BIITSeries

Lab Manual

Distributed Database Systems

A simplified practical work book of Distributed Database management System course for Computer Science, Information Technology and Software Engineering students

Student Name:	
Registration No:	
Section / Semester:	
Submission Date:	
Student Signature:	
Marks Obtained:	
Maximum Marks:	
Examiner Name/Sig:	

Dr. Munir Ahmed
Mr. Nauman Iqbal

Lab Manual Distributed Database Management System: First Edition
Author(s):Dr. Munir Ahmad, Mr. Nauman Iqbal
Publication Date: 23/2/2018
A publication of BIIT Series

Learning Objectives: By the end of the lab work, students should have following skills;

Sr.	Learning Objective	LAB No.
1	Revision of Basic SQL Queries	1
2	UNION / INTERSECTION / SET DIFFERENCE	2
3	Cartesian Product	3
4	JOINS	4
5	Variable Declaration / Condition Statement / Loops / Stored Procedure Basics	5
6	Stored Procedure with Arguments	6
7	Stored Procedure Output Arguments	7
8	Transactions in SQL	8
9	Triggers – After	9
10	Triggers – Instead of Temporary Tables – inserted / deleted	10
11	Case Study Triggers	11
12	Basic Cursors / Cursor Examples	12
13	Database Connectivity JAVA / C#	13
14	Database Connectivity Android	14
15	Database Connectivity iOS	15

16	Lab Project Evaluation	16

LAB # 01

This lab is dedicated to revise all queries taught in database management course. Revision and evaluation of students will be done.

Query List:

List of all queries of Database Management System course are given below

1- Select Statement
   a. Columns
   b. AS
   c. Arithmetic operators (+ , - , *, /)
   d. Convert
   e. Top
   f. Percent
   g. Distinct
2- Where Statement
   a. AND / OR
   b. IN
   c. BETWEEN
   d. NOT
   e. LIKE / [ ] / ^
3- ORDER by
4- Aggregate Functions
   a. SUM
   b. COUNT
   c. MAX
   d. MIN
   e. AVG
5- Group By
6- Having
7- Sub Query
8- Create
   a. Database
   b. Table
9- Add Constraints
   a. PRIMARY KEY
   b. FOREIGN KEY
   c. CHECK
   d. DEFAULT
10- INSERT / UPDATE / DELETE
11- ALTER
   a. Add Column / Drop Column
   b. Add Constraint / Drop Constraint

c. Change Data-Type Column

Query Examples:

Assume given table.

Employee(eid, eName, eAge, eCity, eStreet, eHouseno, eSalary, eBonus, eDepart)
Project (pid, pName, pBonus, pDuration, eid)

1- SELECT

```
SELECT * FROM Employee
```

Above Query will display all rows and columns of Employee Table

```
SELECT eid, eName, eAge FROM Employee
```

Above Query will display all rows and eid, ename and eage columns from Employee table.

```
SELECT ECITY + ‘ ‘ + ESTREET + ‘ ‘ + EHOUSE  AS [ADDRESS] FROM Employee
```

- Display address of employee by merging 3 columns “ ecity, street and ehouse”. Column name will be “ADDRESS”

```
SELECT  Esalary + eBonus  AS [Total Salary] FROM Employee
```

- Displays full salary of all employee by adding eSalary column and eBonus column of table and column name will be “Total Salary”.

```
SELECT eName + ‘salary is ‘ + convert(varchar,eSalary) FROM Employee
```

- Displays name of employee concatenating “salary is “ and salary of employee. Convert function will convert int data-type to varchar of all values from salary column.

NOTE: different datatype columns cannot concatenate.

```
SELECT T0P 3 * FROM Employee
```

- Displays top 3 records from employee table

```
SELECT TOP 50 PERCENT * FROM Employee
```

- Displays top 50 % records of employee table.

```
SELECT DISTINCT eName FROM Employee
```

- Displays all distinct names from employee table. (name will not repeat)

2- WHERE

```
SELECT * FROM Employee WHERE EID = 101
```

- Displays all data of those employees whose id is 101

```
SELECT * FROM Employee WHERE ESALARY > 10000 AND ECITY = ‘ISB’
```

- Displays all those employees who live in ‘isb’ and their salary is above 10000

```
SELECT * FROM Employee WHERE ECITY =’RWP’ OR ECITY= ‘LHR’
```

- Displays all those employees who live in rwp or lhr

```
SELECT * FROM Employee WHERE CITY IN (‘RWP’,’LHR’)
```

- Displays all those employees who live in rwp or lhr

```
SELECT * FROM Employee WHERE AGE BETWEEN 18 AND 30
```

- Displays all those employees whose age is between 18 and 30

```
SELECT * FROM Employee WHERE NAME LIKE ‘a%’
```

- Displays all those employees whose name starts with ‘a’

```
SELECT * FROM Employee WHERE NAME NOT LIKE ‘a%’
```

- Displays all those employees whose name does not starts with ‘a’

```
SELECT * FROM Employee
```

3- ORDER BY

```
SELECT * FROM Employee ORDER BY ENAME
```

- Displays data of all employee in ascending order w.r.t employee’s name

```
SELECT * FROM Employee ORDER BY ENAME DESC
```

- Displays data of all employee in descending order w.r.t employee’s name

4- AGGREGATE FUNCTIONS

```
SELECT COUNT(*) FROM Employee
```

- Display total number of rows in employee table. Column name will appear as ‘No Column’

```
SELECT MAX(eAge) FROM Employee
```

- Displays maximum age from all employees

```
SELECT MIN(eAge) FROM Employee
```

- Display age of youngest employee

```
SELECT SUM(eSalary) FROM Employee
```

- Displays sum of salaries of all employees

```
SELECT AVG(eBonus) FROM Employee
```

5- GROUP BY / HAVING

```
SELECT eCity, COUNT(*) FROM Employee GROUP BY eCity
```

- Displays total number of employees of each city.

```
SELECT eDepart, SUM(eSalary) FROM Employee GROUP BY eDepart
```

- Displays sum of all salaries given to employees of each department.

```
SELECT eDepart, SUM(eSalary) FROM Employee GROUP BY eDepart
          HAVING SUM(eSalary) > 10000
```

- Displays only those departments whose sum of salaries is above 10000.

6- SUB QUERY

```
SELECT eName FROM Employee where eid IN  (SELECT EID FROM PROJECT)
```

- Displays names of all those employees who have been assigned any project.

```
SELECT * FROM Employee where eid IN ( SELECT eid FROM PROJECT WHERE pBonus >
50000)
```

- Displays data of all those employees who are assigned any project and project bonus is more than 50000.

```
SELECT * FROM Employee WHERE eAge IN ( SELECT MIN(eAge) FROM Employee)
```

- Displays data of youngest employees.

7- CREATE DATABASE / TABLE with Constraints

```
CREATE DATABASE DDBS2018
```

- Will create a database named 'DDBS2018'

```
CREATE TABLE EMPLOYEE
(
  Eid  INT PRIMARY KEY,
  ENAME VARCHAR(20),  EAGE INT CHECK (EAGE BETWEEN 18 AND 60),
  ECITY VARCHAR(30), ESTREET VARCHAR(5), EHOUSE VARCHAR(5),
  ESALARY INT CHECK (ESALARY > 10000), EBONUS INT,
  EDEPART VARCHAR(15) CHECK (EDAPART IN ('HR', 'CS' , 'IT')) DEFAULT 'HR'
)
```

Will create Employee table as described above. With following constraints

- Eid as primary key
- Eage between 18 and 60
- Salary above 10000
- Departments must be 'HR' or 'CS' or 'IT' and default department is 'HR'

```
CREATE TABLE PROJECT
(
  PID INT , PNAME VARCHAR(20), PBONUS INT, PDURATION INT,
  EID INT FOREIGN KEY REFERENCES EMPLOYEE(EID),
  PRIMARY KEY (PID, EID)
)
```

Above query will create project table with following constraints.

- Pid and eid combining to form composite key
- Eid is foreign key from Employee table

8- INSERT / UPDATE / DELETE

```
INSERT INTO PROJECT VALUES (1, 'PF', 15000, 10, 1)
```

- Insert a record in project table

```
INSERT INTO PROJECT (PID, EID, PNAME) VALUES (2,1,'DBS')
```

- Insert a record in project table only in mentioned columns, (pid, eid, pname). 'NULL' value will be stored in rest of the columns.

```
UPDATE EMPLOYEE
SET ENAME = 'ADIL'
WHERE EID = 101
```

- Will update employee name to 'adil' whose eid is 101

```
DELETE EMPLOYEE
```

- Delete all data of employee. (it will not delete table, only data of table)

```
DELETE EMPLOYEE WHERE ECITY = 'ISB'
```

- Delete all rows in which city is 'ISB'

9- ALTER

```
ALTER TABLE EMPLOYEE
ADD ERANK VARCHAR(20)
```

- Add new column to employee table of "eRank"

```
ALTER TABLE EMPLOYEE
DROP COLUMN ERANK
```

- Drop column of 'eRank' from employee table.

```
ALTER TABLE EMPLOYEE
ALTER COLUMN EBONUS FLOAT
```

- Change data type of column eBonus from INT to FLOAT

```
ALTER TABLE EMPLOYEE
ALTER COLUMN EID NOT NULL

ALTER TABLE EMPLOYEE
ADD CONSTRAINT PK_EID PRIMARY KEY (EID)
```

- Above query will add new constraint of primary key to column EID (assuming that user forgot to add primary key at the time of creation of table).
- We have to make column NULLABLE first.

```
ALTER TABLE PROJECT
ADD CONSTRAINT FK_EID FOREIGN KEY (EID) REFERENCES EMPLOYEE(EID)
```

- Above query will add new constraint of foreign key to column eid of PROJECT table (assuming that user forgot to add foreign key at the time of creation)

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT CHK_AGE CHECK (EAGE BETWEEN 18 AND 60)
```

- Above query will add new CHECK constraint to Employee table on age column (assuming that user did not add any check constraint on eAge column at time of creation)

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT df_depart DEFAULT ‘HR’ FOR eDepart
```

- Above query will create default constraint for eDepart column

```
ALTER TABLE EMPLOYEE
DROP CONSTRAINT ANY NAME

EXAMPLE
ALTER TABLE EMPLOYEE
DROP CONSTRAINT CHK
```

- Above query is to drop any type of constraint.

TASK 1.1

Employee (Eid, Ename,EAge, ECity, E_Street, E_House#, E_Basic_Salary, E_Bonus,E_Rank, E_Department)

- Eid is Primary key.
- E_Rank can be “Manager, Developer, Admin”
- E_Department can be “HR, CS, IT, Labour”
- Employee Age between 18 and 60.
- Default value for department is “CS”.

Project(p_id, pName, p_bonus,duration_in_months, Eid)

- Pid and Eid combine makes Primary key.

