LAB MANUAL

Joshua 1:9

Be strong and courageous. Do not be terrified; do not be discouraged, for the Lord your God will be with you wherever you go.

Introduction to SQL

The Structured Query Language (SQL) is a database language, pure and simple. SQL is a language with its own syntax and grammar has a number of commands for managing the database structure, powerful functions, many data types, and other useful features.

The SQL statements are used to communicate with Oracle Database.

RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables.

A table is a collection of related data entries and it consists of columns and rows.

DML (Data Manipulation Language)

It is used to retrieve, store, modify, delete, insert and update data in database.

Examples: SELECT, UPDATE, INSERT, and DELETE statements

DDL (Data Definition Language)

It is used to create and modify the structure of database objects in database.

Examples: CREATE, ALTER, DROP, TRUNCATE, and RENAME statements

DCL (Data Control Language)

It is used to create roles, permissions, and referential integrity as well it is used to control access to database by securing it.

Examples: GRANT, REVOKE statements

TCL (Transactional Control Language)

It is used to manage different transactions occurring within a database. It is used to manage the changes made by DML statements.

Examples: COMMIT, ROLLBACK statements

Week I

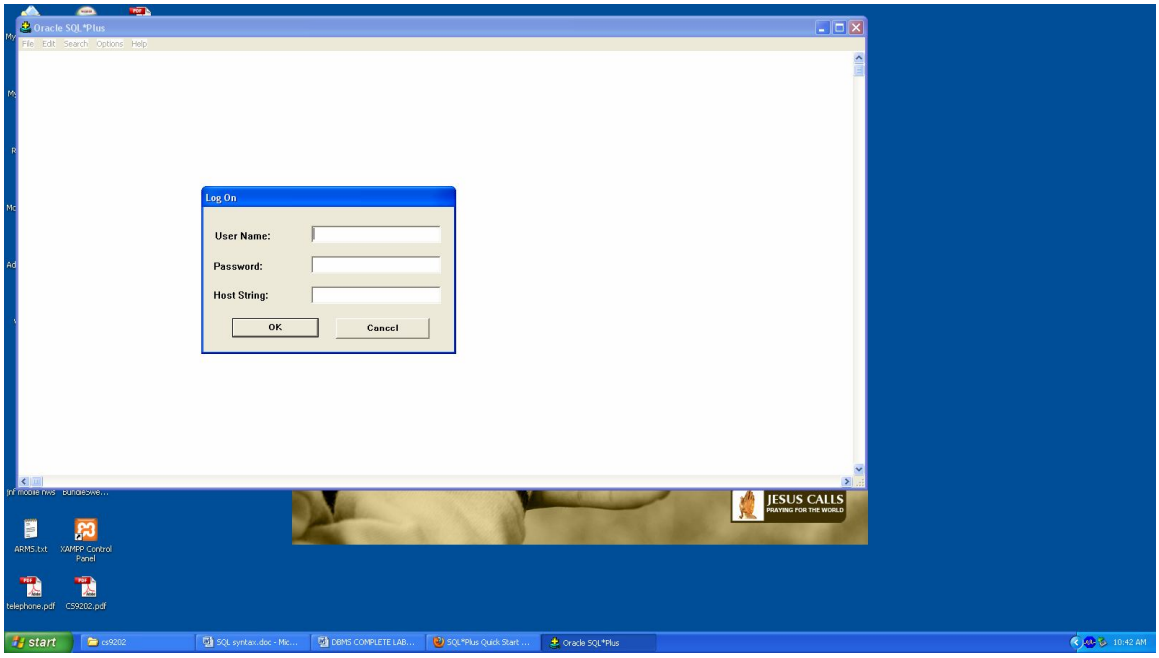
(Practice the environment with simple table creation)

Follow the instructions that will enable you to login and connect to a database.

