Database Management Systems Lab

Subject Code : 18CS4SP04L
Credits : 01 **L-T-P : 0-0-2**

List of Experiments:

1. Create User in Oracle Database and grant and revoke the privileges and use of commit savepoint and rollback command.
2. Create the following:

   (a) Synonym sequences and Index

   (b) Create alter and update views.
3. Create PL/SQL program using cursors, control structure, exception handling
4. Create following:

   (a) Simple Triggers

   (b) Package using procedures and functions.
5. Create the table for

   (a) COMPANY database

   (b) STUDENT database and Insert five records for each attribute.
6. Illustrate the use of SELECT statement
7. Conditional retrieval - WHERE clause
8. Query sorted - ORDER BY clause
9. Perform following:

   (a) UNION, INTERSECTION and MINUS operations on tables.

   (b) UPDATE, ALTER, DELETE, DROP operations on tables
10. Query multiple tables using JOIN operation.
11. Grouping the result of query - GROUP BY clause and HAVING clause
12. Query multiple tables using NATURAL and OUTER JOIN operation.

1. **Create User in Oracle Database and grant and revoke the privileges, and use of commit savepoint and roleback command.**

plsql_online_lab_across_sections

Data Control Language (DCL) Statements DDL, DML, DCL (COMMIT, ROLLBACK, SAVEPOINT, GRANT, REVOKE)

Any relational database must be able to pass the ACID test: it must guarantee atomicity, consistency, isolation, and durability. The principle of atomicity states that all parts of a transaction must complete or none of them. The principle of consistency states that the results of a query must be consistent with the state of the database at the time the query started. The principle of isolation states that an incomplete (that is, uncommitted) transaction must be invisible to the rest of the world. The principle of durability states that once a transaction completes, it must be impossible for the database to lose it.

After DML commands we have to commit transaction in order for the changes to take effect in the DB.

TRANSACTIONS

COMMIT: to explicitly save the database state

TO undo the changes we have made with the COMMIT statement we can use

ROLLBACK

We can also make certain points in our code that will let us ROLLBACK only to that point.

SAVEPOINT name_of_savepoint;

SAVEPOINT BEFORELABPROGRM;

Student, dept, 5 rows each, RESTART; COMMIT;

SAVEPOINT POGRAMLAB;

ROLLBACK TO BEFORELABPROGRM;

ROLLBACK TO name_of_savepoint;

FOR UPDATE

There is an option to do select on some table and lock it for changes, so that we can do update on it.

SELECT * FROM table_name FOR UPDATE;

PRIVILEGES

GRANT: gives user and roles rights and privileges on database objects or schema.

GRANT privilege_name TO user_name_or_role_name;

Privileges:
SESSION – allows user to connect to the DB
RESOURCE – allows user to work with objects on his schema
SELECT ANY TABLE
INSERT ANY TABLE
UPDATE ANY TABLE
DELETE ANY TABLE

We can also use the command to GRANT a role to a user

GRANT role_name TO user_name;

REVOKE: removes or restricts user rights or privileges on database objects.

Data Control Language Statements are used to grant privileges on tables, views, sequences, synonyms, procedures to other users or roles.

The DCL statements are

GRANT: Use to grant privileges to other users or roles.
REVOKE: Use to take back privileges granted to other users and roles.

Privileges are of two types :

- System Privileges
- Object privileges

System Privileges are normally granted by a DBA to users. Examples of system privileges are CREATE SESSION, CREATE TABLE, CREATE USER etc.

Object privileges means privileges on objects such as tables, views, synonyms, procedure. These are granted by owner of the object.

Object Privileges are

ALTER	Change the table definition with the ALTER TABLE statement.
DELETE	Remove rows from the table with the DELETE statement. Note: You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.

SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement.
	Note: You must grant the SELECT privilege on the table along with the UPDATE privilege.

Grant

Grant is use to grant privileges on tables, view, procedure to other users or roles

Examples

Suppose you own emp table. Now you want to grant select,update,insert privilege on this table to other user “SAMI”.

grant select, update, insert on emp to sami;

Suppose you want to grant all privileges on emp table to sami. Then

grant all on emp to sami;

Suppose you want to grant select privilege on emp to all other users of the database. Then

grant select on emp to public;

Suppose you want to grant update and insert privilege on only certain columns not on all the columns then include the column names in grant statement. For example you want to grant update privilege on ename column only and insert privilege on empno and ename columns only. Then give the following statement

grant update (ename),insert (empno, ename) on emp to sami;

To grant select statement on emp table to sami and to make sami be able further pass on this privilege you have to give WITH GRANT OPTION clause in GRANT statement like this.

grant select on emp to sami with grant option;

REVOKE

Use to revoke privileges already granted to other users.

For example to revoke select, update, insert privilege you have granted to Sami then give the following statement.

revoke select, update, insert on emp from sami;

To revoke select statement on emp granted to public give the following command.

revoke select on emp from public;

To revoke update privilege on ename column and insert privilege on empno and ename columns give the following revoke statement.

revoke update, insert on emp from sami;

Note :You cannot take back column level privileges. Suppose you just want to take back insert privilege on ename column then you have to first take back the whole insert privilege and then grant privilege on empno column.

ROLES

A role is a group of Privileges. A role is very handy in managing privileges, Particularly in such situation when number of users should have the same set of privileges.

For example you have four users :Sami, Scott, Ashi, Tanya in the database. To these users you want to grant select ,update privilege on emp table, select,delete privilege on dept table. To do this first create a role by giving the following statement

create role clerks

Then grant privileges to this role.

grant select,update on emp to clerks;
grant select,delete on dept to clerks;

Now grant this clerks role to users like this

grant clerks to sami, scott, ashi, tanya ;

Now Sami, Scott, Ashi and Tanya have all the privileges granted on clerks role.

Suppose after one month you want grant delete on privilege on emp table all these users then just grant this privilege to clerks role and automatically all the users will have the privilege.

grant delete on emp to clerks;

If you want to take back update privilege on emp table from these users just take it back from clerks role.

revoke update on emp from clerks;

To Drop a role

Drop role clerks;

LISTING INFORMATION ABOUT PRIVILEGES

To see which table privileges are granted by you to other users.

SELECT * FROM USER_TAB_PRIVS_MADE

To see which table privileges are granted to you by other users

SELECT * FROM USER_TAB_PRIVS_RECD;

To see which column level privileges are granted by you to other users.

SELECT * FROM USER_COL_PRIVS_MADE

To see which column level privileges are granted to you by other users

SELECT * FROM USER_COL_PRIVS_RECD;

To see which privileges are granted to roles

SELECT * FROM USER_ROLE_PRIVS;

Queries:

Tables Used: Consider the following tables namely “DEPARTMENTS” and “EMPLOYEES”

Their schemas are as follows , Departments (dept _no , dept_ name , dept_location); Employees (emp_id , emp_name , emp_salary);

Q1: Develop a query to grant all privileges of employees table into departments table

Ans: SQL> Grant all on employees to departments;

Grant succeeded.

Q2: Develop a query to grant some privileges of employees table into departments table

Ans: SQL> Grant select, update , insert on departments to departments with grant option;

Grant succeeded.

Q3: Develop a query to revoke all privileges of employees table from departments table

Ans: SQL> Revoke all on employees from departments;

Revoke succeeded.

Q4: Develop a query to revoke some privileges of employees table from departments table

Ans: SQL> Revoke select, update , insert on departments from departments;

Revoke succeeded.

Q5: Write a query to implement the save point

Ans: SQL> SAVEPOINT S1;

Savepoint created.

SQL> select * from emp;

EMPNO ENAME JOB DEPTNO SAL

---------- -------------------- ------------- ---------- ----------

1 Mathi AP 1 10000

2 Arjun ASP 2 15000

3 Gugan ASP 1 15000

4 Karthik Prof 2 30000

SQL> INSERT INTO EMP VALUES(5,'Akalya','AP',1,10000); 1 row created.

SQL> select * from emp;

EMPNO ENAME JOB DEPTNO SAL

---------- -------------------- ------------- ---------- ----------

1 Mathi AP 1 10000

2 Arjun ASP 2 15000

3 Gugan ASP 1 15000

4 Karthik Prof 2 30000

5 Akalya AP 1 10000

Q6: Write a query to implement the rollback

Ans: SQL> rollback s1;

SQL> select * from emp;

EMPNO ENAME JOB DEPTNO SAL

---------- -------------------- ------------- ---------- ----------

1 Mathi AP 1 10000

2 Arjun ASP 2 15000

3 Gugan ASP 1 15000

4 Karthik Prof 2 30000 CS1032

Q6: Write a query to implement the commit

Ans: SQL> COMMIT;

Commit complete.

2. Create the following:

(a) Synonym, sequences and Index

(b) Create alter and update views.

SEQUENCES

A sequence is used to generate numbers in sequence. You can use sequences to insert unique values in Primary Key and Unique Key columns of tables. To create a sequence gives the CREATE SEQUENCE statement.