NOTE: Eid in project is foreign key from Employee.
Write SQL Queries for the following statements.

1- Display whole address of employee along with name, columns names must be “Emp Name” and “Address”.

2- Display 30% top records of Employees.

3- Display names of all those employees whose address is “house number 44, street no 25, Rwp”.

4- Display data of employees who lives in “isb or rwp or khi or lhr”. (do not use or keyword)

5- Display those employees whose age is more than 18 and less than 33. (do not use relational operators “ > or <”.

6- Display all those employees whose name starts with “S” and does not contain “I” in it.

7- Display all cities of employee. (city name must not repeat)

8- Display All employees of “isb”. Employees must be displayed from younger to older. Younger first and older later.

9- Display name of employee and total salary of employee. (total salary is sum of basic salary and bonus)

10- Display total number of employees.

11- Display total number of employees working in each department.

12- Display Total expenditures of company. (total expenditures = basic salary + bonus of employee + all project bonus)

13- Display names of projects in which ‘Ahmar’ is working.

14- Display total project bonus of ‘Danish’.

15- Display total expenditures of only those departments whose total expenditures are more than 1 Million. (total expenditures = sum of basic salary of all employees of that department)

16- Create Employee table.

17- Create Project table.

18- Insert a record in employee table, (you must not enter edepartment and ebonus).

19- change basic salary of all employees, add increment of 5% to all.

20- Delete only those employees whose salary is above 1Million.

21- Change data type employee bonus to “float”.

22- Assume that you forgot to add primary key in Employee table, write a code that add a primary key in employee table.

23- Drop primary key constraint you created in above query.

24- Add a new column ‘e_father_name’ in employee table.

25- Drop E_age column from employee table.

LAB # 2

UNION

The SQL UNION operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements.

Each SELECT statement within the UNION must have the same number of fields in the result sets with similar data types.

Syntax

The syntax for the UNION operator in SQL is:

```
SELECT expression1, expression2, ... expression_n

FROM tables

[WHERE conditions]

UNION

SELECT expression1, expression2, ... expression_n

FROM tables

[WHERE conditions];
```

Parameters or Arguments

expression1, expression2, expression_n
The columns or calculations that you wish to retrieve.

TABLES:
The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

WHERE conditions:
Optional. The conditions that must be met for the records to be selected.

Note:

- There must be same number of expressions in both SELECT statements.
- The corresponding expressions must have the same data type in the SELECT statements. For example: *expression1* must be the same data type in both the first and second SELECT statement.

Example - Single Field With Same Name

Let's look at how to use the SQL UNION operator that returns one field. In this simple example, the field in both SELECT statements will have the same name and data type.

For example:

```
SELECT supplier_id

FROM suppliers

UNION

SELECT supplier_id

FROM orders

ORDER BY supplier_id;
```

In this SQL UNION operator example, if a *supplier_id* appeared in both the *suppliers* and *orders* table, it would appear once in your result set. The UNION operator removes duplicates. If you do **not** wish to remove duplicates, try using the UNION ALL operator.

Now, let's explore this example further will some data.

If you had the *suppliers* table populated with the following records:

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *orders* table populated with the following records:

order_id	order_date	supplier_id
1	2015-08-01	2000
2	2015-08-01	6000
3	2015-08-02	7000
4	2015-08-03	8000

And you executed the following UNION statement:

```
SELECT supplier_id FROM suppliers

UNION

SELECT supplier_id FROM orders ORDER BY supplier_id;
```

You would get the following results:

supplier_id
1000
2000
3000
4000
6000

supplier_id
7000
8000

As you can see in this example, the UNION has taken all *supplier_id* values from both the *suppliers* table as well as the *orders* table and returned a combined result set. Because the UNION operator removed duplicates between the result sets, the supplier_id of 2000 only appears once, even though it is found in both the *suppliers*and *orders* table. If you do not wish to remove duplicates, try using the UNION ALL operator instead.

Example - Different Field Names

It is not necessary that the corresponding columns in each SELECT statement have the same name, but they do need to be the same corresponding data types.

When you don't have the same column names between the SELECT statements, it gets a bit tricky, especially when you want to order the results of the query using the ORDER BY clause.

Let's look at how to use the UNION operator with different column names and order the query results.

For example:

```
SELECT supplier_id, supplier_name FROM suppliers WHERE supplier_id > 2000

UNION

SELECT company_id, company_name FROM companies WHERE company_id > 1000

ORDER BY 1;
```

In this SQL UNION example, since the column names are different between the two SELECT statements, it is more advantageous to reference the columns in the ORDER BY clause by their position in the result set. In this example, we've sorted the results by supplier_id / company_id in ascending order, as denoted by the `ORDER BY 1`. The supplier_id / company_id fields are in position #1 in the result set.

Now, let's explore this example further with data.

If you had the *suppliers* table populated with the following records:

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *companies* table populated with the following records:

company_id	company_name
1000	Microsoft
3000	Apple
7000	Sony
8000	IBM

And you executed the following UNION statement:

```
SELECT supplier_id, supplier_name FROM suppliers WHERE supplier_id > 2000
UNION
SELECT company_id, company_name FROM companies WHERE company_id > 1000
ORDER BY 1;
```

You would get the following results:

supplier_id	supplier_name
3000	Apple
4000	Samsung

supplier_id	supplier_name
7000	Sony
8000	IBM

First, notice that the record with *supplier_id* of 3000 only appears once in the result set because the UNION query removed duplicate entries.

Second, notice that the column headings in the result set are called *supplier_id* and *supplier_name*. This is because these were the column names used in the first SELECT statement in the UNION.

If you had wanted to, you could have aliased the columns as follows:

```
SELECT supplier_id AS ID_Value, supplier_name AS Name_Value
FROM suppliers
WHERE supplier_id > 2000
UNION
SELECT company_id AS ID_Value, company_name AS Name_Value
FROM companies
WHERE company_id > 1000
ORDER BY 1;
```

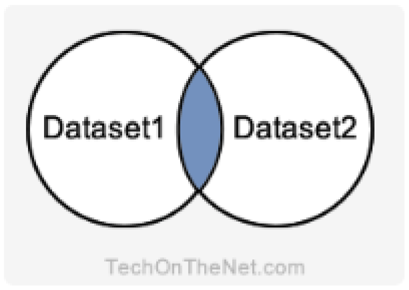
Now the column headings in the result will be aliased as *ID_Value* for the first column and *Name_Value* for the second column.

ID_Value	Name_Value
3000	Apple
4000	Samsung
7000	Sony
8000	IBM

INTERSECTION:

The SQL INTERSECT operator is used to return the results of 2 or more SELECT statements. However, it only returns the rows selected by all queries or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

Intersect Query



Explanation: The INTERSECT query will return the records in the blue shaded area. These are the records that exist in both Dataset1 and Dataset2.

Each SQL statement within the SQL INTERSECT must have the same number of fields in the result sets with similar data types.

NOTE: All rules are same for UNION and INTERSECTION except output.

Examples

Supplier table has following records

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *orders* table populated with the following records:

order_id	order_date	supplier_id
1	2015-08-01	2000

order_id	order_date	supplier_id
2	2015-08-01	6000
3	2015-08-02	7000
4	2015-08-03	8000

And you executed the following UNION statement:

```
SELECT supplier_id FROM suppliers

INTESECT

SELECT supplier_id FROM orders;
```

You would get the following results:

supplier_id
2000

Example 2:

If you had the *suppliers* table populated with the following records:

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *companies* table populated with the following records:

company_id	company_name
1000	Microsoft
3000	Apple
4000	Sony
8000	Samsung

And you executed the following UNION statement:

```
SELECT supplier_id, supplier_name FROM suppliers

INTERSECT

SELECT company_id, company_name FROM companies
```

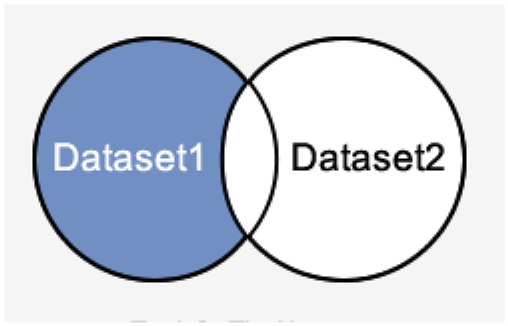
You would get the following results:

supplier_id	supplier_name
3000	Apple
1000	Microsoft

SET DIFFERENCE:

The SQL EXCEPT operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement. Each SELECT statement will define a dataset. The EXCEPT operator will retrieve all records from the first dataset and then remove from the results all records from the second dataset.

Except Query



Explanation: The EXCEPT query will return the records in the blue shaded area. These are the records that exist in Dataset1 and not in Dataset2.
Each SELECT statement within the EXCEPT query must have the same number of fields in the result sets with similar data types.

NOTE: All rules of SET DIFFERENCE and UNION are same except OUTPUT.

Examples:

Supplier table has following records

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *orders* table populated with the following records:

order_id	order_date	supplier_id

order_id	order_date	supplier_id
1	2015-08-01	2000
2	2015-08-01	3000
3	2015-08-02	7000
4	2015-08-03	8000

And you executed the following UNION statement:

```
SELECT supplier_id FROM suppliers

EXCEPT

SELECT supplier_id FROM orders;
```

You would get the following results:

supplier_id
1000
4000

Example 2:

If you had the *suppliers* table populated with the following records:

supplier_id	supplier_name
1000	Microsoft

supplier_id	supplier_name
2000	Oracle
3000	Apple
4000	Samsung

And the *companies* table populated with the following records:

company_id	company_name
1000	Microsoft
3000	Apple
4000	Sony
8000	Samsung

And you executed the following UNION statement:

```
SELECT supplier_id, supplier_name FROM suppliers

EXCEPT

SELECT company_id, company_name FROM companies
```

You would get the following results:

supplier_id	supplier_name
2000	Oracle
4000	Samsung

TASK 2.1

Assume that you have following database relations.
BIIT_Student (CNIC, sName, sAge, sCity, Semester)
NUST_Student (CNIC, sName, sAge, sCity, Semester, phoneNo)
UAAR_Student (CNIC, sName, sAge,CGPA, sCity, Semester)

Write SQL query for the following statements.
NOTE: you must only use UNION / INTERSECTION / SET DIFFERENCE to achieve desired results.

1- Display all CNIC numbers of BIIT students and NUST students. CNIC number must not repeat.

2- Display distinct student names from all above three tables.

3- Display all those students who are studying in BIIT and UAAR at the same time.

4- Display names of those student who are only studying in BIIT but not in NUST.

5- Display all data of students of BIIT who are also enrolled in UAAR and NUST at the same time.

6- Display list of all cities from all tables. (city name must not repeat)

7- Display name of cities which exists in all three tables.

8- Display total number of students of BIIT who are only enrolled in BIIT.