To start with go to

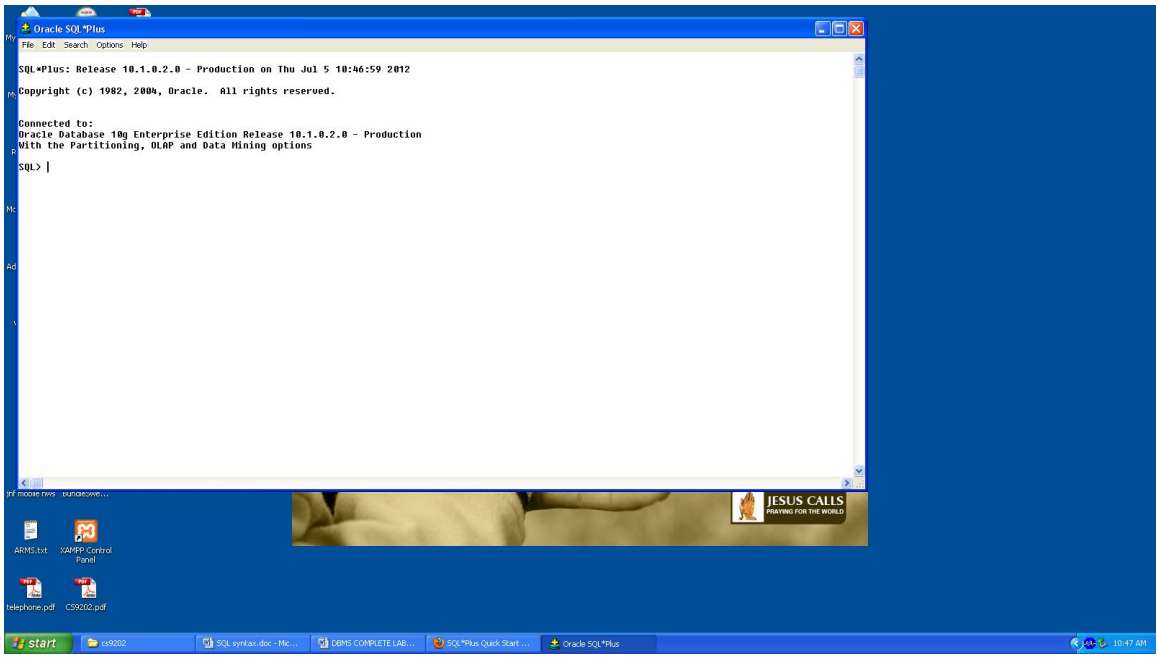
Start > All Programs > Oracle-OraDb10g-home>Application Development> SQL Plus

You can see this screen



Enter the User Name: it3j[1---70] (the order in the roll no)
Password: it
Host String: ora2012a

After successful login you can work with the SQL Prompt



Practice the following statements

SQL CREATE TABLE Syntax

```
CREATE TABLE table_name
(
column_name1 data_type [constraint],
column_name2 data_type [constraint],
column_name3 data_type [constraint],
....
)
[ ] = optional
```

Note: Today you don't want to work with constraints

SQL Data types

`char(size)` Fixed-length character string. Size is specified in parenthesis. Max 255 bytes.

`varchar(size)` Variable-length character string. Max size is specified in parenthesis.

`number(size)` Number value with a max number of column digits specified in parenthesis.

`date` Stores year, month, and day values. Oracle's default format for `DATE` is "`DD-MON-YY`"

`number(size,d) / decimal(p,d)` Number value with a maximum number of digits of "size" total, with a maximum number of "d" digits to the right of the decimal.

`int /integer /smallint` is equivalent to NUMBER(38)

`float(N)` is floating-point with at least N digits

`real, double precision` are floating-point and double-precision floating-point (machine dependent)