CREATING SEQUENCES

```
create sequence bills
   start with 1
   increment by 1
   minvalue 1
   maxvalue 100
   cycle
   cache 10;
```

The above statement creates a sequence bills it will start with 1 and increment by 1. It's maxvalue is 100 i.e. after 100 numbers are generated it will stop if you say NOCYCLE, otherwise if you mention CYCLE then again it will start with no. 1. You can also specify NOMAXVALUE in that case the sequence will generate infinite numbers.

The CACHE option is used to cache sequence numbers in System Global Area (SGA). If you say CACHE 10 then Oracle will cache next 10 numbers in SGA. If you access a sequence number then oracle will first try to get the number from cache, if it is not found then it reads the next number from disk. Since reading the disk is time consuming rather than reading from SGA it is always recommended to cache sequence numbers in SGA. If you say NOCACHE then Oracle will not cache any numbers in SGA and every time you access the sequence number it reads the number from disk.

Accessing Sequence Numbers.

To generate Sequence Numbers you can use NEXTVAL and CURRVAL for example to get the next sequence number of bills sequence type the following command.

Select bills.nextval from dual;

```
BILLS
-----
1
```

NEXTVAL gives the next number in sequence. Whereas, CURRVAL returns the current number of the sequence. This is very handy in situations where you have insert records in Master Detail tables. For example to insert a record in SALES master table and SALES_DETAILS detail table.

insert into sales (billno,custname,amt) values (bills.nextval,'Sami',2300);

insert into sales_details (billno,itemname,qty,rate) values (bills.currval,'Onida',10,13400);

Sequences are usually used as DEFAULT Values for table columns to automatically insert unique numbers.
For Example,

```
create table invoices (invoice_no number(10) default bills.nextval,
    invoice_date date default sysdate,
    customer varchar2(100),
    invoice_amt number(12,2));
```

Now whenever you insert rows into invoices table ommiting invoice_no as follows

```
insert into invoices (customer,invoice_amt) values    ('A to Z Traders',5000);
```

Oracle will insert invoice_no from bills sequence

ALTERING SEQUENCES

To alter sequences use ALTER SEQUENCE statement. For example to alter the bill sequence MAXVALUE give the following command.

```
ALTER SEQUENCE BILLS
   MAXVALUE 200;
```

Except Starting Value, you can alter any other parameter of a sequence. To change START WITH parameter you have to drop and recreate the sequence.

DROPPING SEQUENCES

To drop sequences use DROP SEQUENCE command. For example to drop bills sequence give the following statement

```
drop sequence bills;
```

SYNONYMS

A synonym is an alias for a table, view, snapshot, sequence, procedure, function, or package.

There are two types to SYNONYMS they are

PUBLIC SYNONYM
PRIVATE SYNONYM

If you a create a synonym as public then it can be accessed by any other user with qualifying the synonym name i.e. the user doesn't have to mention the owner name while accessing the synonym. Nevertheless the other user should have proper privilege to access the synonym. Private synonyms needs to be qualified with owner names.

CREATING SYNONYMS

To create a synonym for SCOTT emp table give the following command.

```
create synonym employee for scott.emp;
```

A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced. For example, if a synonym named EMPLOYEE refers to a table or view, then the following statement is valid:

select * from employee;

Suppose you have created a function known as TODAY which returns the current date and time. Now you have granted execute permission on it to every other user of the database. Now these users can execute this function but when the call they have to give the following command:

select scott.today from dual;

Now if you create a public synonym on it then other users don't have to qualify the function name with owner's name. To define a public synonym give the following command.

create public synonym today for scott.today;

Now the other users can simply type the following command to access the function.

select today from dual;

Dropping Synonyms

To drop a synonym use the DROP SYNONYM statement. For example, to drop EMPLOYEE synonym give the statement

drop synonym employee;

Listing information about synonyms

To see synonyms information give the following statement.

select * from user_synonyms;

INDEXES

Use indexes to speed up queries. Indexes speeds up searching of information in tables. So create indexes on those columns which are frequently used in WHERE conditions. Indexes are helpful if the operations return only small portion of data i.e. less than 15% of data is retrieved from tables.

Follow these guidelines for creating indexes

- Do not create indexes on small tables i.e. where number of rows are less. (Full table scan itself will be faster if table is small)
- Do not create indexes on those columns which contain many null values.
- Do not create BTree index on those columns which contain many repeated values. In this case create BITMAP indexes on these columns.

- Limit the number of indexes on tables because, although they speed up queries, but at the same time DML operations becomes very slow as all the indexes have to updated whenever an Update, Delete or Insert takes place on tables.

Creating Indexes

To create an Index give the create index command. For example the following statement creates an index on empno column of emp table.

create index empno_ind on emp (empno);

If two columns are frequently used together in WHERE conditions then create a composite index on these columns. For example, suppose we use EMPNO and DEPTNO oftenly together in WHERE condition. Then create a composite index on these column as given below

create index empdept_ind on emp (empno,deptno);

The above index will be used whenever you use empno or deptno column together, or you just use empno column in WHERE condition. The above index will not be used if you use only deptno column alone

BITMAP INDEXES

Create Bitmap indexes on those columns which contains many repeated values and when tables are large. City column in EMP table is a good canditate for Bitmap index because it contain many repeated values. To create a composite index give the following command.

create bitmap index city_ind on emp (city);

FUNCTION BASED Indexes

Function Based indexes are built on expressions rather than on column values. For example if you frequently use the expression SAL+COMM in WHERE conditions then create a Function base index on this expression like this

create index salcomm_ind on emp (sal+comm);

Now, whenever you use the expression SAL+COMM in where condition then oracle will use SALCOMM_IND index.

DROPPING INDEXES

To drop indexes use DROP INDEX statement. For example to drop SALCOMM_IND give the following statement

drop index salcomm_ind;

Listing Information about indexes

To see how many indexes are there in your schema and its information, give the following statement.

select * from user_indexes;

Views

Views are known as logical tables. They represent the data of one of more tables. A view derives its data from the tables on which it is based. These tables are called base tables. Views can be based on actual tables or another view also.

Whatever DML operations you performed on a view they actually affect the base table of the view. You can treat views same as any other table. You can Query, Insert, Update and delete from views, just as any other table.

Views are very powerful and handy since they can be treated just like any other table but do not occupy the space of a table.

The following sections explain how to create, replace, and drop views using SQL commands.

Creating Views

Suppose we have EMP and DEPT table. To see the empno, ename, sal, deptno, department name and location we have to give a join query like this.

```
select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc
    From emp e, dept d where e.deptno=d.deptno;
```

So everytime we want to see emp details and department names where they are working we have to give a long join query. Instead of giving this join query again and again, we can create a view on these table by using a CREATE VIEW command given below

```
create view emp_det as select e.empno,
e.ename,e.sal,e.deptno,d.dname,d.loc
    from emp e, dept d where e.deptno=d.deptno;
```

Now to see the employee details and department names we don't have to give a join query, we can just type the following simple query.

```
select * from emp_det;
```

This will show same result as you have type the long join query. Now you can treat this EMP_DET view same as any other table.

For example, suppose all the employee working in Department No. 10 belongs to accounts department and most of the time you deal with these people. So every time you have to give a DML or Select statement you have to give a WHERE condition like .....WHERE DEPTNO=10. To avoid this, you can create a view as given below

```
CREATE VIEW accounts_staff AS
  SELECT Empno, Ename, Deptno
  FROM Emp
```

```
  WHERE Deptno = 10
  WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Now to see the account people you don't have to give a query with where condition you can just type the following query.

```
select * from accounts_staff;
select sum(sal) from accounst_staff;
select max(sal) from accounts_staff;
```

Replacing/Altering Views

To alter the definition of a view, you must replace the view using one of the following methods:

- A view can be dropped and then re-created. When a view is dropped, all grants of corresponding view privileges are revoked from roles and users. After the view is re-created, necessary privileges must be regranted.
- A view can be replaced by redefining it with a CREATE VIEW statement that contains the OR REPLACE option. This option replaces the current definition of a view, but preserves the present security authorizations.

For example, assume that you create the ACCOUNTS_STAFF view, as given in a previous example. You also grant several object privileges to roles and other users. However, now you realize that you must redefine the ACCOUNTS_STAFF view to correct the department number specified in the WHERE clause of the defining query, because it should have been 30. To preserve the grants of object privileges that you have made, you can replace the current version of the ACCOUNTS_STAFF view with the following statement:

```
CREATE OR REPLACE VIEW Accounts_staff AS
  SELECT Empno, Ename, Deptno
  FROM Emp
  WHERE Deptno = 30
  WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Replacing a view has the following effects:

- Replacing a view replaces the view's definition in the data dictionary. All underlying objects referenced by the view are not affected.
- If previously defined but not included in the new view definition, then the constraint associated with the WITH CHECK OPTION for a view's definition is dropped.
- All views and PL/SQL program units dependent on a replaced view become invalid.