9- Display total number of students who are enrolled in all three universities.

10- Display all those students of NUST who are also enrolled in BIIT but not in UAAR.

LAB # 3

Objectives:

This lab manual will teach students about Cartesian Product.

OUTPUT of Cartesian product of two tables is written as

- Number of rows of both tables are multiplied
- Number of columns of both tables are added.

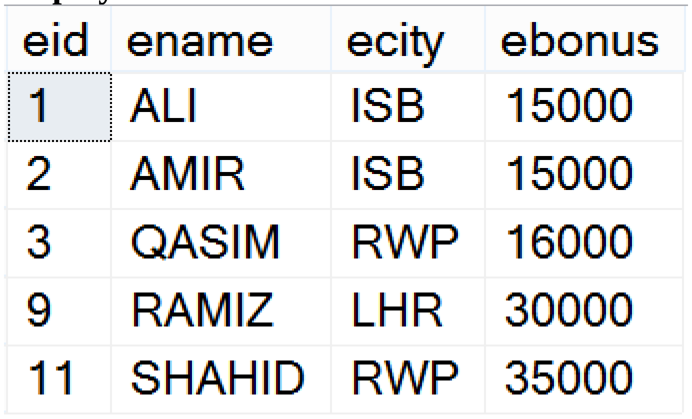
Example:

If number of rows and columns in TABLE1 are 5 and 3 respectively. And number of rows and columns in TABLE 2 are 5 and 4 respectively. Then TABLE1 cross product TABLE2 will have 25 rows and 7 columns in output.

First row of TABLE1 will be written with all rows of TABLE 2.
Second row of TABLE1 will be written with all rows of TABLE 2 and so on.

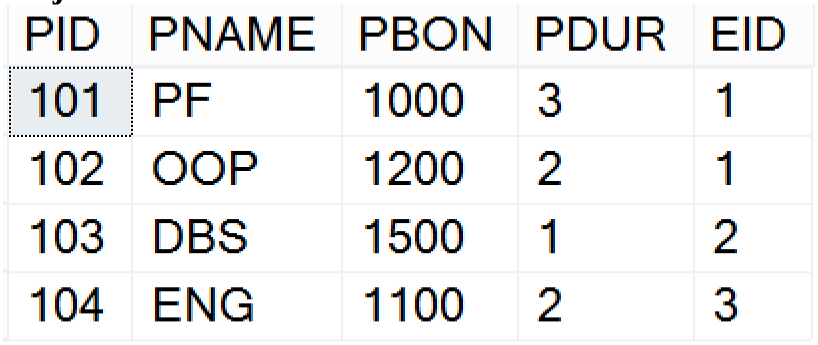
Assume following tables.

Employee



eid	ename	ecity	ebonus
1	ALI	ISB	15000
2	AMIR	ISB	15000
3	QASIM	RWP	16000
9	RAMIZ	LHR	30000
11	SHAHID	RWP	35000

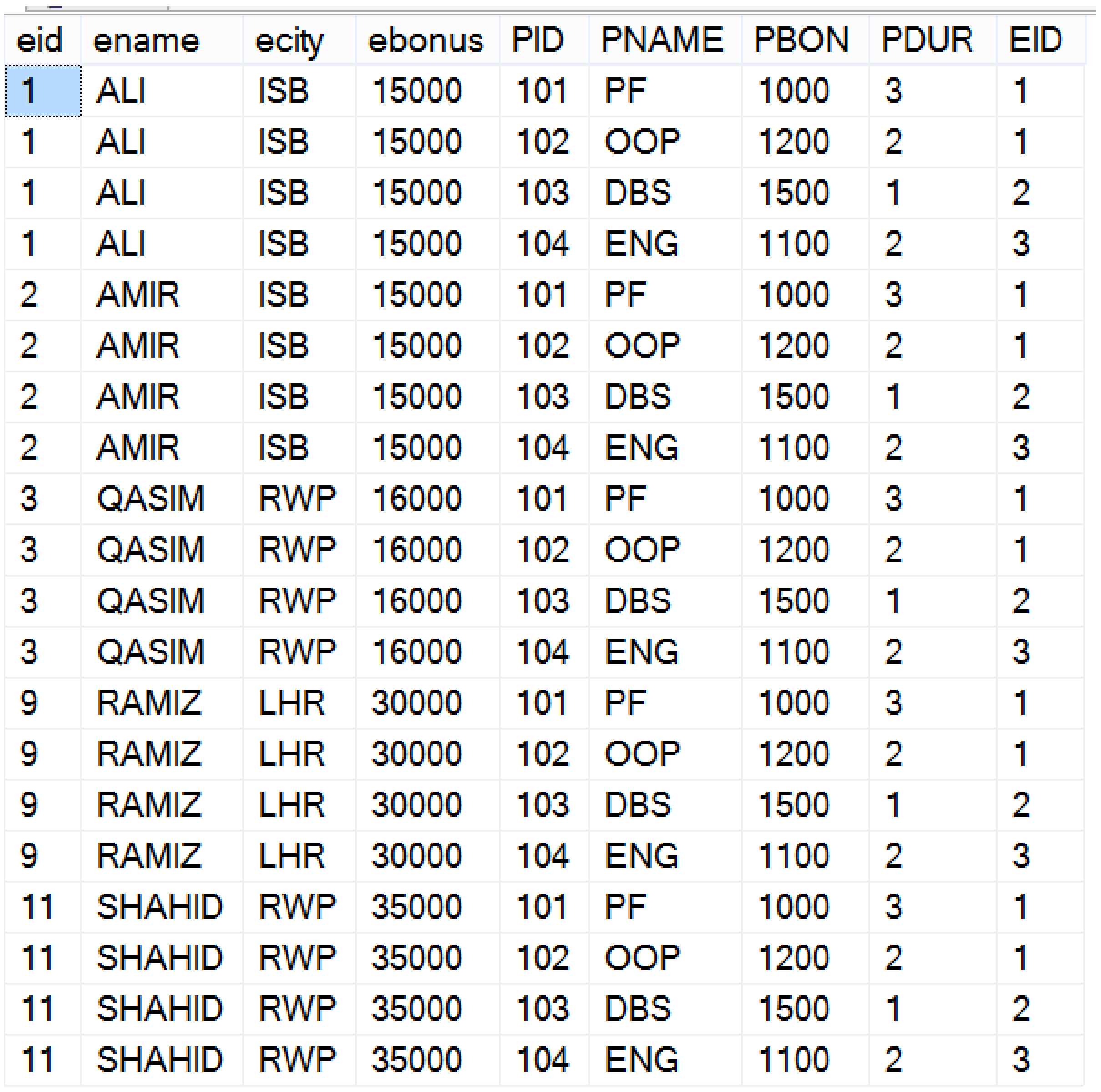
Project:



PID	PNAME	PBON	PDUR	EID
101	PF	1000	3	1
102	OOP	1200	2	1
103	DBS	1500	1	2
104	ENG	1100	2	3

Result of following query is given:

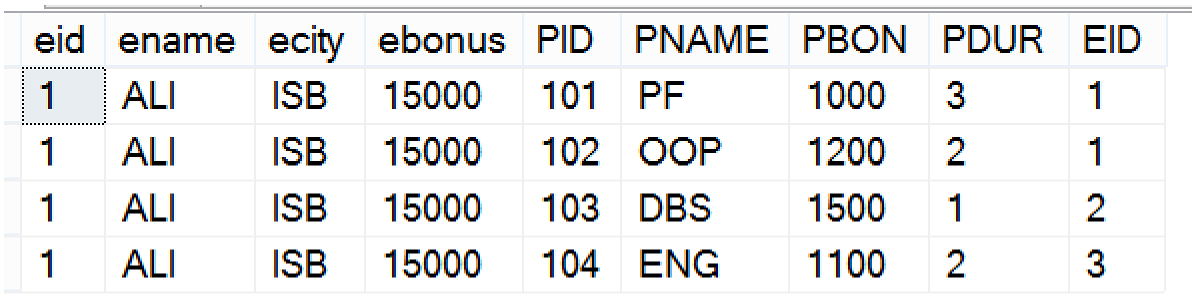
```
SELECT * from Employee , Project ;
```



eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
1	ALI	ISB	15000	103	DBS	1500	1	2
1	ALI	ISB	15000	104	ENG	1100	2	3
2	AMIR	ISB	15000	101	PF	1000	3	1
2	AMIR	ISB	15000	102	OOP	1200	2	1
2	AMIR	ISB	15000	103	DBS	1500	1	2
2	AMIR	ISB	15000	104	ENG	1100	2	3
3	QASIM	RWP	16000	101	PF	1000	3	1
3	QASIM	RWP	16000	102	OOP	1200	2	1
3	QASIM	RWP	16000	103	DBS	1500	1	2
3	QASIM	RWP	16000	104	ENG	1100	2	3
9	RAMIZ	LHR	30000	101	PF	1000	3	1
9	RAMIZ	LHR	30000	102	OOP	1200	2	1
9	RAMIZ	LHR	30000	103	DBS	1500	1	2
9	RAMIZ	LHR	30000	104	ENG	1100	2	3
11	SHAHID	RWP	35000	101	PF	1000	3	1
11	SHAHID	RWP	35000	102	OOP	1200	2	1
11	SHAHID	RWP	35000	103	DBS	1500	1	2
11	SHAHID	RWP	35000	104	ENG	1100	2	3

Example 2:

```
SELECT * from Employee e, Project p where e.ename = 'ali'
```

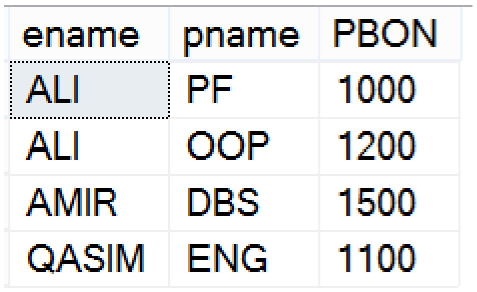


eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
1	ALI	ISB	15000	103	DBS	1500	1	2
1	ALI	ISB	15000	104	ENG	1100	2	3

Example 3:

```
SELECT e.eName, p.pName, p.pBon from Employee e, Project p

where e.eid = p.eid
```



ename	pname	PBON
ALI	PF	1000
ALI	OOP	1200
AMIR	DBS	1500
QASIM	ENG	1100

(above query is displaying names of all those employees who have been assigned any project).

Conclusion:

Cartesian product is used whenever data from multiple tables is required to be viewed.

TASK 3.1

Consider Following Tables.
Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester, Grade)

Write SQL queries for the following statements.
NOTE: you must use “CARTESIAN PRODUCT” query only.

1- Display names of all those students who have “A” grades.

2- Display student name and ccode of all those students who have ‘F’ grade in 5th semester.

3- Display data of all those students who have enrolled at least 10 courses.