`time` Stores hour:minute:second

`timestamp` Has both DATE and TIME components

To VIEW the relation schema

```
DESC table_name;
```

The INSERT INTO Statement

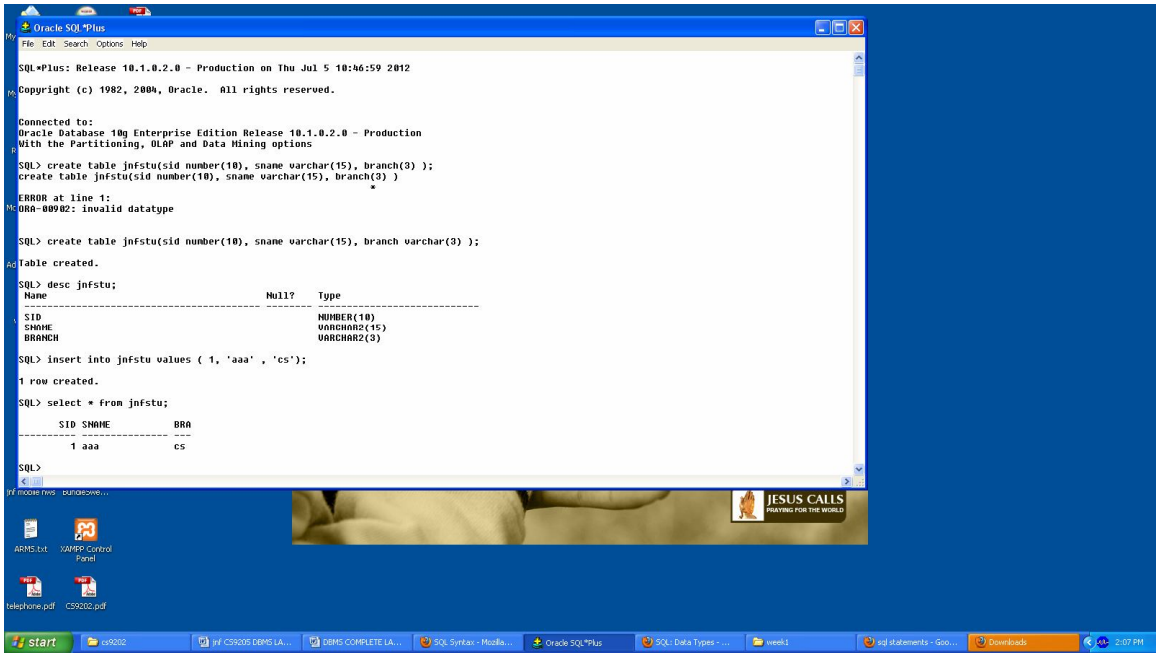
```
INSERT INTO table_name VALUES (value1, value2, value3,...)

INSERT INTO table_name (column1, column2, column3,...) VALUES
(value1, value2, value3,...)

INSERT INTO student VALUES ( &sid, &sname, &gender, &age, &branch,
&cgpa);
```

SQL> insert into student values (&sid, &sname, &gender, &age, &branch, &cgpa);
Enter value for sid: 1
Enter value for sname: 'anand'
Enter value for gender: 'm'
Enter value for age: 20
Enter value for branch: 'it'
Enter value for cgpa: 9
old 1: insert into student values (&sid, &sname, &gender, &age, &branch, &cgpa)
new 1: insert into student values (1, 'anand', 'm', 20, 'it', 9)
1 row created.

SQL> /
Enter value for sid: 2
Enter value for sname: 'akila'
Enter value for gender: 'f'
Enter value for age: 20
Enter value for branch: 'it'
Enter value for cgpa: 9.8
old 1: insert into student values (&sid, &sname, &gender, &age, &branch, &cgpa)
new 1: insert into student values (2, 'akila', 'f', 20, 'it', 9.8)
1 row created.

Simple SELECT statement

SELECT * [column_list] FROM table-name;



Creating the Tables

STUDENT, FACULTY, COURSE, CRSENRL

```
CREATE TABLE Student (sid NUMBER(10), sname VARCHAR(20), gender
CHAR(1), age INTEGER, branch CHAR(3) , cgpa REAL);

CREATE TABLE Course (courseid CHAR(6), cname VARCHAR(20), credit INT
, maxenrl SMALLINT, fid NUMBER(6) );

CREATE TABLE Faculty (fid NUMBER (6), fname VARCHAR (20), dept CHAR
(3), rank CHAR (4), salary REAL);

CREATE TABLE Crsenrl (courseid CHAR (6), sid NUMBER (10), grade CHAR
(1));
```

Modifying Relation Schemas

SQL ALTER TABLE Syntax

The SQL ALTER TABLE command is used to modify the definition (structure) of a table by modifying the definition of its columns. The ALTER command is used to perform the following functions.

1) Add, drop, modify table columns
2) Add and drop constraints
3) Enable and Disable constraints

To add a column

This can add a new column to the table.

```
ALTER TABLE table_name ADD column_name datatype;
```

```
ALTER TABLE Students ADD phoneno NUMBER(10);
```

To modify a column

This can modify the data type and size of the column.

```
ALTER TABLE table_name MODIFY column_name datatype;
```

```
SQL> desc course;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 COURSEID                                  NOT NULL CHAR(6)
 CNAME                                     VARCHAR2(20)
 CREDIT                                    NUMBER(38)
 MAXENRL                                   NUMBER(38)
 FID                                       NUMBER(6)

SQL> alter table course modify credit number(1);

Table altered.

SQL> desc course;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 COURSEID                                  NOT NULL CHAR(6)
 CNAME                                     VARCHAR2(20)
 CREDIT                                    NUMBER(1)
 MAXENRL                                   NUMBER(38)
 FID                                       NUMBER(6)

SQL> SQL> alter table course modify cname varchar2(15);

Table altered.
```

Conditions to remember while altering the structure of the table.