With some restrictions, rows can be inserted into, updated in, or deleted from a base table using a view. The following statement inserts a new row into the EMP table using the ACCOUNTS_STAFF view:

```
INSERT INTO Accounts_staff
  VALUES (199, 'ABID', 30);
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.

2. If a view is defined with WITH CHECK OPTION, then a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.

3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.

4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

The constraint created by WITH CHECK OPTION of the ACCOUNTS_STAFF view only allows rows that have a department number of 10 to be inserted into, or updated in, the EMP table. Alternatively, assume that the ACCOUNTS_STAFF view is defined by the following statement (that is, excluding the DEPTNO column):

```
CREATE VIEW Accounts_staff AS
   SELECT Empno, Ename
   FROM Emp
   WHERE Deptno = 10
   WITH CHECK OPTION CONSTRAINT ica_Accounts_cnst;
```

Considering this view definition, you can update the EMPNO or ENAME fields of existing records, but you cannot insert rows into the EMP table through the ACCOUNTS_STAFF view because the view does not let you alter the DEPTNO field. If you had defined a DEFAULT value of 10 on the DEPTNO field, then you could perform inserts.

If you don’t want any DML operations to be performed on views, create them WITH READ ONLY option. Then no DML operations are allowed on views.

Modifying a Join View

Oracle allows you, with some restrictions, to modify views that involve joins. Consider the following simple view:

```
CREATE VIEW Emp_view AS
   SELECT Ename, Empno, deptno FROM Emp;
```

This view does not involve a join operation. If you issue the SQL statement:

```
UPDATE Emp_view SET Ename = 'SHAHRYAR' WHERE Empno = 109;
```

then the EMP base table that underlies the view changes, and employee 109's name changes from ASHI to SHAHRYAR in the EMP table.

However, if you create a view that involves a join operation, such as:

```
CREATE VIEW Emp_dept_view AS
  SELECT e.Empno, e.Ename, e.Deptno, e.Sal, d.Dname, d.Loc
    FROM Emp e, Dept d    /* JOIN operation */
     WHERE e.Deptno = d.Deptno
       AND d.Loc IN ('HYD', 'BOM', 'DEL');
```

then there are restrictions on modifying either the EMP or the DEPT base table through this view.

A modifiable join view is a view that contains more than one table in the top-level FROM clause of the SELECT statement, and that does not contain any of the following:

- DISTINCT operator
- Aggregate functions: AVG, COUNT, GLB, MAX, MIN, STDDEV, SUM, or VARIANCE
- Set operations: UNION, UNION ALL, INTERSECT, MINUS
- GROUP BY or HAVING clauses
- START WITH or CONNECT BY clauses
- ROWNUM pseudocolumn

Any UPDATE, INSERT, or DELETE statement on a join view can modify only one underlying base table. The following example shows an UPDATE statement that successfully modifies the EMP_DEPT_VIEW view:

```
UPDATE Emp_dept_view
  SET Sal = Sal * 1.10
    WHERE Deptno = 10;
```

The following UPDATE statement would be disallowed on the EMP_DEPT_VIEW view:

```
UPDATE Emp_dept_view
  SET Loc = 'BOM'
    WHERE Ename = 'SAMI';
```

This statement fails with an ORA-01779 error ("cannot modify a column which maps to a non key-preserved table"), because it attempts to modify the underlying DEPT table, and the DEPT table is not key preserved in the EMP_DEPT view.

3. Create PL/SQL program using cursors, control structure, exception handling.

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

PL/SQL is not a standalone programming language; it is a tool within the Oracle programming environment. **SQL* Plus** is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.

To run PL/SQL programs, you should have the Oracle RDBMS Server installed in your machine. This will take care of the execution of the SQL commands. The most recent version of Oracle RDBMS is 11g.

Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −

S.No	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block −

```
DECLARE
  <declarations section>
BEGIN
```

```
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

The 'Hello World' Example

```
DECLARE
  message  varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result −

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

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.
```

CURSOR PROGRAM FOR ELECTRICITY BILL CALCULATION:

```
SQL> create table bill(name varchar2(10), address varchar2(20), city varchar2(20), unit
number(10));
Table created.
SQL> insert into bill values('&name','&addess','&city','&unit');
Enter value for name: yuva
Enter value for addess: srivi
Enter value for city: srivilliputur
```

```
Enter value for unit: 100
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('yuva','srivi','srivilliputur','100')
1 row created.
SQL> /
Enter value for name: nithya
Enter value for addess: Lakshmi nagar
Enter value for city: sivakasi
Enter value for unit: 200
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('nithya','Lakshmi nagar','sivakasi','200')
1 row created.
SQL> /
Enter value for name: maya
Enter value for addess: housing board
Enter value for city: sivakasi
Enter value for unit: 300
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('maya','housing board','sivakasi','300')
1 row created.
SQL> /
Enter value for name: jeeva
Enter value for addess: RRR nagar
Enter value for city: sivaganagai
Enter value for unit: 400
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('jeeva','RRR nagar','sivaganagai','400')
1 row created.
SQL> select * from bill;
NAME ADDRESS CITY UNIT
---------- -------------------- -------------------- ---------
yuva srivi srivilliputur 100
nithya Lakshmi nagar sivakasi 200
maya housing board sivakasi 300
jeeva RRR nagar sivaganagai 400

SQL> declare
2 cursor c is select * from bill;
3 b bill %ROWTYPE;
4 begin
5 open c;
6 dbms_output.put_line('Name Address city Unit Amount');
7 loop
8 fetch c into b;
```

```
9 if(c % notfound) then
10 exit;
11 else
12 if(b.unit<=100) then
13 dbms_output.put_line(b.name||' '||b.address||' '||b.city||' '||b.unit||' '||b.uni t*1);
14 elsif(b.unit>100 and b.unit<=200) then
15 dbms_output.put_line(b.name||' '||b.address||' '||b.city||' '||b.unit||' '||b.unit*2);
16 elsif(b.unit>200 and b.unit<=300) then
17 dbms_output.put_line(b.name||' '||b.address||' '||b.city||' '||b.unit||' '||b.unit*3);
18 elsif(b.unit>300 and b.unit<=400) then
19 dbms_output.put_line(b.name||' '||b.address||' '||b.city||' '||b.unit||' '||b.unit*4);
20 else
21 dbms_output.put_line(b.name||' '||b.address||' '||b.city||' '||b.unit||' '||b.unit*5);
22 end if;
23 end if;
24 end loop;
25 close c;
26 end;
27 /
Name Address city Unit Amount
yuva srivi srivilliputur 100 100
nithya Lakshmi nagar sivakasi 200 400
maya housing board sivakasi 300 900
jeeva RRR nagar sivaganagai 400 1600
PL/SQL procedure successfully completed.
```

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;
```

4. Create following:

(a) Simple Triggers

(b) Package using procedures and functions.

SIMPLE TRIGGER FOR DISPLAYING GRADE OF THE STUDENT

```
SQL> create table stdn(rollno number(3),name varchar(2),m1 number(3),m2 number(3),m3
number(3),tot num
ber(3),avrg number(3),result varchar(10));
Table created.
SQL> create or replace trigger t1 before insert on stdn
2 for each row
3 begin
4 :new.tot:=:new.m1+:new.m2+:new.m3;
5 :new.avrg:=:new.tot/3;
6 if(:new.m1>=50 and :new.m2>=50 and :new.m3>=50) then
7 :new.result:='pass';
8 else
9 :new.result:='Fail';
10 end if;
11 end;
```

```
12 /
Trigger created.
SQL> insert into stdn values(101,'SM',67,89,99,",",");
1 row created.
SQL> select * from stdn;
ROLLNO NA M1 M2 M3 TOT AVRG RESULT
--------- -- --------- --------- --------- --------- --------- ----------
101 SM 67 89 9 9 255 85 pass
```

PROCEDURE TO INSERT NUMBER

```
SQL> create table emp1(id number(3),First_name varchar2(20));
Table created.
SQL> insert into emp1 values(101,'Nithya');
1 row created.
SQL> insert into emp1 values(102,'Maya');
1 row created.
SQL> select * from emp1;
ID FIRST_NAME
--------- --------------------
101 Nithya
102 Maya
SQL> set serveroutput on;
SQL> create or replace
2 procedure insert_num(p_num number)is
3 begin
4 insert into emp1(id,First_name) values(p_num,user);
5 end insert_num;
6 /
Procedure created.
SQL> exec insert_num(3);
PL/SQL procedure successfully completed.
SQL> select * from emp1;
ID FIRST_NAME
--------- --------------------
101 Nithya
102 Maya
103 SCOTT
```

FUNCTION TO FIND FACTORIAL

```
SQL> create or replace function fact(n number)
2 return number is
3 i number(10);
4 f number:=1;
5 begin
```