4- Display grade of ‘Ali’ in course ‘CS101’.

5- Display all Grades of ‘Irshad’ in 7th semester.

6- Display names of all courses and grades of ‘Amir’.

7- Display list of courses which are taught in 7th semester. (course name must not repeat)

8- Display list of all students by name and course name they have enrolled.

9- Display total number of ‘F’ grade by ‘Amir’.

10- Display all those students who have not enrolled any course yet.

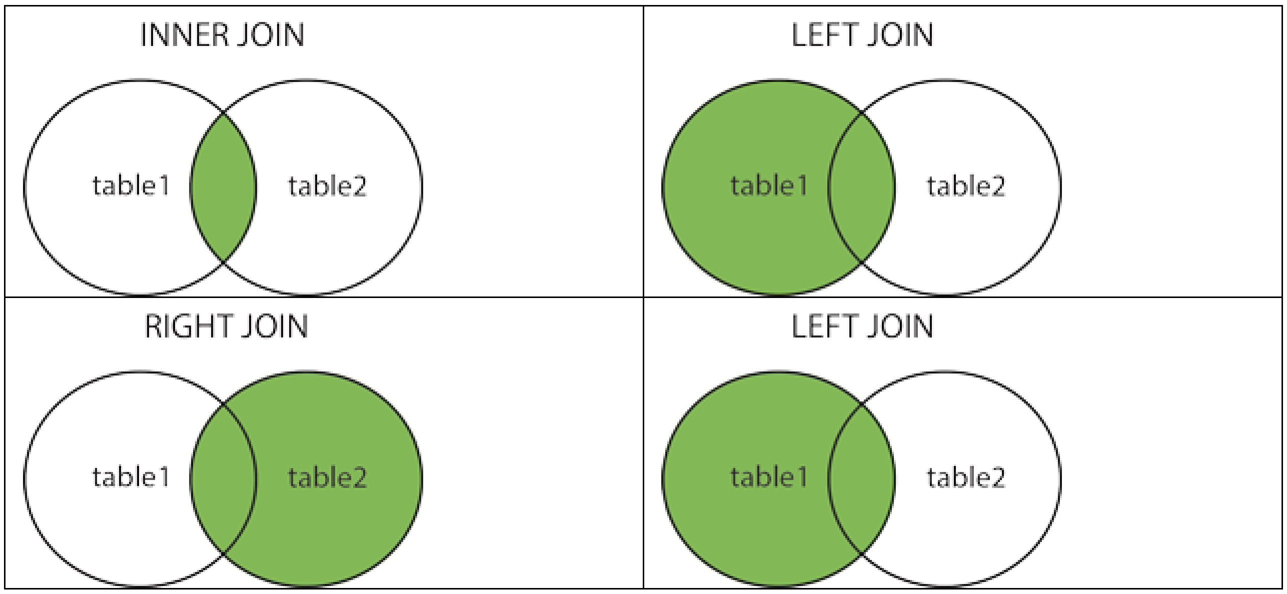
LAB # 4

JOINS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
JOIN condition is written by "ON" keyword. However where can be used but not for joining purpose.

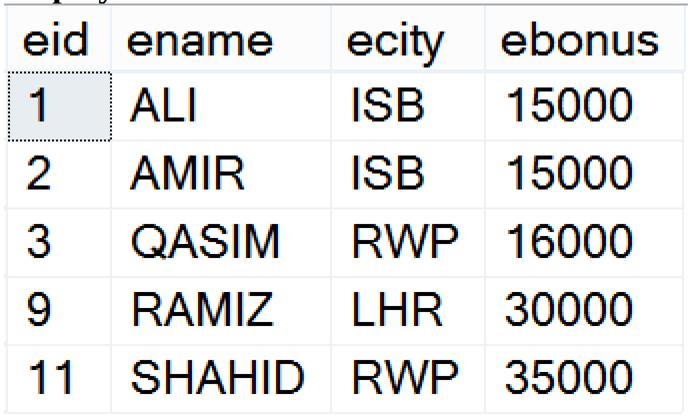
Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table



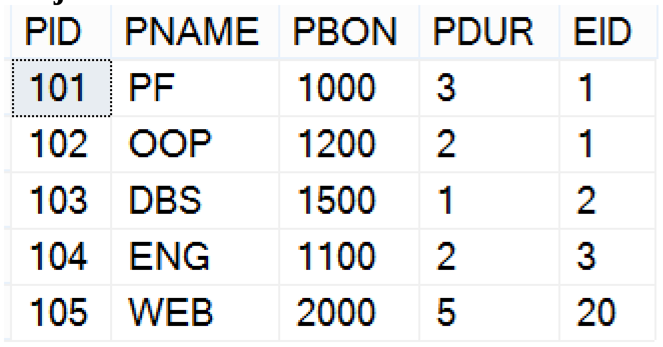
Assume following tables.

Employee



eid	ename	ecity	ebonus
1	ALI	ISB	15000
2	AMIR	ISB	15000
3	QASIM	RWP	16000
9	RAMIZ	LHR	30000
11	SHAHID	RWP	35000

Project:



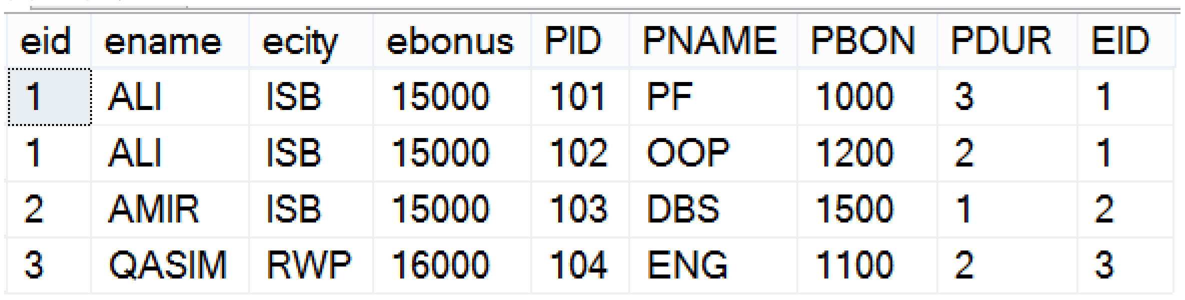
PID	PNAME	PBON	PDUR	EID
101	PF	1000	3	1
102	OOP	1200	2	1
103	DBS	1500	1	2
104	ENG	1100	2	3
105	WEB	2000	5	20

INNER JOIN

It displays only those records of both tables where condition is true.

```
SELECT * from Employee e INNER JOIN Project p ON e.eid = p.eid
```

OUTPUT:



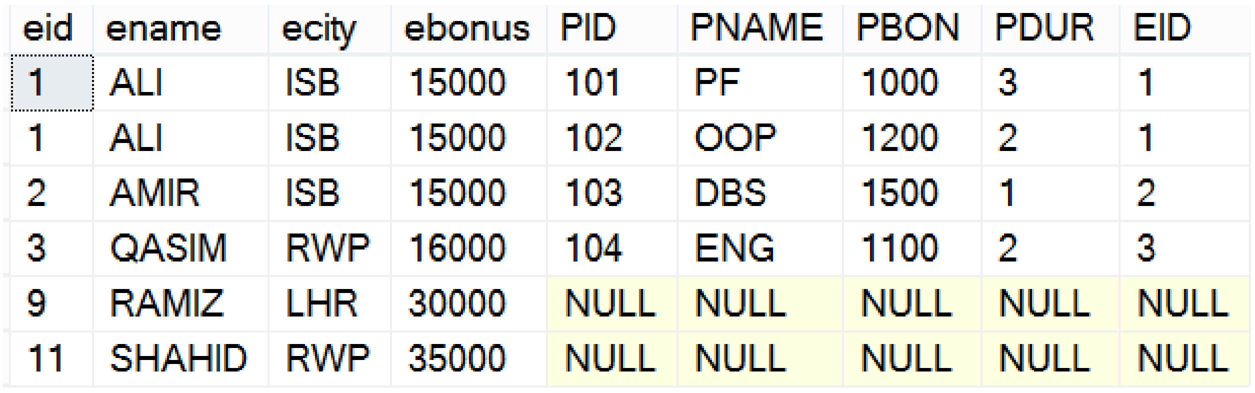
eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
2	AMIR	ISB	15000	103	DBS	1500	1	2
3	QASIM	RWP	16000	104	ENG	1100	2	3

LEFT JOIN

It displays all records of left table and those records of right tables where condition is true.

```
SELECT * from Employee e LEFT JOIN Project p ON e.eid = p.eid
```

OUTPUT:



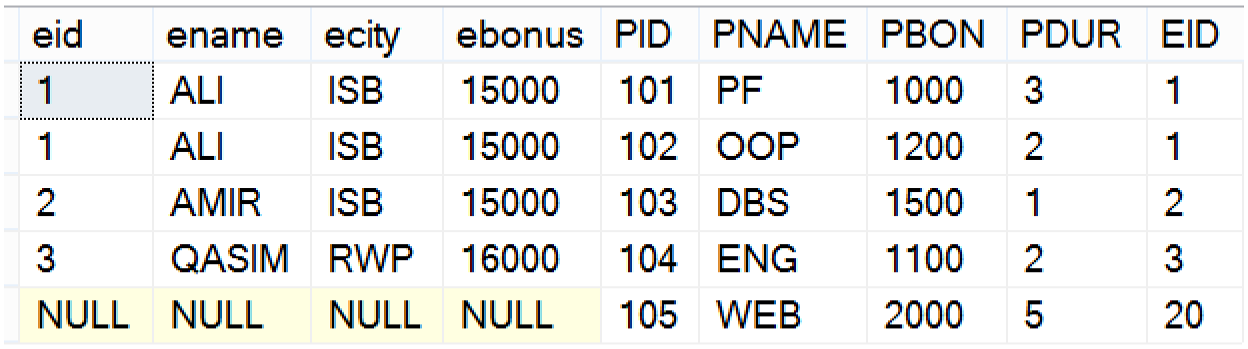
eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
2	AMIR	ISB	15000	103	DBS	1500	1	2
3	QASIM	RWP	16000	104	ENG	1100	2	3
9	RAMIZ	LHR	30000	NULL	NULL	NULL	NULL	NULL
11	SHAHID	RWP	35000	NULL	NULL	NULL	NULL	NULL

RIGHT JOIN:

It displays all records of RIGHT table and those records of LEFT tables where condition is true.

```
SELECT * from Employee e RIGHT JOIN Project p ON e.eid = p.eid
```

OUTPUT:



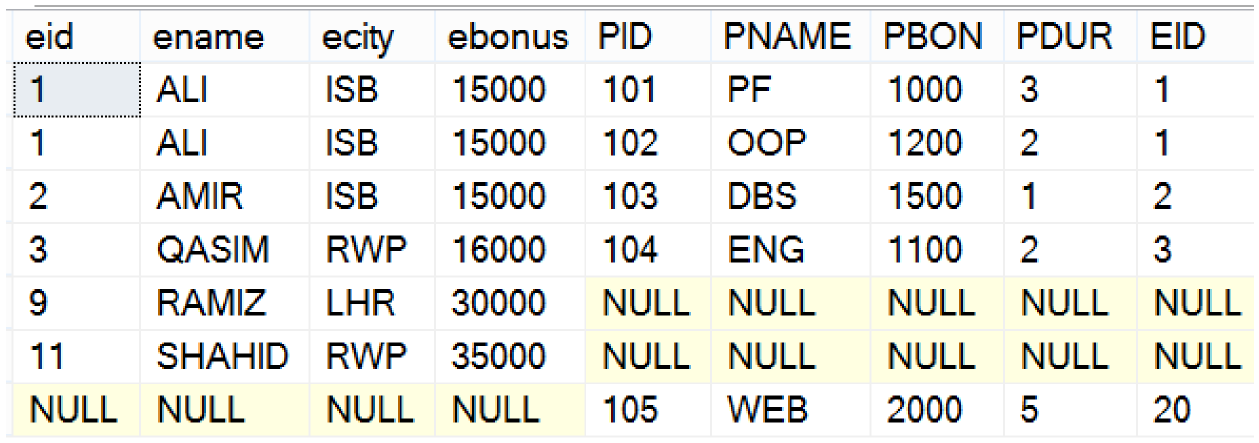
eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
2	AMIR	ISB	15000	103	DBS	1500	1	2
3	QASIM	RWP	16000	104	ENG	1100	2	3
NULL	NULL	NULL	NULL	105	WEB	2000	5	20

FULL OUTER JOIN

It displays all records of BOTH tables.

```
SELECT * from Employee e FULL OUTER JOIN Project p ON e.eid = p.eid
```

OUTPUT:



eid	ename	ecity	ebonus	PID	PNAME	PBON	PDUR	EID
1	ALI	ISB	15000	101	PF	1000	3	1
1	ALI	ISB	15000	102	OOP	1200	2	1
2	AMIR	ISB	15000	103	DBS	1500	1	2
3	QASIM	RWP	16000	104	ENG	1100	2	3
9	RAMIZ	LHR	30000	NULL	NULL	NULL	NULL	NULL
11	SHAHID	RWP	35000	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	105	WEB	2000	5	20

Example Query:

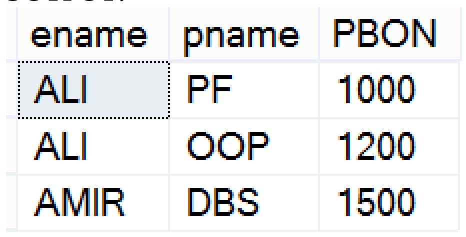
```
select e.ename, p.pname , p.PBON from employee e

inner join project p on e.eid = p.eid

where e.ecity = 'isb'

order by e.ename
```

OUTPUT:



ename	pname	PBON
ALI	PF	1000
ALI	OOP	1200
AMIR	DBS	1500

TASK 4.1

Consider Following Tables.
Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester, Grade)

Write SQL queries for the following statements.
NOTE: you must use “JOINS” query only.

1- Display names of all those students who have “A” grades.

2- Display student name and ccode of all those students who have ‘F’ grade in 5th semester.

3- Display data of all those students who have enrolled at least 10 courses.

4- Display grade of ‘Ali’ in course ‘CS101’.

5- Display all courses of ‘Amir’ along with grade, also display names of courses which are not yet enrolled by ‘Amir’.

6- Display course names along with number of enrolled student in it. If no one has yet enrolled that particular course, it must display ‘0’ in number of enrollment column.

7- Display list of courses which are taught in 7th semester. (course name must not repeat)

8- Display list of all students by name and course name they have enrolled.

9- Display names of students and names of courses they have enrolled. Also display student’s name that have not enrolled any course yet.

10- Display all those students who have not enrolled any course yet.

LAB # 5

Objectives:

- Variable Declaration / Usage / Assignment in SQL
- Condition Statements
- LOOP in SQL
- Stored Procedure

Following is the example of variable declaration with condition statement and loops.

How to Declare a Variable

DECLARE @VNAME DATATYPE
----- variable is declared by "declare" keyword in SQL. Afterwards variable name and then data type is mentioned.
NOTE: variable name must start with "@" symbol.
For example:

```
DECLARE @name varchar(20)
```

In above line of code, @name is name of variable and varchar(20) is data type of variable.

How to Assign Value to Variable

SET @VARIABLE_NAME = VALUE

For example:

```
DECLARE @NAME VARCHAR(20) , @AGE INT
SET @NAME = 'SHAH NAWAZ'
SET @AGE = 34
PRINT @NAME
PRINT @AGE
```

OUTPUT:

```
SHAH NAWAZ
34
```

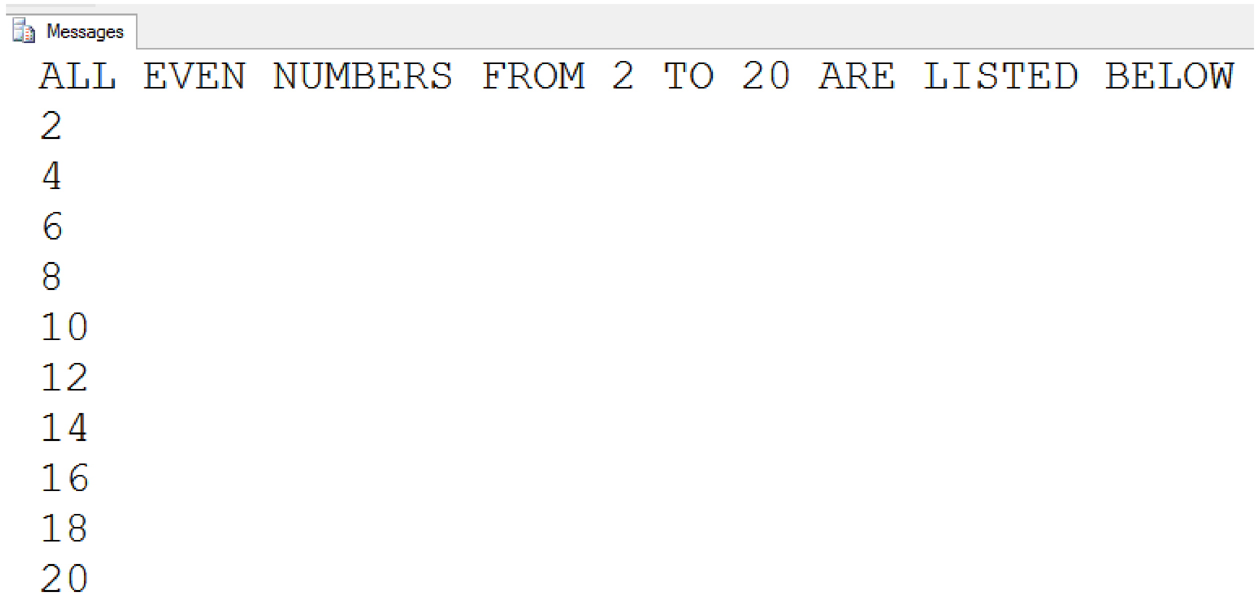
"Print statement displays data by default in new line".

Following is the Example of Using condition statement and LOOP in SQL:

```
DECLARE @COUNT INT
SET @COUNT = 1
PRINT 'ALL EVEN NUMBERS FROM 2 TO 20 ARE LISTED BELOW'
```

```
WHILE (@COUNT <= 20)
BEGIN
        IF(@COUNT % 2 = 0)
        BEGIN
              PRINT @COUNT
        END
        SET @COUNT = @COUNT + 1
END
```

OUTPUT:


Messages

```
ALL EVEN NUMBERS FROM 2 TO 20 ARE LISTED BELOW
2
4
6
8
10
12
14
16
18
20
```

STORED PROCEDURE:

- Stored procedures are just like functions in programming.
- Used to store query to save time of writing query again and again.
- It reduces chances of errors.
- Once created, stored procedures remains in database until dropped.

General Syntax to Create Stored Procedures:

```
CREATE PROC procedure_name
( @arguments if any)
AS
SQL statement 1
SQL statement 2
```

SQL statement ‘N’

How to Call a Stored Procedure:

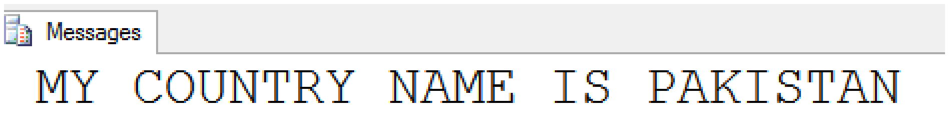
EXEC procedure_name

Example 1:

```
CREATE PROC PRINT_NAME
AS
PRINT 'MY COUNTRY NAME IS PAKISTAN'

-- EXECUTING PROCEDURE
EXEC PRINT_NAME
```

OUTPUT:



```
MY COUNTRY NAME IS PAKISTAN
```

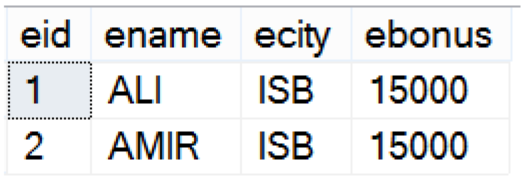
Example 2:

```
CREATE PROC ISB_EMPLOYEES
AS
BEGIN
        SELECT * FROM EMPLOYEE WHERE ECITY = 'ISB'
END

-- EXECUTING PROCEDURE
EXEC ISB_EMPLOYEES
```

NOTE: begin / end keywords are used to define BODY of procedure / condition statement / loops. It is optional to define body of procedure.

OUTPUT:



eid	ename	ecity	ebonus
1	ALI	ISB	15000
2	AMIR	ISB	15000

TASK #5.1

Write SQL queries for the following statements.

1- Write SQL code which declares two variables for name and address. Store your name and address in respective variables. Display name and address in a single line using print statement.

2- Display table of 10 in given format using loop.
10 x 1 = 10
10 x 2 = 20
.
.
.
10 x 10 = 100

3- Write a code which declares a variable and stores age of youngest employee from "Employee" table. And display name of youngest employee 5 times his age.

4- Assume that in “Employee” table, Emp_id are assigned from 1 to onward. ie first employee’s eid is 1, second employee’s eid is 2 and so on. Write a code that displays name of eldest employee without using aggregate function of “max”. You can use count function once.