1. Column name and table name cannot be changed.
2. Column cannot be dropped.
3. Size cannot be decreased if data is already inserted.

Week II

(Practice table creation with constraints)

SQL Constraints

Constraints are used to limit the type of data that can go into a table.

Constraints can be specified when a table is created (with the CREATE TABLE statement) or after the table is created (with the ALTER TABLE statement).

Below are the lists of constraints:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

SQL Primary key:

This constraint defines a column or combination of columns which uniquely identifies each row in the table.

Syntax to define a Primary key at column level:

```
column name datatype [CONSTRAINT constraint_name] PRIMARY KEY
```

```
CREATE TABLE Student (sid NUMBER(10) CONSTRAINT Stu_PKey PRIMARY KEY,
sname VARCHAR(20) NOT NULL, gender CHAR(1),age INTEGER, branch
CHAR(3) , cgpa REAL );

CREATE TABLE Course (courseid CHAR(6) PRIMARY KEY,cname
VARCHAR(20),credit INT , maxenrl SMALLINT, fid NUMBER(6) );

CREATE TABLE Faculty (fid NUMBER (6) PRIMARY KEY, fname VARCHAR (20),
dept CHAR (3), rank CHAR (4), salary REAL);
```

Syntax to define a Primary key at table level:

```
[CONSTRAINT constraint_name] PRIMARY KEY (column_name1,column_name2,..)
```

```
CREATE TABLE Student (sid NUMBER(10), sname VARCHAR(20), gender
CHAR(1), age INTEGER, branch CHAR(3) , cgpa REAL, CONSTRAINT Stu_PKey
PRIMARY KEY (sid) );

CREATE TABLE Crsenrl (courseid CHAR (6),sid NUMBER(10), grade CHAR
(1), PRIMARY KEY( courseid, sid));
```

To alter already created table, use ALTER statement

```
ALTER TABLE Student ADD PRIMARY KEY (sid)

ALTER TABLE Student ADD CONSTRAINT Stu_PKey PRIMARY KEY (sid)

ALTER TABLE Student DROP CONSTRAINT Stu_PKey
```

SQL Foreign key or Referential Integrity:

This constraint identifies any column referencing the PRIMARY KEY in another table. It establishes a relationship between two columns in the same table or between different tables. For a column to be defined as a Foreign Key, it should be a defined as a Primary Key in the table which it is referring. One or more columns can be defined as Foreign key.

Syntax to define a Foreign key at column level:

```
[CONSTRAINT constraint_name] REFERENCES
Referenced_Table_name(column_name)

CREATE TABLE Course
(courseid CHAR(6) PRIMARY KEY,
cname VARCHAR(20),
credit INT ,
maxenrl SMALLINT,
fid NUMBER(6) REFERENCES Faculty (fid) );

CREATE TABLE Crsenrl
(courseid CHAR (6) CONSTRAINT Cou_FKey REFERENCES Course (courseid),
sid NUMBER (10) CONSTRAINT Stu_FKey REFERENCES Student (sid),
grade CHAR (1),
PRIMARY KEY( courseid, sid));
```

Syntax to define a Foreign key at table level:

```
[CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES
referenced_table_name(column_name);



CREATE TABLE Course (courseid CHAR(6) PRIMARY KEY,
cname VARCHAR(20),
credit INT ,
maxenrl SMALLINT,
fid NUMBER(6),
FOREIGN KEY (fid) REFERENCES Faculty (fid),
 );


CREATE TABLE Crsenrl (courseid CHAR (6),
sid NUMBER (10),
grade CHAR (1),
```

```
PRIMARY KEY( courseid, sid),
CONSTRAINT Cou_FKey  FOREIGN KEY (courseid) REFERENCES Course
(courseid),
CONSTRAINT Stu_FKey FOREIGN KEY (sid) REFERENCES Student (sid)
);
```

To alter already created table, use ALTER statement

```
ALTER TABLE Course ADD FOREIGN KEY (fid) REFERENCES Faculty (fid)

ALTER TABLE Course ADD CONSTRAINT Fac_FKey FOREIGN KEY (fid)
REFERENCES Faculty (fid)

ALTER TABLE Course DROP CONSTRAINT Fac_FKey
```