```
6 for i in 1..N loop
7 f:=f*i;
8 end loop;
9 return f;
10 end;
11 /
Function created.
SQL> select fact(2) from dual;
FACT(2)
---------
2
```

Program 5. Create the table for

(a) COMPANY database

(b) STUDENT database and Insert five records for each attributes.

Creation of DATABASE for COMPANY.

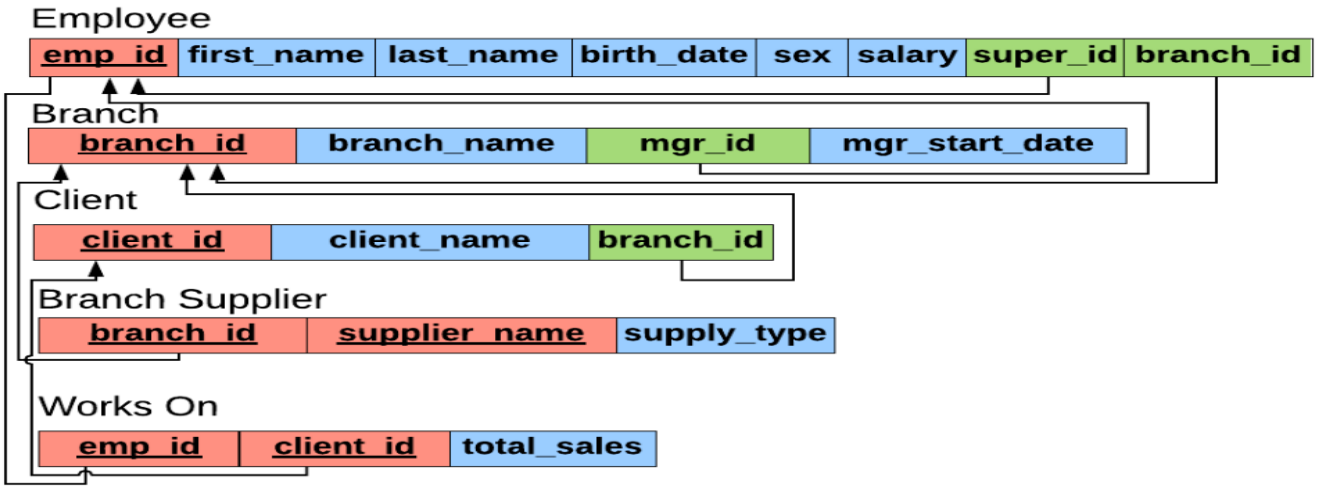
Create database COMPANY;

Creation of tables:

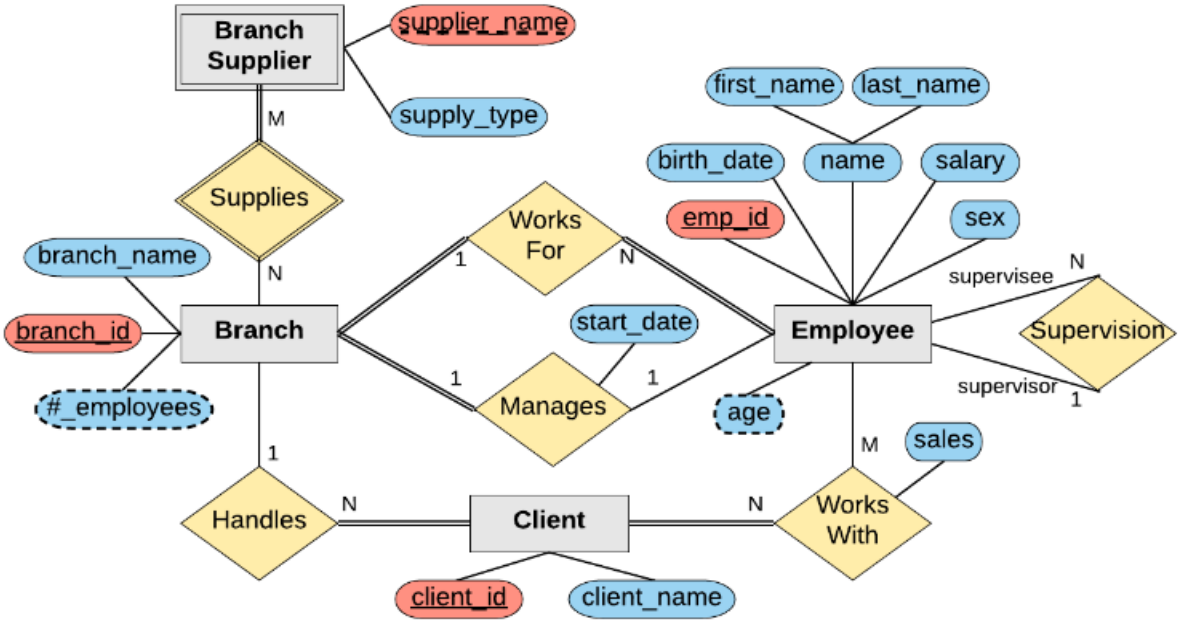
```
CREATE TABLE employee (
  emp_id INT PRIMARY KEY,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  birth_day DATE,
  sex VARCHAR(1),
  salary INT,
  super_id INT,
  branch_id INT);

CREATE TABLE branch (
  branch_id INT PRIMARY KEY,
  branch_name VARCHAR(40),
  mgr_id INT,
  mgr_start_date DATE,
  FOREIGN KEY(mgr_id) REFERENCES employee(emp_id) ON DELETE SET NULL);

ALTER TABLE employee
ADD FOREIGN KEY(branch_id)
REFERENCES branch(branch_id)
ON DELETE SET NULL;

ALTER TABLE employee
ADD FOREIGN KEY(super_id)
REFERENCES employee(emp_id)
ON DELETE SET NULL;


CREATE TABLE client (
  client_id INT PRIMARY KEY,
  client_name VARCHAR(40),
  branch_id INT,
  FOREIGN KEY(branch_id) REFERENCES branch(branch_id) ON DELETE SET NULL);

CREATE TABLE works_with (
  emp_id INT,
  client_id INT,
  total_sales INT,
  PRIMARY KEY(emp_id, client_id),
  FOREIGN KEY(emp_id) REFERENCES employee(emp_id) ON DELETE CASCADE,
  FOREIGN KEY(client_id) REFERENCES client(client_id) ON DELETE CASCADE);
```

```
CREATE TABLE branch_supplier (
  branch_id INT,
  supplier_name VARCHAR(40),
  supply_type VARCHAR(40),
  PRIMARY KEY(branch_id, supplier_name),
  FOREIGN KEY(branch_id) REFERENCES branch(branch_id) ON DELETE CASCADE);
```

Company Database Schema



Company ER Diagram



Insertion of rows to the tables employee, branch,branh_supplier, clients and works_with respectively:

```
INSERT INTO employee VALUES(100, 'K L ', 'Rahul', '1-JAN-17', 'M', 250000, NULL,
NULL);
INSERT INTO employee VALUES(101, 'Virat', 'Kohli', '9-FEB-7', 'M', 450000, NULL,
NULL);
INSERT INTO employee VALUES(102, 'Rahul', 'Dravid', '19-MAR-20', 'M', 650000,
NULL, NULL);
INSERT INTO employee VALUES(103, 'Sachin', 'Tendulkar', '10-MAY-21', 'M', 850000,
NULL, NULL);
INSERT INTO employee VALUES(104, 'M S', 'Dhoni', '1-SEP-05', 'M', 750000, NULL,
NULL);

INSERT INTO branch VALUES(3, 'Stamford', 106, '1-APR-17');
INSERT INTO branch VALUES(3, 'Stamford', 106, '1-MAY-19');
INSERT INTO branch VALUES(3, 'Stamford', 106, '1-FEB-20');
INSERT INTO branch VALUES(3, 'Stamford', 106, '1-JAN-21');
INSERT INTO branch VALUES(3, 'Stamford', 106, '1-DEC-20');

INSERT INTO branch_supplier VALUES(2, 'Hammer Mill', 'Paper');
INSERT INTO branch_supplier VALUES(2, 'Uni-ball', 'Writing Utensils');
INSERT INTO branch_supplier VALUES(3, 'Patriot Paper', 'Paper');
INSERT INTO branch_supplier VALUES(2, 'J.T. Forms & Labels', 'Custom Forms');
INSERT INTO branch_supplier VALUES(3, 'Uni-ball', 'Writing Utensils');
INSERT INTO branch_supplier VALUES(3, 'Hammer Mill', 'Paper');

INSERT INTO client VALUES(400, 'Dunmore Highschool', 2);
INSERT INTO client VALUES(401, 'Lackawana Country', 2);
INSERT INTO client VALUES(402, 'FedEx', 3);
INSERT INTO client VALUES(403, 'John Daly Law, LLC', 3);
INSERT INTO client VALUES(404, 'Scranton Whitepages', 2);

INSERT INTO works_with VALUES(105, 400, 55000);
INSERT INTO works_with VALUES(102, 401, 267000);
INSERT INTO works_with VALUES(108, 402, 22500);
INSERT INTO works_with VALUES(107, 403, 5000);
INSERT INTO works_with VALUES(108, 403, 12000);
```

Creation of DATABASE for STUDENT.

Create database STUDENT;

Creation of tables:

```
CREATE TABLE student( sid  int  not null,name text not null, primary key(sid));

CREATE TABLE teachers(tid  int  not null,name text not null,primary key(tid));

CREATE TABLE subjects(subid int  not null,name  text not null,primary key(subid));
```

```
CREATE TABLE grades(studentID int not null references students(sid),teacherID int not null references teachers(tid),subjectID int not null references subjects(subid),grade varchar(3),primary key(studentID, teacherID, subjectID));
```

Insertion of rows to the tables student,teachers,subjects and grades respectively:

```
INSERT  INTO student (sid, name) VALUES(1, 'Simon');
INSERT  INTO student (sid, name) VALUES(2, 'Alvin');
INSERT  INTO student (sid, name) VALUES(3, 'Theo');
INSERT  INTO student (sid, name) VALUES(4, 'Brittany');
INSERT  INTO student (sid, name) VALUES(5, 'Jenette');
INSERT  INTO student (sid, name) VALUES(6, 'Elenor');
INSERT  INTO student (sid, name) VALUES(7, 'Stu');

INSERT  INTO teachers (tid, name) VALUES (1, 'Washington');
INSERT  INTO teachers (tid, name) VALUES (2, 'Adams');
INSERT  INTO teachers (tid, name) VALUES (3, 'Jefferson');
INSERT INTO teachers (tid, name) VALUES (4, 'Lincoln');

INSERT  INTO subjects (subid, name) VALUES (1, 'History');
INSERT  INTO subjects (subid, name) VALUES (2, 'Biology');
INSERT  INTO subjects (subid, name) VALUES (3, 'SF');

INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (1, 2, 1, 'A');
INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (1, 2, 2, 'B');
INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (7, 4, 3, 'C+');
INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (7, 3, 2, 'F');
INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (6, 2, 1, 'B+');
INSERT  INTO grades (studentID, teacherID, subjectID, grade) VALUES (2, 4, 3, 'C');
```

PROGRAM 6: Illustrate the use of SELECT statement

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.

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;

SQL DISTINCT Clause

SELECT DISTINCT column1, column2....columnN

FROM table_name;

SQL WHERE Clause

SELECT column1, column2....columnN

FROM table_name WHERE CONDITION;

SQL AND/OR Clause

SELECT column1, column2....columnN

FROM table_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

SQL IN Clause

SELECT column1, column2....columnN

FROM table_name

WHERE column_name IN (val-1, val-2,...val-N);

SQL BETWEEN Clause

SELECT column1, column2....columnN

FROM table_name

WHERE column_name BETWEEN val-1 AND val-2;

SQL LIKE Clause

```
SELECT column1, column2....columnN
 FROM table_name
 WHERE column_name LIKE { PATTERN };
```

SQL ORDER BY Clause

```
SELECT column1, column2....columnN
 FROM table_name
WHERE CONDITION ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY Clause

```
 SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT Clause

```
 SELECT COUNT(column_name)
 FROM table_name
WHERE CONDITION;
```

Queries for company DATABASE

I)Find all employees

```
SELECT *
FROM employee;
```

2)Find all clients

```
SELECT *
FROM clients;
```

3)Find all employees ordered by salary

SELECT *

from employee

ORDER BY salary ASC/DESC;

4)Find all employees ordered by sex then name

SELECT *

from employee

ORDER BY sex, name;

5)Find the first 5 employees in the table

SELECT *

from employee

LIMIT 5;

6)Find the first and last names of all employees

SELECT first_name, employee.last_name

FROM employee;

7)Find the forename and surnames names of all employees

SELECT first_name AS forename, employee.last_name AS surname

FROM employee;

8)Find out all the different genders

SELECT DISCINCT sex

FROM employee;

9)Find all male employees

SELECT *

FROM employee

WHERE sex = 'M';

10)Find all employees at branch 2

```
SELECT *
FROM employee
WHERE branch_id = 2;
```

11)Find all employee's id's and names who were born after 1969

```
SELECT emp_id, first_name, last_name
FROM employee
WHERE birth_day >= 1970-01-01;
```

12)Find all female employees at branch 2

```
SELECT *
FROM employee
WHERE branch_id = 2 AND sex = 'F';
```

13)Find all employees who are female & born after 1969 or who make over 80000

```
SELECT *
FROM employee
WHERE (birth_day >= '1970-01-01' AND sex = 'F') OR salary > 80000;
```

14) Find all employees born between 1970 and 1975

```
SELECT *
FROM employee
WHERE birth_day BETWEEN '1970-01-01' AND '1975-01-01';
```

15) Find all employees named Jim, Michael, Johnny or David

```
SELECT *
FROM employee
WHERE first_name IN ('Jim', 'Michael', 'Johnny', 'David');
```

16)Find the number of employees

```
SELECT COUNT(super_id)
```

```
FROM employee;
```

17)Find the average of all employee's salaries

```
SELECT AVG(salary)
FROM employee;
```

18)Find the sum of all employee's salaries

```
SELECT SUM(salary)
FROM employee;
```

19)Find out how many males and females there are

```
SELECT COUNT(sex), sex
FROM employee
GROUP BY sex
```

20)Find the total sales of each salesman

```
SELECT SUM(total_sales), emp_id
FROM works_with
GROUP BY client_id;
```

21)Find the total amount of money spent by each client

```
SELECT SUM(total_sales), client_id
FROM works_with
GROUP BY client_id;
```

22)Find a list of employee and branch names

```
SELECT employee.first_name AS Employee_Branch_Names
FROM employee
UNION
SELECT branch.branch_name
FROM branch;
```

23)Find a list of all clients & branch suppliers' names

```
SELECT client.client_name AS Non-Employee_Entities, client.branch_id AS Branch_ID
FROM client
UNION
SELECT branch_supplier.supplier_name, branch_supplier.branch_id
FROM branch_supplier;
```

Queries for student DATABASE

1)Find all Students

```
select *
from Students;
```

2) **Find all teachers**

```
select *
from teachers;
```

3) **Find all subjects**

```
select *
from subjects;
```

4) **Find all grades**

```
select *
from grades;
```

5) Students in order by name:

```
select *
from students
order by name ASC;
```

6)Names of students in any class taught by Adams:

```
select name
```

```
from students
where sid in
  (select studentID
   from grades
   where teacherID in
     (select tid
      from teachers
      where name = 'Adams')
   );
```

7)Names of teachers who taught Biology:

```
select name
from teachers
where tid in
  (select teacherID
   from grades
   where subjectID in
     (select subid
      from subjects
      where name = 'Biology')
   );
```

8)Namaes of teachers who have not yet taught:

```
select name
from teachers
where tid not in
  (select teacherID
```

```
   from grades);
```

9)Names of students who have not yet taken any classes:

```
select name
from students
where sid not in
  (select studentID
   from grades);
```

10)Names of students in the same class:

```
select name
from students
where sid in
  (SELECT studentID
   FROM grades g1
   WHERE
     (SELECT COUNT(*)
      FROM grades g2
      WHERE g1.subjectID = g2.subjectID
        AND g1.teacherID = g2.teacherID ) > 1
   ORDER BY subjectID
  );


select t.name   as "Teacher",
       sub.name as "Subject",
       s.name   as "Student"
from grades g1,
```

```
    grades g2,
    students s,
    teachers t,
    subjects sub
where g1.teacherID = g2.teacherID
  and g1.subjectID = g2.subjectID
  and g1.studentID = s.sid
  and g1.teacherID = t.tid
  and g1.subjectID = sub.subid
order by t.name, sub.name, s.name;
```

Program 7: Conditional retrieval - WHERE clause

Where clause is used to fetch a particular row or set of rows from a table. This clause filters records based on given conditions and only those row(s) comes out as result that satisfies the condition defined in WHERE clause of the SQL query.

```
SELECT Column_name1, Column_name2, ....
FROM Table_name
WHERE Condition;
```

Types of conditions:

Condition	SQL Operators
Comparison	=, >, >=, <, <=, <>
Range filtering	BETWEEN
Match a character pattern	LIKE
List filtering [Match any of a list of values]	IN
Null testing	IS NULL

Comparison Operators:

SQL Operators	Meaning
=	Equal to
>, <	Greater than, less than
>= , <=	Greater than or equal to, Less than or equal to
<>	Not equal to

Let an employee table has the following columns:
(employee_id,first_name,last_name,email,phone_number,hire_date,job_id, salary,commission_pct,manager_id,department_id)

The following query display the employee_id, first_name, last_name, department_id of employees whose departmet_id=100 :

```
SELECT employee_id, first_name,
last_name, department_id
FROM employees
WHERE department_id=100;
```

The following query displays the employee_id, job_id, salary of employees whose last_name='Lorentz'.

```
SELECT employee_id, job_id, salary
FROM employees
WHERE last_name = 'Lorentz';
```

Example: WHERE clause using comparison conditions in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose salary is greater than or equal to 4000 :

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary>=4000;
```

Example: WHERE clause using expression in SQL

The following query displays the first_name, last_name , salary and (salary+(salary*commission_pct)) as Net Salary of employees whose Net Salary is in the range 10000 and 15000 and who gets atleast a percentage of commission_pct.

```
SELECT first_name,last_name,salary,
(salary+(salary*commission_pct)) AS "Net Salary"
FROM employees
WHERE
(salary+(salary*commission_pct))
BETWEEN 10000 AND 15000
AND commission_pct>0
```

Example: WHERE clause using BETWEEN condition in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose salary is greater than or equal to 4000 and less than equal to 6000 where 4000 is thelower limit and 6000 is the upper limit of the salary.

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary BETWEEN 4000 AND 6000;
```

Example: WHERE clause using IN condition in SQL

The following query displays the employee_id, first_name, last_name, department_id and salary of employees whose department_id 60, 90 or 100.

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE department_id IN(60,90,100);
```

Example: WHERE clause using LIKE condition in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose first_name starting with 'S'.