LAB# 6

Objectives:

- Stored procedures with arguments

General Syntax:

```
CREATE PROC NAME_PROC
(@ARGUMENT1 DATATYPE,
@ARGUMENT2 DATATYPE,
@ARGUMENT_ N DATATYPE)
AS
BEGIN
SQL STATEMENT 1
SQL STATEMENT 2
SQL STATEMENT N
END
```

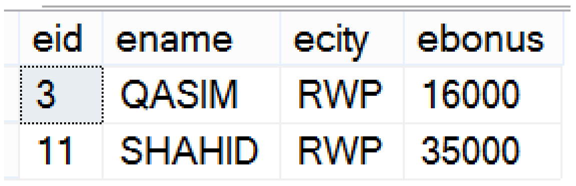
Example 1:

```
CREATE PROC NAMES
(
 @CITY VARCHAR(20)
)
AS
```

```
BEGIN
        SELECT * FROM EMPLOYEE WHERE ECITY = @CITY
END

-- EXECUTING PROCEDURE
EXEC NAMES 'RWP'
```

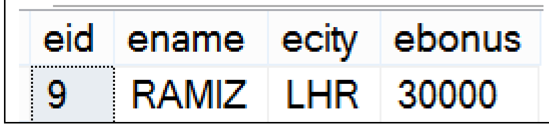
OUTPUT:



eid	ename	ecity	ebonus
3	QASIM	RWP	16000
11	SHAHID	RWP	35000

```
-- EXECUTING PROCEDURE
EXEC NAMES 'LHR'
```

OUTPUT:



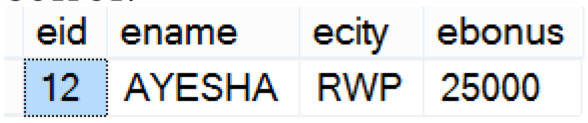
eid	ename	ecity	ebonus
9	RAMIZ	LHR	30000

Example 2:

```
CREATE PROC INS_EMP
(
        @EID INT, @ENAME VARCHAR(20),
        @CITY VARCHAR(20), @BONUS INT
)
AS
BEGIN
        INSERT INTO EMPLOYEE VALUES (@EID, @ENAME , @CITY, @BONUS)
        SELECT * FROM EMPLOYEE WHERE EID = @EID
END

-- EXECUTING PROCEDURE
EXEC INS_EMP 12, 'AYESHA', 'RWP', 25000
```

OUTPUT:



eid	ename	ecity	ebonus
12	AYESHA	RWP	25000

Details:
On calling of stored procedure “INS_EMP”, it stored values in “employee” table and then displayed

values inserted by user.

Example 3:

```
CREATE PROC INS_EMP_CHK_PRIMARY
(
        @EID INT, @ENAME VARCHAR(20),
        @CITY VARCHAR(20), @BONUS INT
)
AS
BEGIN
        DECLARE @COUNT INT;
        SET @COUNT = (SELECT COUNT(*) FROM EMPLOYEE WHERE EID = @EID)
        IF( @COUNT = 0)
                INSERT INTO EMPLOYEE VALUES (@EID, @ENAME , @CITY, @BONUS)
        ELSE
                PRINT 'PRIMARY KEY VIOLATION'
        SELECT * FROM EMPLOYEE WHERE EID = @EID
END

-- EXECUTING PROCEDURE
EXEC INS_EMP 12, 'AYESHA', 'RWP', 25000
```

- Example 3 procedure will check for primary key constraint as well. If eid already exists, procedure will print ‘PRIMARY KEY VIOLATION’

TASK 6.1

Write SQL queries for the following statements.

1- Write a stored procedure which takes an integer value as argument from user and displays its table up to 10 in given format.
For example: if user send 7 as an argument then,

   7 x 1 = 7
   7 x 2 = 14
   ……
   7 x 10 = 70

2- Write a stored procedure which takes two arguments from user, display table of first argument from 1 to value of second argument.

Consider Following Tables.
Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester, Grade)

3- Write a stored procedure which insert values in Student table and make sure following constraints must not violate. (do not apply constraint using SQL server management studio)
   a. regNo must not repeat
   b. sCGPA must be between 0.0 and 4.0
   c. sAge must be between 18 and 25

4- Write a stored procedure which inserts values in Enrollment table and make sure following constraint must not violate.
   a. regNo and cCode combines form primary key. Primary key rule must not violate.
   b. regNo and cCode are also foreign keys from Student and Course tables respectively, make sure foreign key constraint must not violate. Only those values must be entered which exists in parent tables.

5- Write a Stored procedure which takes regno from user as an argument and displays name of student and grades of student in each course he/she has enrolled.

LAB # 7

Objectives:

- Learning stored procedures with output parameters

OUTPUT Parameter in Stored Procedure:

If you specify the OUTPUT keyword for a parameter in the procedure definition, the stored procedure can return the current value of the parameter to the calling program when the stored procedure exits. To save the value of the parameter in a variable that can be used in the calling program, the calling program must use the OUTPUT keyword when executing the stored procedure.

- OUTPUT parameter returns its value to the calling variable before exiting.
- User must write OUTPUT at the time of defining argument and at the time of calling to user returned value.

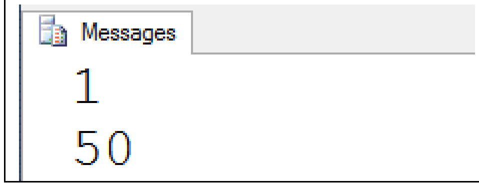
Example 1:

```
CREATE PROC RET_VAL
(
	@V INT OUTPUT
)
AS
BEGIN
	SET @V = 50
END

-- EXECUTING PROCEDURE
```

```
DECLARE @VAL INT
SET @VAL = 1
PRINT @VAL
EXEC RET_VAL @VAL OUTPUT
PRINT @VAL
```

OUTPUT:



```
1
50
```

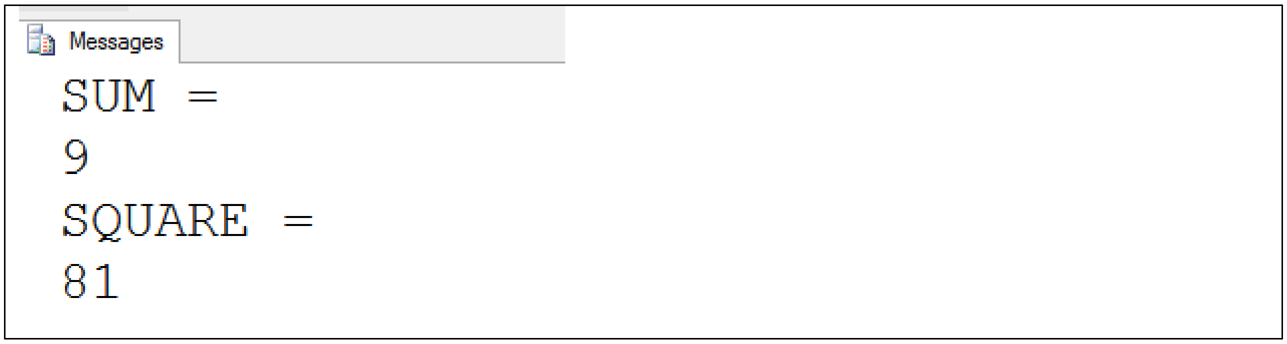
Details:
In above stored procedure, an argument is defined as OUTPUT. And its value is set to 50. During calling of procedure, a variable is declared and assigned value = 1 and printed. After calling, value of same variable is printed which is "50".

Example 2:

```
CREATE PROC GET_SUM
(@VAL1 INT, @VAL2 INT, @SUM INT OUTPUT)
AS
BEGIN
            SET @SUM = @VAL1 + @VAL2
END
--------------------------------------------------------
CREATE PROC GET_SQUARE
( @VAL INT, @SQ INT OUTPUT)
AS
BEGIN
            SET @SQ = @VAL * @VAL
END
-------------------------------------------------------

DECLARE @SUM INT, @SQUARE INT
EXEC GET_SUM 4, 5, @SUM OUTPUT
EXEC GET_SQUARE @SUM, @SQUARE OUTPUT
PRINT 'SUM = '; PRINT @SUM
PRINT 'SQUARE = '; PRINT @SQUARE
```

OUTPUT:



```
SUM =
9
SQUARE =
81
```

Details:
In above two stored procedures "GET_SUM" and "GET_SQUARE", output parameters are used to return values after calculations.
"GET_SUM" stored procedure takes three arguments, one of them is output parameters which will return sum of first two.
"GET_SQUARE" stored procedure takes two arguments and return square of first value.

TASK # 7.1

Write SQL queries for the given statements.
Consider following Database Schema.
Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester,quality_points,Grade)

1- Write a stored procedure which takes 3 arguments from user and return maximum of them. Display maximum value at calling place.

2- Write a stored procedure which returns age and name of student whose registration number is ‘101’.

3- Write following stored procedures
    a. Return_Quality_Points
    b. Return_Total_Credit_hours
    c. Update_CGPA

- Return_Quality_Points procedure will take regNo of student and return total quality points achieved by him/her.
- Return_Total_Credit_hours procedure will take regNo of student and return total credit hours enrolled by him/her.
- Update_CGPA will take regNO, total quality points, total credit hours, calculate CGPA and update in student table against student’s data.

4- Write following stored procedures
    a. Return_Limit_Courses
    b. Insert_Student_Enrollment

- Return_Limit_Courses will take regNo and semester no as argument and return “yes” or “no”. Procedure will return “yes”, if total credit hours enrolled in that current semester are less 18. Otherwise return “no”.
- Insert_Student_Enrollment will take enrollment data from user and returned value from above procedure as well. If value returned is “yes”, than data must be inserted in enrollment table, otherwise print ‘credit hours exceeded’.

LAB #8

Objectives:

- Learn about Transactions
- How to implement transactions in SQL

Transaction:

"sequence of operations performed as a single logical unit of work".
A transaction has four key properties that are abbreviated ACID. ACID is an acronym for Atomic, Consistent, Isolated and Durability.

Syntax:

```
BEGIN TRAN
-- SQL STATEMENT 1
-- SQL STATEMENT 2
-- SQL STATEMENT 3
-- SQL STATEMENT N
COMMIT / ROLLBACK
```

Example 1:

```
BEGIN TRAN

        update employee
        set ebonus = 50000
        where eid=101

        update employee
        set ebonus = 23000
        where eid=102

COMMIT
```

Details:

In above example, begin tran will start a transaction. First update query will update bonus of employee 101. Second update query will update bonus of employee 102. Here if user thinks that there is an error or he/she wants to revert all changes performed in transaction. He/she will right "ROLLBACK" statement instead of "COMMIT". ROLLBACK will revert all the changes made by both update queries. On other hand, COMMIT statement will make changes permanent.

Example 2:

```
BEGIN TRAN
INSERT INTO EMPLOYEE
 (EID, ENAME, EAGE, EBSAL) VALUES (1,'AMMAD',32,23000)

DECLARE @E INT
SET @E = @@ERROR

UPDATE EMPLOYEE
SET  EBON = EBONUS + 2000

IF @E <> 0
        ROLLBACK
ELSE
        COMMIT
```

Details:

In above example, user wants to perform two changes in database.