Referential Integrity Options:

We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)

Oracle allows the following possibilities

- NO ACTION

  ON DELETE NO ACTION left blank (deletion/update rejected)

- SET TO NULL: set child tuples to NULL

  ON DELETE SET NULL

- CASCADE: propagate deletion/update

  DELETE: delete child tuples

  ON DELETE CASCADE

  UPDATE: set child tuples to updated values

  ON UPDATE CASCADE

```
CREATE TABLE Crsenrl (courseid CHAR (6), sid NUMBER (10),
grade CHAR (1),
PRIMARY KEY( courseid, sid),
CONSTRAINT Cou_FKey  FOREIGN KEY (courseid) REFERENCES Course
(courseid) ON DELETE CASCADE ,
CONSTRAINT Stu_FKey FOREIGN KEY (sid) REFERENCES Student (sid) ON
DELETE CASCADE
);
```

ON DELETE CASCADE to delete all child rows when a parent row is deleted.

ON DELETE SET NULL action allows data that references the parent key to be deleted, but not updated. When referenced data in the parent key is deleted, all rows in the child table that depend on those parent key values have their foreign keys set to null.

ON DELETE NO ACTION (which is the default) prevents deleting a parent when there are children (would be nice arrangement for real life)

SQL Not Null Constraint:

This constraint ensures all rows in the table contain a definite value for the column which is specified as not null. Which means a null value is not allowed.

Syntax to define a Not Null constraint:

```
[CONSTRAINT constraint name] NOT NULL
```

```
CREATE TABLE Student (sid NUMBER(10) PRIMARY KEY, sname VARCHAR(20)
NOT NULL, gender CHAR(1) NOT NULL ,age INTEGER, branch CHAR(3) ,
cgpa REAL );
```

SQL Unique Key:

This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.

Syntax to define a Unique key at column level:

```
[CONSTRAINT constraint_name] UNIQUE
```

```
CREATE TABLE Course (courseid CHAR(6), cname VARCHAR(20) UNIQUE ,
credit INT , maxenrl SMALLINT, fid NUMBER(6) );
```

Syntax to define a Unique key at table level:

```
[CONSTRAINT constraint_name] UNIQUE(column_name)
```

```
CREATE TABLE Course (courseid CHAR(6), cname VARCHAR(20), credit INT
, maxenrl SMALLINT, fid NUMBER(6), UNIQUE (cname) );
```

SQL Check Constraint:

This constraint defines a business rule on a column. All the rows must satisfy this rule. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint at column level:

```
[CONSTRAINT constraint_name] CHECK (condition)
```

```
CREATE TABLE Student (sid NUMBER(10), sname VARCHAR(20), gender
CHAR(1) CHECK (gender IN ('M','F')), age INTEGER, branch CHAR(3)
CHECK (branch IN ('CSE' , 'IT', 'ECE','EEE')), cgpa REAL);
```

Syntax to define a Check Constraint at table level:

```
[CONSTRAINT constraint_name] CHECK (condition)
```

```
CREATE TABLE Student (sid NUMBER(10), sname VARCHAR(20), gender
CHAR(1), age INTEGER, branch CHAR(3), cgpa REAL,
CONSTRAINT chk_gen CHECK (gender IN ('M','F'));
```

To alter already created table, use ALTER statement

```
ALTER TABLE Student ADD CHECK (gender IN ('M','F'))
```

```
ALTER TABLE Student ADD CONSTRAINT chk_gen CHECK (gender IN
('M','F'))
```

```
ALTER TABLE student ADD CHECK(branch IN ('cse' , 'it', 'ece', 'eee')
);
```

```
ALTER TABLE Student DROP CONSTRAINT chk_gen
```

SQL Default Constraint:

The DEFAULT constraint is used to insert a default value into a column.

The default value will be added to all new records, if no other value is specified.

```
CREATE TABLE Course (courseid CHAR(6), cname VARCHAR(20) UNIQUE ,
credit INT DEFAULT 3, maxenrl SMALLINT, fid NUMBER(6) );
```

```
ALTER TABLE Course MODIFY credit DEFAULT 3
```

```
ALTER TABLE Course ALTER COLUMN credit DROP DEFAULT
```

To view the constraints we have created, use the following statement

```
SELECT CONSTRAINT_NAME,CONSTRAINT_TYPE, TABLE_NAME

FROM USER_CONSTRAINTS;
```

Week III

(Practice simple queries using SELECT statement)

SQL SELECT Statement

```
SELECT [DISTINCT]   *
                    column_list}
FROM table-name
[WHERE Clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause];

SELECT column_name(s) FROM table_name
```

SQL is not case sensitive. SELECT is the same as select.

```
SELECT DISTINCT column_name(s) FROM table_name
```

SQL uses single quotes around text values (most database systems will also accept double quotes).

