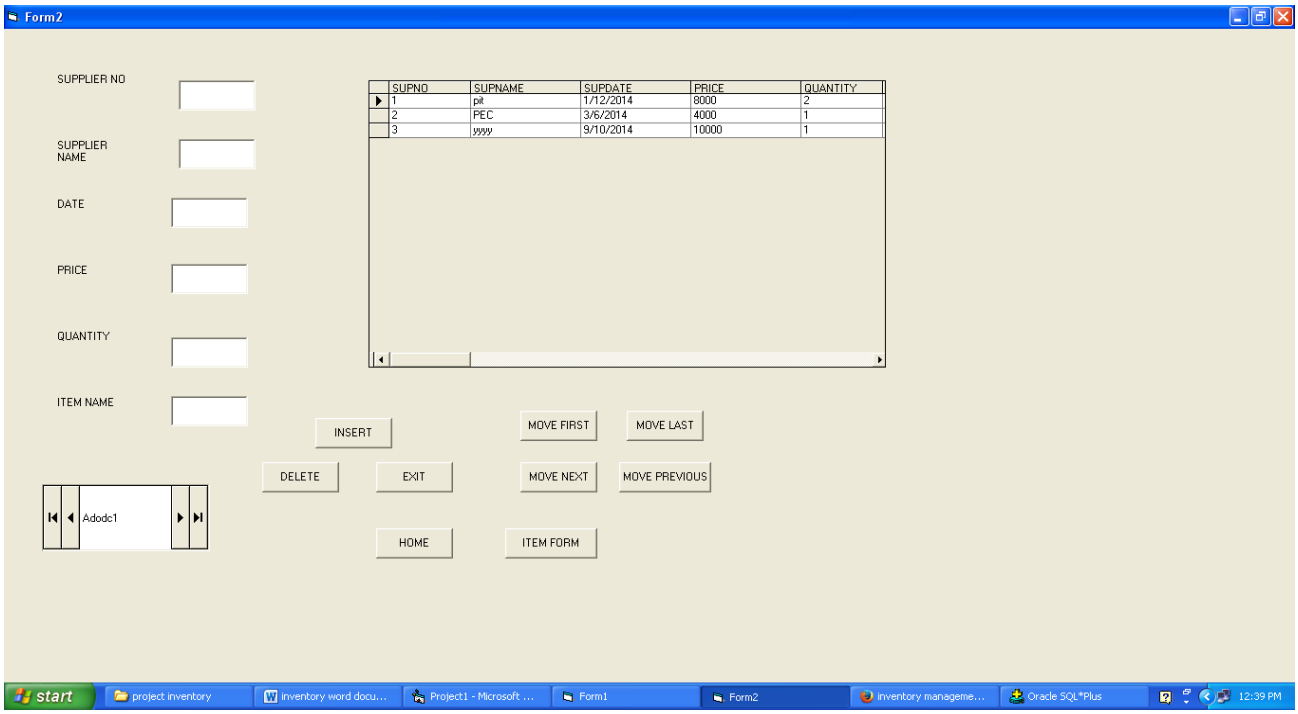
```
Private Sub Command5_Click()
If Adodc1.Recordset.EOF = False Then
Adodc1.Recordset.MoveNext
Else
MsgBox "END OF FILE!", vbOKOnly, "Warning"
End If
End Sub

Private Sub Command6_Click()
Adodc1.Recordset.MoveFirst
End Sub

Private Sub Command7_Click()
Adodc1.Recordset.MoveLast
End Sub

Private Sub Command8_Click()
If Adodc1.Recordset.BOF = False Then
Adodc1.Recordset.MovePrevious
Else
MsgBox "BEGIN OF FILE!!", vbOKOnly, "Warning"
End If
End Sub

Private Sub Command9_Click()
Form1.Show
End Sub
```

SCREEN SHOTS:

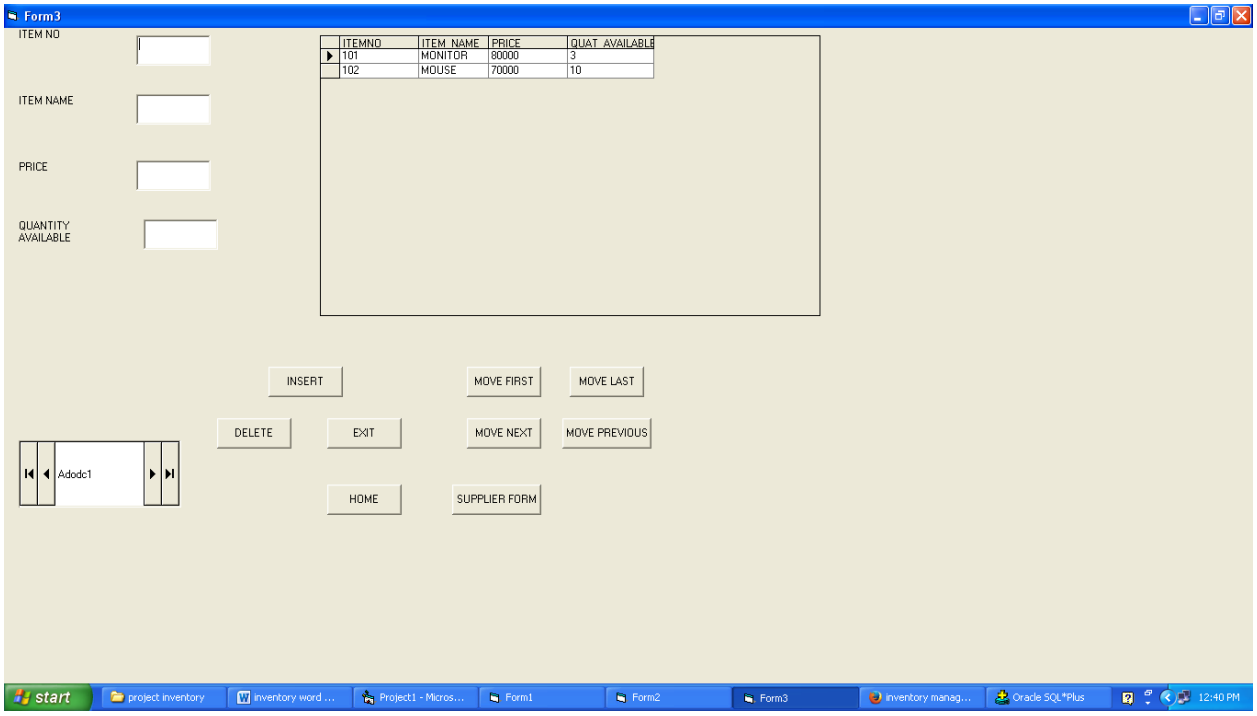
HOME PAGE



SUPPLIER FORM



ITEM FORM



Result:

Thus the Inventory Control System Mini project has been implemented and executed successfully.