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE first_name LIKE('S%');
```

Example : WHERE clause using NULL condition in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose department_id is null.

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE department_id IS NULL;
```

Example : WHERE clause using the AND operator in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose first_name starting with 'S' and salary greater than or equal to 4000.

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE first_name LIKE('S%')
AND salary>=4000;
```

Example: WHERE clause using the OR operator in SQL

The following query displays the employee_id, first_name, last_name and salary of employees whose first_name starting with 'S' or 'A'.

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE first_name LIKE('S%')
OR first_name LIKE('A%')
```

Example: WHERE clause using the NOT operator in SQL

The following query displays the employee_id, first_name, last_name and salary of employees except the department_id 90, 60 or 100 :

```
SELECT employee_id, first_name, last_name,
department_id, salary
FROM employees
WHERE department_id
NOT IN (90, 60, 100);
```

Program 8: Query sorted - ORDER BY clause

The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order. Oracle sorts query results in ascending order by default.
Syntax for using SQL ORDER BY clause to sort data is:

```
SELECT column-list
FROM table_name [WHERE condition]
[ORDER BY column1 [, column2, .. columnN] [DESC]];
```

database table "employee";

id	name	dept	age	salary	location
100	Ramesh	Electrical	24	25000	Bangalore
101	Hrithik	Electronics	28	35000	Bangalore
102	Harsha	Aeronautics	28	35000	Mysore
103	Soumya	Electronics	22	20000	Bangalore
104	Priya	InfoTech	25	30000	Mangalore

For Example: If you want to sort the employee table by salary of the employee, the sql query would be.

```
SELECT name, salary FROM employee ORDER BY salary;
```

The output would be like

name	salary
Soumya	20000
Ramesh	25000
Priya	30000
Hrithik	35000
Harsha	35000

The query first sorts the result according to name and then displays it.
You can also use more than one column in the ORDER BY clause.
If you want to sort the employee table by the name and salary, the query would be like,

```
SELECT name, salary FROM employee ORDER BY name, salary;
```

The output would be like:

name	salary
Soumya	20000
Ramesh	25000

Priya	30000
Harsha	35000
Hrithik	35000

NOTE: The columns specified in ORDER BY clause should be one of the columns selected in the SELECT column list.

You can represent the columns in the ORDER BY clause by specifying the position of a column in the SELECT list, instead of writing the column name.

The above query can also be written as given below,

```
SELECT name, salary FROM employee ORDER BY 1, 2;
```

By default, the ORDER BY Clause sorts data in ascending order. If you want to sort the data in descending order, you must explicitly specify it as shown below.

```
SELECT name, salary
FROM employee
ORDER BY name, salary DESC;
```

The above query sorts only the column 'salary' in descending order and the column 'name' by ascending order.

If you want to select both name and salary in descending order, the query would be as given below.

```
SELECT name, salary
FROM employee
ORDER BY name DESC, salary DESC;
```

How to use expressions in the ORDER BY Clause?

Expressions in the ORDER BY clause of a SELECT statement.

For example: If you want to display employee name, current salary, and a 20% increase in the salary for only those employees for whom the percentage increase in salary is greater than 30000 and in descending order of the increased price, the SELECT statement can be written as shown below

```
SELECT name, salary, salary*1.2 AS new_salary
FROM employee
WHERE salary*1.2 > 30000
ORDER BY new_salary DESC;
```

The output for the above query is as follows.

name	salary	new_salary
Hrithik	35000	37000
Harsha	35000	37000
Priya	30000	36000

NOTE: Aliases defined in the SELECT Statement can be used in ORDER BY Clause.

9 (a) UNION, INTERSECTION and MINUS operations on tables.

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minu

Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

```
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

Example:

The First table

ID	NAME
1	Jack
2	Harry

3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

Example: Using the above First and Second table.
Union All query will be like:

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

The result set table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

Example:
Using the above First and Second table.
Intersect query will be:

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

```
SELECT column_name FROM table1
MINUS
SELECT column_name FROM table2;
```

Example
Using the above First and Second table.
Minus query will be:

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

9 (b) UPDATE, ALTER, DELETE, DROP operations on tables

UPDATE:

The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE *table_name*
SET *column1 = value1, column2 = value2, ...*
WHERE *condition*;

Example:

Below is a Customers sample database:

CustomerID	CustomerName	ContactName	Address	City	Postal Code	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbkö	Christina Berglun	Berguvsvägen 8	Luleå	S-958 22	Sweden

	p	d				

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	Postal Code	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbkö	Christina Berglu	Berguvsvägen 8	Luleå	S-958 22	Sweden

	p	nd				

UPDATE Multiple Records

It is the WHERE clause that determines how many records will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

Example

```
UPDATE Customers
SET ContactName='Juan'
WHERE Country='Mexico';
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	Postal Code	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Juan	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Juan	Mataderos 2312	México D.F.	05023	Mexico
4	Around the	Thomas Hardy	120 Hanove	London	WA1 1DP	UK

	Horn		r Sq.			
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-95822	Sweden

ALTER:

SQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

ALTER TABLE *table_name*
ADD *column_name datatype*;

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE *table_name*
DROP COLUMN *column_name*;

The following SQL deletes the "Email" column from the "Customers" table:

Example

ALTER TABLE Customers
DROP COLUMN Email;

ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

SQL Server / MS Access:

ALTER TABLE *table_name*
ALTER COLUMN *column_name datatype*;

My SQL / Oracle (prior version 10G):

ALTER TABLE *table_name*
MODIFY COLUMN *column_name datatype*;

Oracle 10G and later:

ALTER TABLE *table_name*
MODIFY *column_name datatype*;

SQL ALTER TABLE Example

Look at the "Persons" table:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons
ADD DateOfBirth date;

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City	DateOfBirth
1	Hansen	Ola	Timoteivn 10	Sandnes	
2	Svendson	Tove	Borgvn 23	Sandnes	
3	Pettersen	Kari	Storgt 20	Stavanger	

Change Data Type Example
Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:
ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;

DROP COLUMN Example
Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:
ALTER TABLE Persons
DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

DELETE:

SQL DELETE Statement
The SQL DELETE statement is used to delete rows from a table. Generally, DELETE statement removes one or more records form a table.
Syntax

DELETE FROM table_name WHERE some_condition;

Sample Table

EMPLOYEE

EMP_ID	EMP_NAME	CITY	SALARY	AGE
1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
4	Kristen	Washington	500000	29
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

Deleting Single Record

Delete the row from the table EMPLOYEE where EMP_NAME = 'Kristen'. This will delete only the fourth row.

Query

```
DELETE FROM EMPLOYEE
WHERE EMP_NAME = 'Kristen';
```

Output: After executing this query, the EMPLOYEE table will look like:

EMP_ID	EMP_NAME	CITY	SALARY	AGE

1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

Deleting Multiple Record

Delete the row from the EMPLOYEE table where AGE is 30. This will delete two rows(first and third row).

Query

DELETE FROM EMPLOYEE WHERE AGE= 30;

Output: After executing this query, the EMPLOYEE table will look like:

EMP_ID	**EMP_NAME**	**CITY**	**SALARY**	**AGE**
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

Delete all of the records

Delete all the row from the EMPLOYEE table. After this, no records left to display. The EMPLOYEE table will become empty.
Syntax

DELETE * FROM table_name;
or
DELETE FROM table_name;

Query

DELETE FROM EMPLOYEE;

Output: After executing this query, the EMPLOYEE table will look like:

EMP_ID	EMP_NAME	CITY	SALARY	AGE

DROP :

SQL DROP Keyword

DROP COLUMN

The DROP COLUMN command is used to delete a column in an existing table.
The following SQL deletes the "ContactName" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN ContactName;
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:
SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

MySQL:
ALTER TABLE Persons
DROP INDEX UC_Person;

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:
SQL Server / Oracle / MS Access:

ALTER TABLE Persons
DROP CONSTRAINT PK_Person;

MySQL:
ALTER TABLE Persons
DROP PRIMARY KEY;

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:
SQL Server / Oracle / MS Access:
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;

MySQL:
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;

DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:
SQL Server / Oracle / MS Access:

ALTER TABLE Persons

DROP CONSTRAINT CHK_PersonAge;

MySQL:
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;

DROP DEFAULT

The DROP DEFAULT command is used to delete a DEFAULT constraint.
To drop a DEFAULT constraint, use the following SQL:
SQL Server / Oracle / MS Access:

ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT;
MySQL:

ALTER TABLE Persons
ALTER City DROP DEFAULT;

DROP INDEX

The DROP INDEX command is used to delete an index in a table.
MS Access:
DROP INDEX *index_name* ON *table_name*;
SQL Server:
DROP INDEX *table_name*.*index_name*;
DB2/Oracle:
DROP INDEX *index_name*;
MySQL:
ALTER TABLE *table_name*
DROP INDEX *index_name*;

DROP DATABASE

The DROP DATABASE command is used is to delete an existing SQL database.
The following SQL drops a database named "testDB":

Example
DROP DATABASE testDB;

DROP TABLE

The DROP TABLE command deletes a table in the database.
The following SQL deletes the table "Shippers":

Example
DROP TABLE Shippers;

DROP VIEW

The DROP VIEW command deletes a view.
The following SQL drops the "Brazil Customers" view:
Example
DROP VIEW [Brazil Customers];

10) Query multiple tables using JOIN operation.