Retrieve the names of all students.

```
SELECT sname FROM Student;
```

Display the contents of student table.

```
SELECT *  FROM Student;
```

Retrieve the names , branch and cgpa of all students.

```
SELECT sname, branch, cgpa  FROM Student;
```

Display the different courses enrolled by students

```
SELECT DISTINCT courseid FROM Crsenrl;
          (OR)
SELECT DISTINCT cname FROM Course;
```

Retrieve the different branches available in the college.

```
SELECT DISTINCT branch FROM Student;
```

WHERE Clause

*The **where** clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria described after the keyword **where**.*

```
SELECT column_name(s) FROM table_name WHERE column_name operator value
```

Operators Allowed in the WHERE Clause

Comparison Operators

Comparison operators are used to compare the column data with specific values in a condition.

= Equal
<> Not equal
> Greater than
< Less than
>= Greater than or equal
<= Less than or equal

Retrieve the names of Information Technology students.

```
SELECT sname FROM Student WHERE branch = 'IT';
```

Select names and cgpa's of students whose cgpa are more than 8.

```
SELECT sname, cgpa FROM Student WHERE cgpa > 8;
```

Logical Operators

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

The AND & OR Operators

The AND operator displays a record if both the first condition and the second condition is true.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

The OR operator displays a record if either the first condition or the second condition is true.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

We can also combine AND and OR (use parenthesis to form complex expressions).

NOT reverses the meaning of the logical operator

Retrieve the names, branch and cgpa of students who are either in IT branch or have cgpa more than 8.

```
SELECT sname, branch, cgpa  FROM Student
WHERE (branch = 'IT')  OR (cgpa > 8);
```

Retrieve the names and branch of all students who are not in IT or in ECE.

```
SELECT sname, branch  FROM Student
WHERE (branch <> 'IT')  AND (branch <> 'ECE');
```

Other special Comparison operators

The other comparison keywords available in SQL which are used to enhance the search capabilities of a SQL query.

BETWEEN Between an inclusive range
LIKE Search for a pattern
IN To specify multiple possible values for a column
IS NULL column value does not exist.

In some versions of SQL the <> operator may be written as!=

SQL IN Syntax

```
SELECT column_name(s) FROM table_name
WHERE column_name [NOT] IN (value1,value2,...)
```

List the names and branches of students who are either in CSE or in ECE or in EEE.

```
SELECT sname, branch  FROM Student
WHERE  branch IN ('CSE' , 'ECE' , 'EEE');
```

SQL BETWEEN Syntax

```
SELECT column_name(s) FROM table_name
WHERE column_name  [NOT] BETWEEN value1 AND value2
```

List the name and cgpa's of student whose cgpa lies between 6.5 and 8.5;

```
SELECT sname FROM Student
WHERE  cgpa BETWEEN 6.5 AND 8.5 ;
```

SQL LIKE Operator

```
SELECT column_name(s) FROM table_name
WHERE column_name [NOT]LIKE pattern
```

SQL Wildcards

SQL wildcards can substitute for one or more characters when searching for data in a database.

SQL wildcards must be used with the SQL LIKE operator.

With SQL, the following wildcards can be used:

Wildcard	**Description**
%	A substitute for zero or more characters
_	A substitute for exactly one character

Display the names of all students whose names starts with 'p%';

```
SELECT sname  FROM Student
WHERE  sname  LIKE 'p%' ;
```

SQL NULL Operator

```
SELECT column_name(s) FROM table_name
WHERE column_name IS [NOT] NULL
```

List the name of any student whose branch is unknown.

```
SELECT sname, branch  FROM Student
WHERE  branch IS NULL ;
```

The ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set by a specified column.

The ORDER BY keyword sort the records in ascending order by default.

If you want to sort the records in a descending order, you can use the DESC keyword.