1- **Insert**
2- **Update**

For a transaction, both queries must execute, if one of fail then transaction must "ROLLBACK" so that database remains unchanged. Insert query can return error in case of "PRIMARY KEY VIOLATION". In case insert query returns an error, the global variable of @@ERROR will have value other then "ZERO".

In Above example, after executing insert statement, value of variable @@ERROR is stored in variable @E so that it can be used later. If @E has value other than “ZERO”, it means that value is not inserted. In this case update query must also not execute. If @E has value other than “ZERO”, “ROLLBACK” statement will execute and remove all changes performed by system.

Task #8.1

Consider following database schema and write SQL query for each of the statement given below.

Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester,quality_points,Grade)

1- Write a procedure which must insert values in Student table. After insertion, you must check whether inserted value is stored more than once or not. If count of inserted value exists more than once, system must ‘ROLLBACK’ and inserted value must be deleted by system.

2- Write a procedure which must perform following task
   a. Insets 6 records in Enrollment table (each record for the course enrolled in current semester).
   b. If any of insert statement fails due to any constraint (primary key, foreign key, check), all inserted data must be reverted by using “ROLLBACK”

LAB # 9

Objectives:

- Triggers
- AFTER Triggers with example

Triggers:

- Are like functions
- Called by system
  - On change on database
    - INSERT
    - UPDATE
    - DELETE
- User cannot call triggers
- Triggers do not have arguments

Types of Triggers:

1. AFTER (INSERT / UPDATE / DELETE)
2. INSTEAD OF (INSERT / UPDATE / DELETE)

AFTER TRIGGER:

- AFTER trigger is called by system after the action is performed.

GENERAL SYNTAX:

```
CREATE TRIGGER T_NAME
ON <TABLE NAME>
AFTER <EVENT>
AS
BEGIN
      SQL STATEMENT 1
      SQL STATEMENT 2
      SQL STATEMENT N

END
```

Example 1:

```
CREATE TRIGGER TNAME ON EMP
AFTER INSERT
AS
BEGIN
        SELECT * FROM EMP
END
```

Details:
Above trigger will be called by system whenever data is inserted in “EMP” table of database. In body of trigger a single select statement is written which will show table of employee.

Example 2:

```
CREATE TRIGGER SH_UPDATE ON EMP
AFTER UPDATE
AS
BEGIN
        PRINT 'DATA HAS BEEN UPDATED'
END
```

Details:
Above trigger will be executed whenever data is updated in “EMP” table. And “PRINT” statement will execute after updating data.

Example 3:

```
CREATE TRIGGER DEL ON EMP
AFTER DELETE
AS
```

```
BEGIN
        SELECT COUNT(*) AS 'REMAING RECORDS' FROM EMP
END
```

Details:
Above trigger will be executed whenever data is deleted from “EMP” table. Trigger will display number of remaining records in “EMP” table after delete.

TASK 9.1

Consider given database schema and write triggers for each statement.

Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester,quality_points,Grade)

1- A trigger which display all data of student table whenever a record is inserted in Student table.

2- A trigger which deletes all data from course table whenever total number of records in "Course" table reaches 10.

3- A trigger which displays all data of "Enrollment" table whenever a record is deleted from "Enrollment" table.

4- A trigger which displays data of "Student" table on insertion of every 10th record.

5- A trigger that checks age of Student after insertion, delete all those employees whose age is above 25.

LAB #10

Objectives:

- Instead of Trigger
- Temporary tables in Triggers

INSTEAD OF TRIGGERS:

- Skips original action (INSERT / UPDATE DELETE), instead of execute queries which are written inside the trigger.
- For example if an insert trigger is called, instead of inserting data in table, system will call the trigger and execute queries written in trigger.

Example 1:

```
CREATE TRIGGER INSERT_EMPLOYEE ON EMP
INSTEAD OF INSERT
AS
BEGIN
	SELECT * FROM EMP
END
```

Details:
Above query will not allow user to insert data in "EMP". System will display EMP table instead of inserting data in table.

Example 2:

```
CREATE TRIGGER DONT_DELETE ON EMP
INSTEAD OF DELETE
AS
BEGIN
	PRINT 'YOU CANNOT DELETE DATA'
END
```

Details:
Above query will never allow user to delete any data from "EMP" table ever. Instead of deleting data from table, system will call the trigger which will display "YOU CANNOT DELETE DATA".

Temporary Tables in Triggers:

Temporary tables used in triggers and Database table for which trigger is called have same design (same columns).

There are two temporary tables used in triggers.
1- INSERTED
2- DELETED

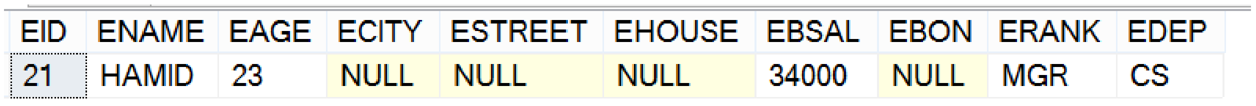
- The deleted table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the deleted table. The deleted table and the trigger table ordinarily have no rows in common.
- The inserted table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added to both the inserted table and the trigger table. The rows in the inserted table are copies of the new rows in the trigger table.
- An update transaction is similar to a delete operation followed by an insert operation; the old rows are copied to the deleted table first, and then the new rows are copied to the trigger table and to the inserted table.

Example 1:

```
CREATE TRIGGER SHOW_INSERTED
ON EMP AFTER INSERT
AS
BEGIN
        SELECT * FROM inserted
END

---- inserting data in emp table
INSERT INTO EMP (EID, ENAME , EAGE , EBSAL, ERANK)
VALUES  (21, 'HAMID', 23, 34000, 'MGR')
```

OUTPUT:



EID	ENAME	EAGE	ECITY	ESTREET	EHOUSE	EBSAL	EBON	ERANK	EDEP
21	HAMID	23	NULL	NULL	NULL	34000	NULL	MGR	CS

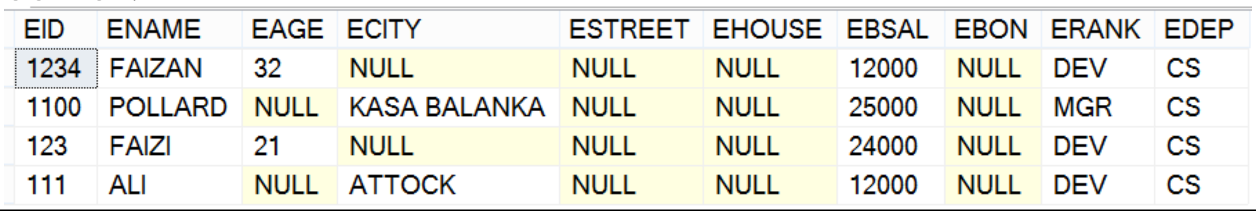
Details:
Above trigger is created on "EMP" table, whenever data is inserted in "EMP" table, "SHOW_INSERTED" trigger will be called by system. In body of trigger, all data of "inserted" table is displayed. You can see OUTPUT of trigger. Effected rows copied to inserted table which is shown in the output.

Example 2:

```
CREATE TRIGGER SHOW_DELETED
ON EMP AFTER DELETE
AS
BEGIN
        SELECT * FROM deleted
END

DELETE EMP WHERE EID > 100
```

OUTPUT:



EID	ENAME	EAGE	ECITY	ESTREET	EHOUSE	EBSAL	EBON	ERANK	EDEP
1234	FAIZAN	32	NULL	NULL	NULL	12000	NULL	DEV	CS
1100	POLLARD	NULL	KASA BALANKA	NULL	NULL	25000	NULL	MGR	CS
123	FAIZI	21	NULL	NULL	NULL	24000	NULL	DEV	CS
111	ALI	NULL	ATTOCK	NULL	NULL	12000	NULL	DEV	CS

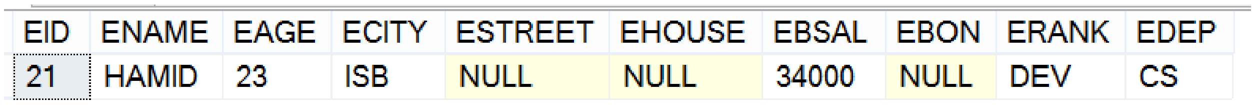
Details:
You can see only those values are copied to deleted table which are effected by delete query.

Example 3:

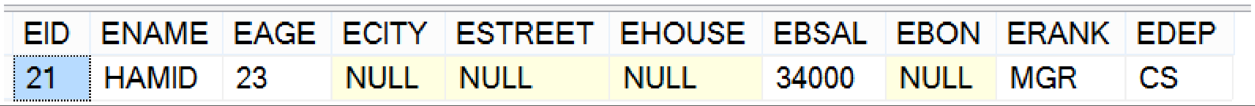
```
CREATE TRIGGER SHOW_UPDATED
ON EMP AFTER UPDATE
AS
BEGIN
	SELECT * FROM inserted
	SELECT * FROM deleted
END

UPDATE EMP
SET ECITY = 'ISB', ERANK = 'DEV'
WHERE EID = 21

OUTPUT:
```



EID	ENAME	EAGE	ECITY	ESTREET	EHOUSE	EBSAL	EBON	ERANK	EDEP
21	HAMID	23	ISB	NULL	NULL	34000	NULL	DEV	CS



EID	ENAME	EAGE	ECITY	ESTREET	EHOUSE	EBSAL	EBON	ERANK	EDEP
21	HAMID	23	NULL	NULL	NULL	34000	NULL	MGR	CS

Details:
You can see, inserted table showing all values after update, which are inserted in the table and deleted table showing all the values which are deleted from table. Basically delete table has all the values before update and inserted table has all the values after update.

TASK # 10.1

Consider given database schema and write triggers for each statement.

Student (regNo, sName, sCity, sCGPA, sAge)
Student_BACKUP(regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester,quality_points,Grade)
Enrollment_BACKUP(regNo, cCode, semester,quality_points,Grade)

1- Whenever a record is inserted in student table, system must check following constraints in inserted value.
   a. sAge between 18 and 25
   b. sCGPA between 0.0 and 4.0

   if any of constraints are violated, system must delete record inserted by user.

2- Write a mechanism which should not allow user to update Grade in Enrollment table. Other columns of Enrollment table can be updated.

3- Write a mechanism which should copy those record to “Enrollment_BACKUP” whose GRADE is ‘F’, whenever a record is inserted in Enrollment Table.

4- Write a mechanism which stores all deleted record to Student_BACKUP table whenever a record is deleted from Student table.

5- Write a trigger which must make sure primary key is note violated in course table whenever a record is inserted in course table.