SQL JOIN

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

1. INNER JOIN
2. LEFT JOIN

3. RIGHT JOIN
4. FULL JOIN

Consider the two tables below:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

The simplest Join is INNER JOIN.

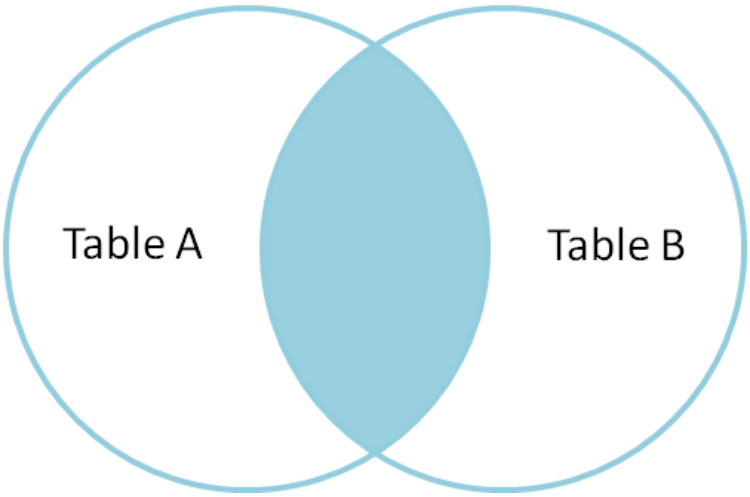
INNER JOIN: The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.

Syntax: SELECT
table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table
matching_column: Column common to both the tables.

Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



Example Queries(INNER JOIN)

This query will show the names and age of students enrolled in different courses. SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;

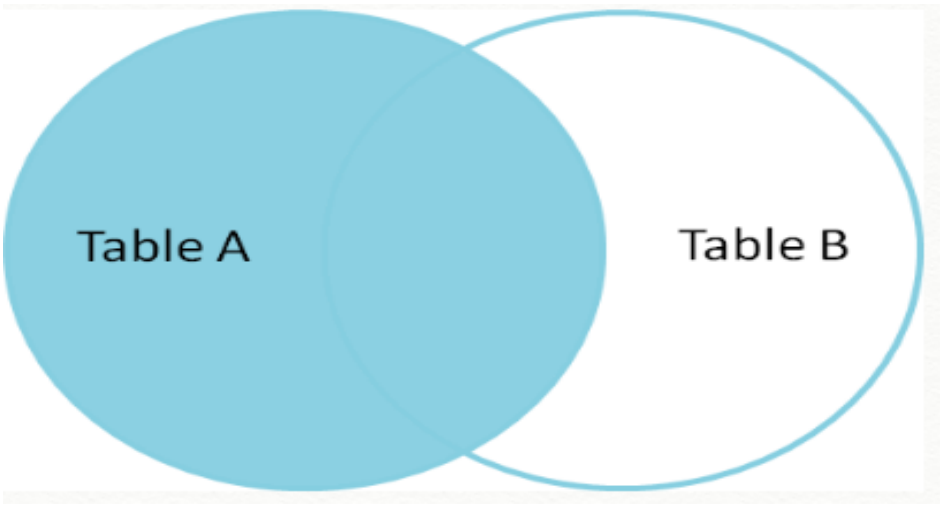
Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

LEFT JOIN: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax: SELECT
table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.



Example Queries(LEFT JOIN):SELECT
Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;

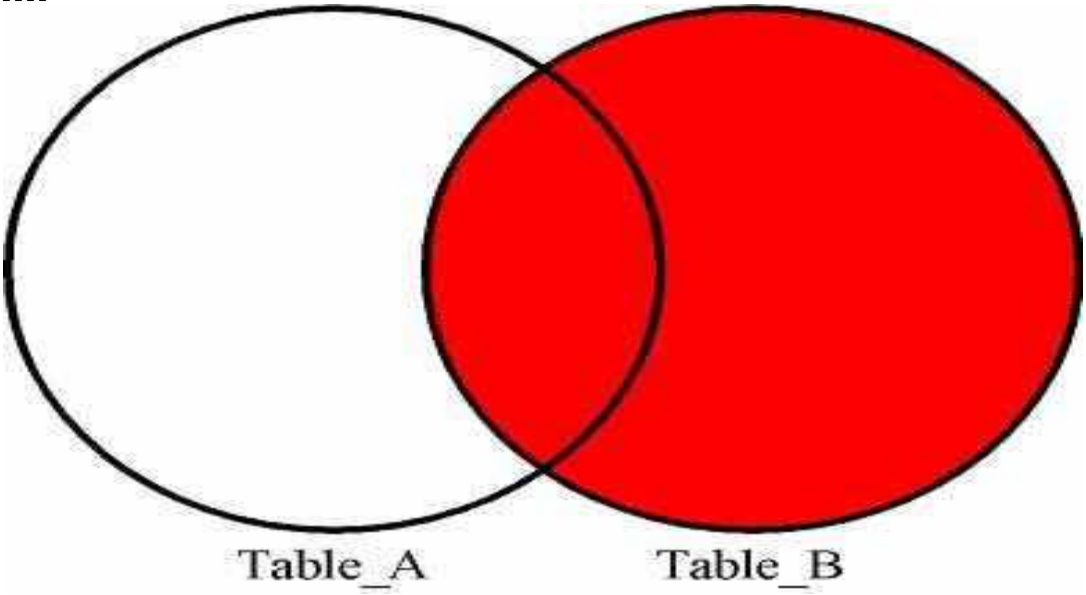
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

RIGHT JOIN: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax: SELECT
table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.



Example Queries(RIGHT JOIN):SELECT
Student.NAME,StudentCourse.COURSE_ID
FROM Student

RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

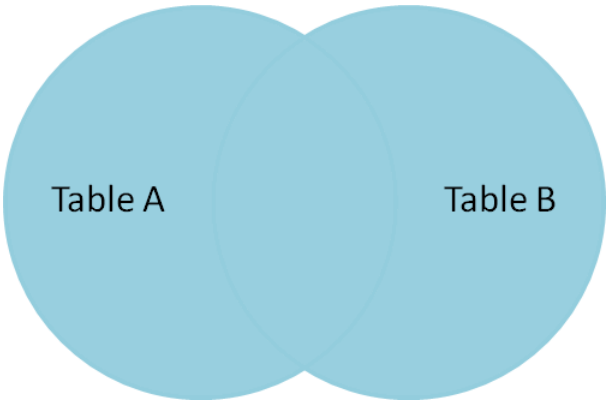
NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

FULL JOIN: FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain NULL values.

Syntax: SELECT
table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table

matching_column: Column common to both the tables.



Example Queries(FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*
NULL	9
NULL	10
NULL	11

Consider the following relations containing student class information:

Student (snum: integer, sname: string, major: string, level: string, age: integer)

Class (cname: string, meets at: time, room: string, fid: integer)

Enrolled (snum: integer, cname: string)

Faculty (fid: integer, fname: string, deptid: integer)

11. Grouping the result of query - GROUP BY clause and HAVING clause

The **GROUP BY Clause** is used to **group** rows with same values.

The **GROUP BY Clause** is used together with the SQL **SELECT** statement.

The **SELECT** statement used in the **GROUP BY clause** can only be used contain column names, aggregate functions, constants and expressions.

Syntax

SELECT statements... GROUP BY column_name1 [, column_name2,...] [HAVING condition];

Example:

Find the names of all classes that either meet in room R128 or have five or more students enrolled.

```
SQL>SELECT C.cname  FROM Class C
WHERE C.room = 'R128'  OR C.cname IN (SELECT E.cname
FROM Enrolled E  GROUP BY E.cname  HAVING COUNT (*) >= 5)
```

12. Query multiple tables using NATURAL and OUTER JOIN operation.

Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

The syntax for Natural Join is,

```
SELECT ATTRIBUTE
FROM     TABLE1 NATURAL JOIN TABLE2;
```

EXAMPLE:

SELECT C. cname FROM Class C, faculty f

faculty NATURAL JOIN class;

OUTER JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

Left Outer Join

Right Outer Join

Full Outer Join

RIGHT Outer Join

The right outer join returns a result set table with the matched data from the two tables being joined, then the remaining rows of the right table and null for the remaining left table's columns

SYNTAX

SELECT * FROM Table Name RIGHT OUTER JOIN class ON (Table1.attribute = Table2.attribute);

EXAMPLE

LEFT Outer Join

The left outer join returns a result set table with the matched data from the two tables and then the remaining rows of the left table and null from the right table's columns.

SYNTAX:

SELECT * FROM table1 Left OUTER JOIN table2 ON (Table1.attribute = Table2.attribute);

EXAMPLE:

SELECT * FROM faculty Left OUTER JOIN class ON (class.fid = faculty.fid);

Full Outer Join

The full outer join returns a result set table with the matched data of two table then remaining rows of both left table and then the right table.