SQL ORDER BY Syntax

```
SELECT column_name(s) FROM table_name ORDER BY column_name(s) ASC|DESC
```

Retrieve the list of students in alphabetical order

```
SELECT sname FROM Student
ORDER BY sname;
```

Retrieve a list of all students with cgpa above 5.0 in the ascending order of cgpa.

```
SELECT sname, cgpa FROM Student
WHERE cgpa > 5.0
ORDER BY cgpa;
```

Retrieve a list of students by branch and with branch arranges by highest cgpa first.

```
SELECT sname, branch, cgpa FROM Student
ORDER BY branch, cgpa DESC;
```

SQL Aggregate Functions

The syntax for using functions is,

```
SELECT "function type" ("column_name") FROM "table_name"
```

AVG: Average of the column.

```
SELECT AVG(column) FROM table_name
```

COUNT: Number of records.

```
SELECT COUNT(column) FROM table_name
```

MAX: Maximum of the column.

```
SELECT MAX(column) FROM table_name
```

MIN: Minimum of the column.

```
SELECT MIN(column) FROM table_name
```

SUM: Sum of the column.

```
SELECT SUM(column) FROM table_name
```

What is the total salary paid to the faculty?

```
SELECT sum(salary) FROM faculty;
```

Display the number of faculty members and the average salary paid to them for the faculty in IT department.

```
SELECT count(*), avg(salary) FROM faculty
where dept = 'IT';
```

SQL GROUP BY Syntax

```
SELECT column1, SUM (column2) FROM table_name  GROUP BY column1
```

SQL HAVING Syntax

```
SELECT column1, SUM(column2) FROM table_name
GROUP BY column1
HAVING (arithmetic function condition)
```

What is the average salary paid to each department?

```
SELECT dept, avg(salary) FROM faculty
group by dept';
```

What is the total salary paid by rank in each department?

```
SELECT dept, rank, sum(salary) FROM faculty
group by dept, rank;
```

List the departments having an average salary above Rs. 100000, order the results by average salary in the descending order of it.

```
SELECT dept, avg(salary) FROM faculty
GROUP BY dept
HAVING avg(salary) > 100000
ORDER BY avg(salary) DESC;
```

Week IV

(Practice Join queries)

SQL Joins

SQL Joins are used to relate information in different tables. A Join condition is a part of the sql query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL WHERE Clause of select, update, delete statements.

The Syntax for joining two tables is:

Earlier version of SQL uses this syntax

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

New version

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

Display the list of courses and faculty members teaching the course.

```
SELECT courseid, cname, fname FROM faculty, course
WHERE faculty.fid = course.fid;
```

Different SQL JOINs

- **JOIN / INNER JOIN/ EQUI JOIN**: Return rows when there is at least one match in both tables
- **LEFT JOIN**: Return all rows from the left table, even if there are no matches in the right table
- **RIGHT JOIN**: Return all rows from the right table, even if there are no matches in the left table
- **FULL JOIN**: Return rows when there is a match in one of the tables
- **SELF JOIN**: is used to join a table to itself, as if the table were two tables, temporarily renaming at least one table in the SQL statement.

SQL INNER JOIN Keyword

The INNER JOIN keyword return rows when there is at least one match in both tables.

SQL INNER JOIN Syntax

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

Display the list of courses and faculty members teaching the course.

```
SELECT courseid, cname, fname FROM faculty INNER JOIN course
ON faculty.fid = course.fid;
```

```
COURSE CNAME           FNAME
------ --------------- --------------------
1      a1              aaa
2      a2              bbb
3      a3              ccc
4      a4              ddd
5      a5              eee
6      a6              aaa
```

Why Use SQL OUTER JOIN?

Use the SQL OUTER JOIN whenever multiple tables must be accessed through a SQL SELECT statement and results should be returned if there is not a match between the JOINed tables.

It can be useful when there is a need to merge data from two tables and to include all rows from both tables without depending on a match. Another use is to generate a large result set for testing purposes.

We might possibly want to use left joins for queries that have null rows in the dependent (many) side of one-to-many relationships and right joins on those queries that generate null rows in the independent side.

For example if an employee doesn't *have a dependent, then their record won't show up at all because there's no matching record in the DEPENDENT table.*

So, we can use a left *join which keeps all the data on the "left" (i.e. the first table) and pulls in any matching data on the "right" (the second table)*

SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all rows from the left table (table_name1), even if there are no matches in the right table (table_name2).