LAB # 11

CASE STUDY TRIGGERS:

BIIT software house wants to maintain a database of its employees, projects assigned to employees and accounts. Database schema is given below.

1- **EMPLOYEE**(eid, eName, eAge, eCity, eRank, eDep)
2- **PROJECT** (pid, pName, pBonusAmout)
3- **Project_Allocation** (eid, pid)
4- **ACCOUNTS** (eid, Year, Month, Basic_Salary, Project_Bonus)
5- **ACCOUNTS_BACKUP**(eid, Year, Month, Basic_Salary, Project_Bonus)
6- **LOG**(srNo, table_name , Date, Time, event, no_of_rows_effected)

Perform following tasks

a- Whenever a record is inserted or updated from ACCOUNTS table, it must also be inserted / updated in ACCOUNT_BACKUP table as well. Do not allow user to delete any record from ACCOUNTS or ACCOUNTS_BACKUP table. On insertion in ACCOUNTS table, Project_Bonus will be calculated by system on the basis of projects assigned to employee.

b- Make sure primary foreign keys must not violate in any table. You must use triggers to apply any type of constraints. All keys are mentioned at the end.

c- LOG table will maintain every activity of Database. Whenever a record is inserted, updated or deleted in database. All data regarding event will be stored in LOG table.

LAB #12

Objectives:

- Learning about cursors in SQL

Cursor is a database object to retrieve data from a result set one row at a time, instead of the T-SQL commands that operate on all the rows in the result set at one time. We use cursor when we need to update records in a database table in singleton fashion means row by row.

Life Cycle of Cursor

1. **Declare Cursor**

A cursor is declared by defining the SQL statement that returns a result set.

2. **Open**

A Cursor is opened and populated by executing the SQL statement defined by the cursor.

3. **Fetch**

When cursor is opened, rows can be fetched from the cursor one by one or in a block to do data manipulation.

4. **Close**

After data manipulation, we should close the cursor explicitly.

5. Deallocate

Finally, we need to delete the cursor definition and released all the system resources associated with the cursor.

Syntax to Declare Cursor

Declare Cursor SQL Command is used to define the cursor with many options that impact the scalability and loading behavior of the cursor. The basic syntax is given below

1. DECLARE cursor_name CURSOR
2. [LOCAL | GLOBAL] --define cursor scope
3. [FORWARD_ONLY | SCROLL] --define cursor movements (forward/backward)
4. [STATIC | KEYSET | DYNAMIC | FAST_FORWARD] --basic type of cursor
5. [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC] --define locks
6. FOR select_statement --define SQL Select statement
7. FOR UPDATE [col1,col2,...coln] --define columns that need to be updated

Syntax to Open Cursor

A Cursor can be opened locally or globally. By default it is opened locally. The basic syntax to open cursor is given below:

1. OPEN [GLOBAL] cursor_name --by default it is local

Syntax to Fetch Cursor

Fetch statement provides the many options to retrieve the rows from the cursor. NEXT is the default option. The basic syntax to fetch cursor is given below:

1. FETCH [NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n]
2. FROM [GLOBAL] cursor_name
3. INTO @Variable_name[1,2,..n]

Syntax to Close Cursor

Close statement closed the cursor explicitly. The basic syntax to close cursor is given below:

1. CLOSE cursor_name --after closing it can be reopen

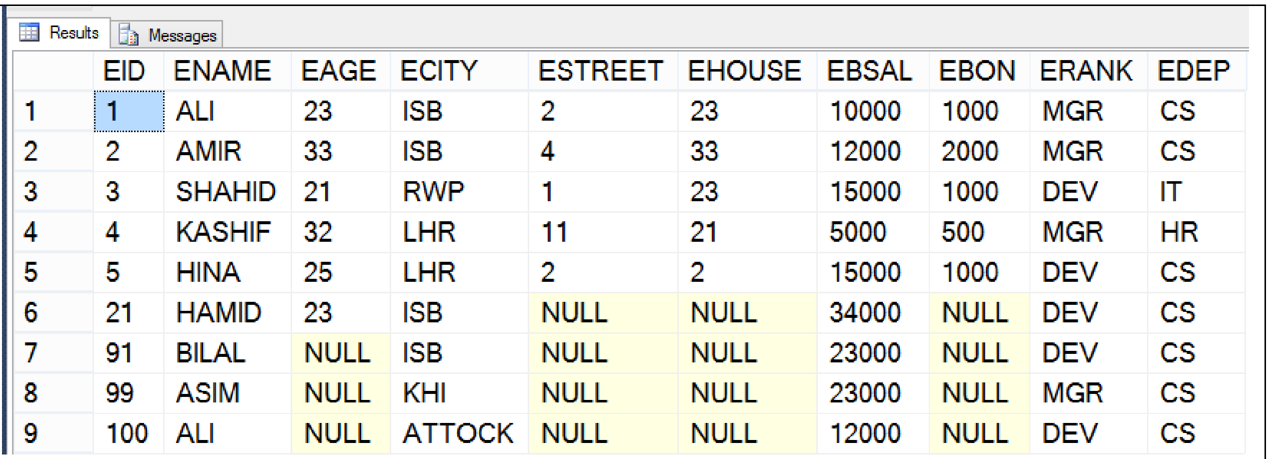
Syntax to Deallocate Cursor

Deallocate statement delete the cursor definition and free all the system resources associated with the cursor. The basic syntax to close cursor is given below:

1. DEALLOCATE cursor_name --after deallocation it can't be reopen

Example:

Given is Table of Emp:

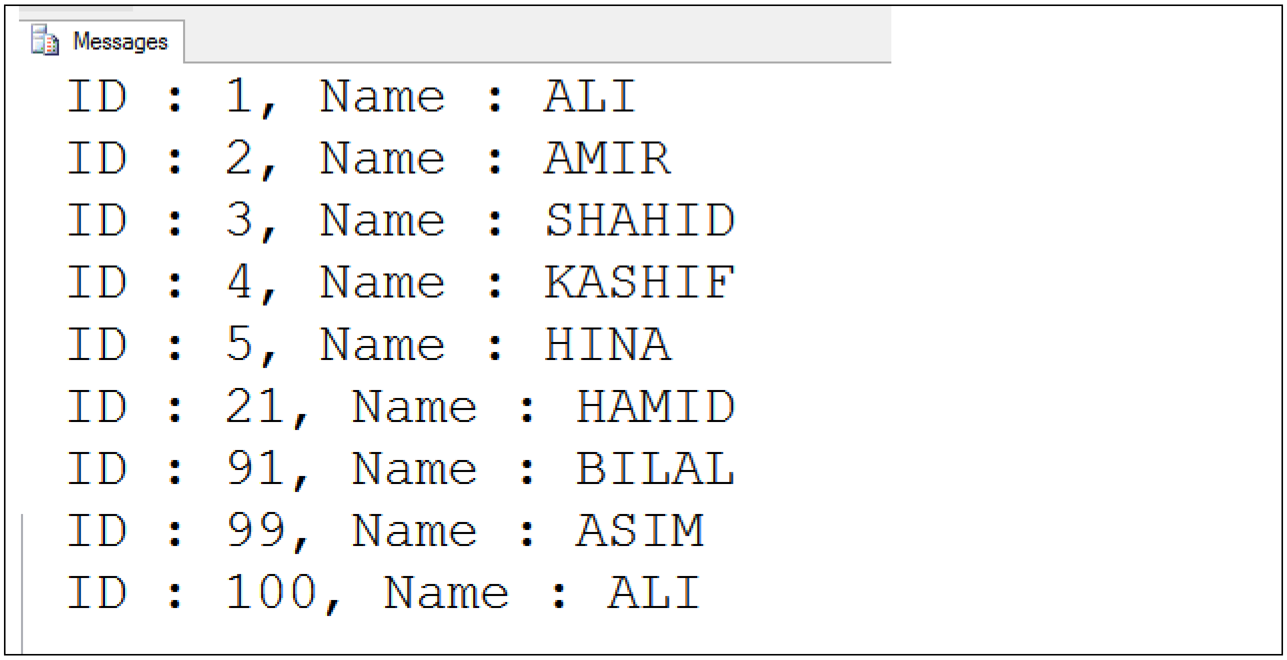


Results | Messages

	EID	ENAME	EAGE	ECITY	ESTREET	EHOUSE	EBSAL	EBON	ERANK	EDEP
1	1	ALI	23	ISB	2	23	10000	1000	MGR	CS
2	2	AMIR	33	ISB	4	33	12000	2000	MGR	CS
3	3	SHAHID	21	RWP	1	23	15000	1000	DEV	IT
4	4	KASHIF	32	LHR	11	21	5000	500	MGR	HR
5	5	HINA	25	LHR	2	2	15000	1000	DEV	CS
6	21	HAMID	23	ISB	NULL	NULL	34000	NULL	DEV	CS
7	91	BILAL	NULL	ISB	NULL	NULL	23000	NULL	DEV	CS
8	99	ASIM	NULL	KHI	NULL	NULL	23000	NULL	MGR	CS
9	100	ALI	NULL	ATTOCK	NULL	NULL	12000	NULL	DEV	CS

```
DECLARE @Id int
DECLARE @name varchar(50)
 DECLARE cur_emp CURSOR
STATIC FOR
SELECT  eid, ENAME from Emp
OPEN cur_emp
IF @@CURSOR_ROWS > 0
 BEGIN
 FETCH NEXT FROM cur_emp INTO @Id,@name
 WHILE @@Fetch_status = 0
 BEGIN
 PRINT 'ID : '+ convert(varchar(20),@Id)+', Name : '+@name
 FETCH NEXT FROM cur_emp INTO @Id,@name
 END
END
CLOSE cur_emp
DEALLOCATE cur_emp
```

OUTPUT:


Messages

```
ID : 1, Name : ALI
ID : 2, Name : AMIR
ID : 3, Name : SHAHID
ID : 4, Name : KASHIF
ID : 5, Name : HINA
ID : 21, Name : HAMID
ID : 91, Name : BILAL
ID : 99, Name : ASIM
ID : 100, Name : ALI
```

TASK 12.1

Consider the following schema and write SQL query for given statements.

Student (regNo, sName, sCity, sCGPA, sAge)
Course (cCode, cName, credit_hours)
Enrollment (regNo, cCode, semester,quality_points,Grade)

1- Display 2nd row of Student table using Cursors.

2- Display all records of Enrollment table which are placed at even rows.

3- Execute given query in SQL and do as directed.
   - UPDATE STUDENT SET ECITY = ‘ISB’ WHERE EID>101
   - Count how many records have been updated in above query using cursor.

4- Write a mechanism which must not allow user to delete more than 1 records using cursor and transaction.

LAB # 13

- PRESENTATIONS

LAB # 14

- LAB PROJECT

LAB # 15

- LAB PROJECT