SYNTAX

SELECT * FROM table1 FULL OUTER JOIN table2 ON (Table1.attribute = Table2.attribute);

EXAMPLE

SELECT * FROM faculty FULL OUTER JOIN class ON (class.fid = faculty.fid);

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

Student

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

StudentCourse

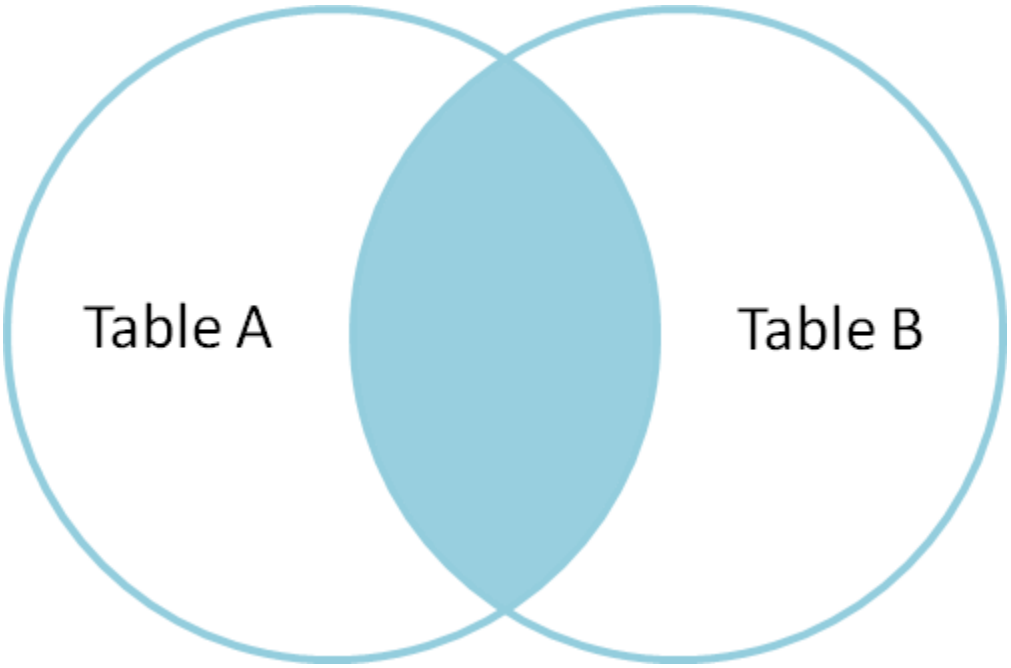
The simplest Join is INNER JOIN.

1. **INNER JOIN:** The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.
   Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
```

ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables
Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.



Example Queries(INNER JOIN)

- This query will show the names and age of students enrolled in different courses.

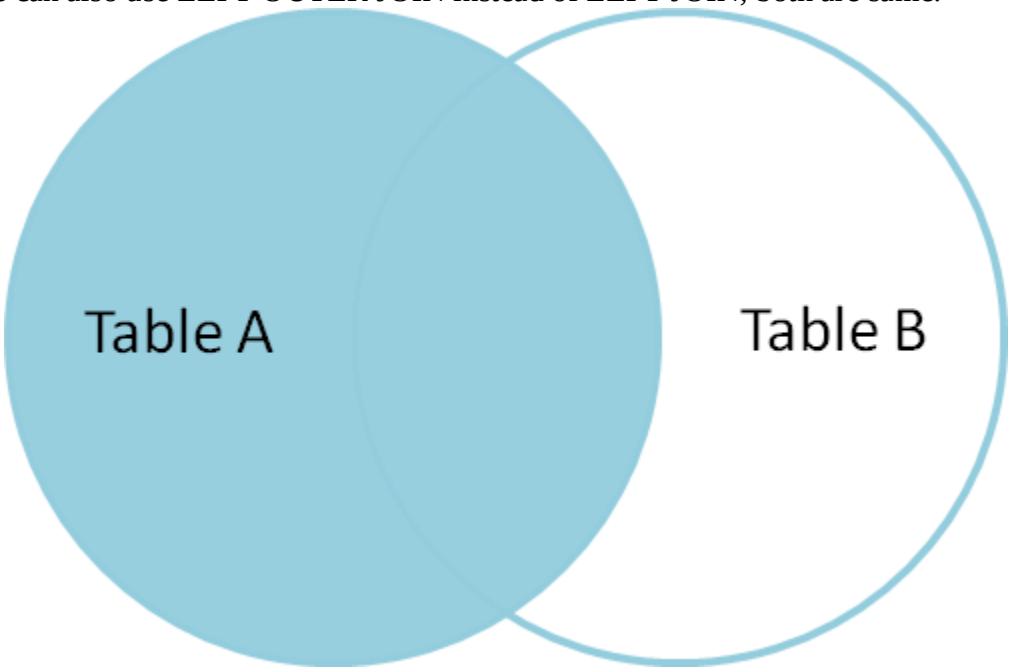
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

LEFT JOIN: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.**Syntax:**
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1

LEFT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.



Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

OUTPUT:

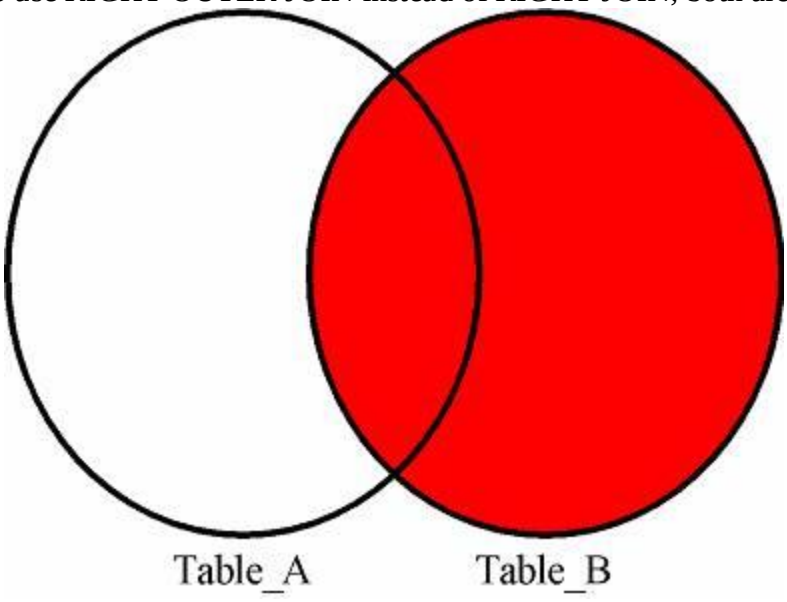
NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

RIGHT JOIN: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
table1: First table.
table2: Second table
matching_column: Column common to both the tables.

Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are same.



Example Queries(RIGHT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

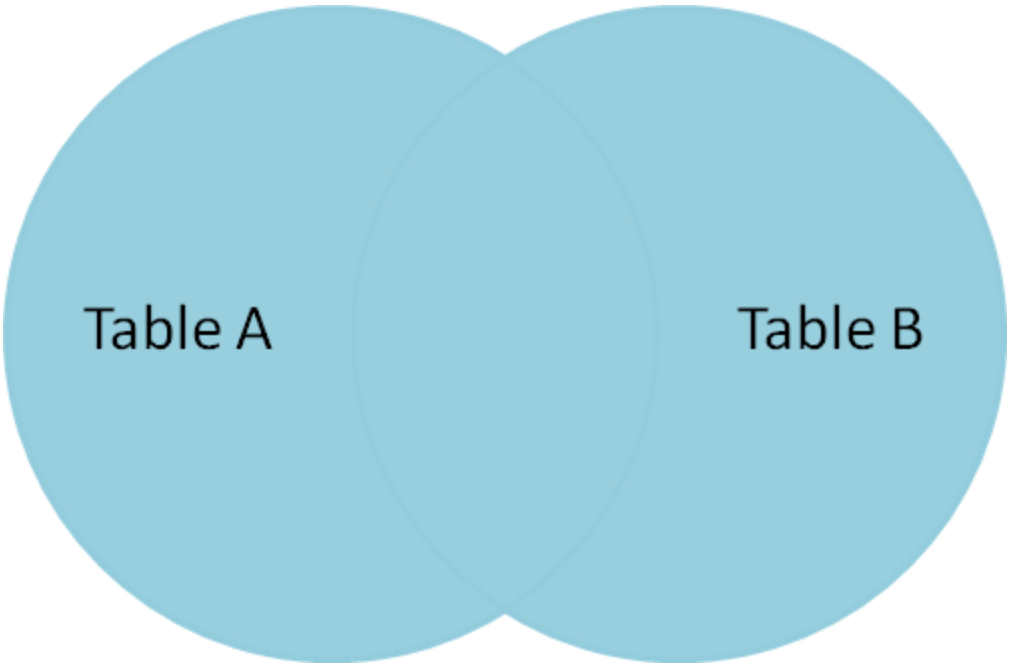
OUTPUT:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

FULL JOIN: FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.**Syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

table1: First table.
table2: Second table
matching_column: Column common to both the tables.



Example Queries(FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

OUTPUT:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*
NULL	9
NULL	10
NULL	11