SQL LEFT JOIN Syntax

```
SELECT column_name(s) FROM table_name1 LEFT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

```
SELECT courseid, cname, fname FROM faculty LEFT JOIN course ON
faculty.fid = course.fid;
```

```
SELECT courseid, cname, fname FROM faculty, course WHERE
faculty.fid (+) = course.fid;
```

```
COURSE CNAME           FNAME
------ --------------- --------------------
1      a1              aaa
2      a2              bbb
3      a3              ccc
4      a4              ddd
5      a5              eee
6      a6              aaa
                       fff
```

SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all the rows from the right table (table_name2), even if there are no matches in the left table (table_name1).

SQL RIGHT JOIN Syntax

```
SELECT column_name(s) FROM table_name1 RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

```
SELECT courseid, cname, fname FROM faculty RIGHT JOIN course
ON faculty.fid = course.fid;

SELECT courseid, cname, fname FROM faculty, course WHERE
faculty.fid =  course.fid (+);
```

```
COURSE CNAME           FNAME
------ --------------- --------------------
1      a1              aaa
2      a2              bbb
3      a3              ccc
4      a4              ddd
5      a5              eee
6      a6              aaa
7      a7
```

SQL FULL JOIN Keyword

The FULL JOIN keyword return rows when there is a match in one of the tables.

SQL FULL JOIN Syntax

```
SELECT column_name(s)FROM table_name1  FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

```
SELECT courseid, cname, fname FROM faculty FULL JOIN course
ON faculty.fid = course.fid;
```

```
COURSE CNAME           FNAME
------ --------------- --------------------
1      a1              aaa
2      a2              bbb
3      a3              ccc
4      a4              ddd
5      a5              eee
6      a6              aaa
                       fff
7      a7
```

In which course student ‘anand’ has enrolled?

```
SELECT courseid FROM student JOIN crsenrl
ON student.sid = crsenrl.sid and sname = ‘anand’;
```

Provide a class roster of students enrolled in ‘CS9202’. The report should include courseid, cname and sname.

```
SELECT course.courseid, cname, sname
FROM student, course, crsenrl
WHERE course.Courseid = 1 and
student.sid = crsenrl.sid and
course.courseid = crsenrl.courseid;
```

SQL Self Join

List the names of al Asst.Professors earning more than any professor.

```
SELECT f.fname, f.salary FROM faculty f, faculty s
WHERE f.rank = ‘AP’ AND s.rank = ‘Prof’
AND f.salary > s.salary;
```

List the names of the faculty members who work in the same department as Mr.X

```
SELECT s.fname AS faculties  FROM faculty f, faculty s
WHERE f.fname = ‘aaa’ AND f.dept = s.dept;
```

Week V

(Practice Sub queries/ Nested Queries)

Sub Queries/ Nested Queries

The queries that contain another complete SELECT statements nested within it i.e. embedded within the WHERE clause are called Nested Queries. The nested SELECT statement is called an "inner query" or an "inner SELECT." The main query is called "outer SELECT" or "outer query". Many nested queries can be written using JOIN operation. But the use of nested query is to avoid explicit coding of JOIN which is a very expensive database operation and to improve query performance.

However, in many cases, the use of nested queries is necessary and cannot be replaced by a JOIN operation.

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

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN can be used within the subquery.\

Subqueries with the SELECT Statement

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

NOTE:
1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle.
2) If a subquery is not dependent on the outer query it is called a non-correlated subquery.

Sub query Comparison test (=, <, >, <= , >=)

This makes use of six comparison operators (=, <>, <, <=, >, >=) available with the simple comparison test. The subquery specified in this test must produce a single value of the appropriate data type

Sub query Set Membership test (IN, NOT IN**)**

This test is used when we need to compare a value from the row being tested to a *set* of values produced by a subquery.

[NOT] IN

```
expression IN (subquery)
```

Existence test (EXISTS, NOT EXISTS**)**

[NOT] EXISTS (subquery)

```
SELECT col1 FROM table_name1 WHERE EXISTS (SELECT * FROM table_name2
WHERE table_name1.col2 = table_name2.col1;);
```

The EXISTS condition tests for the existence of a set of values. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is TRUE; if the subquery returns no rows, the result of EXISTS is FALSE.

Quantified comparison test (ALL, ANY, SOME **)**

ANY and SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=).
Used to compare a single test value to a column of data values produced by a subquery.

SOME is a synonym for ANY. IN is equivalent to = ANY.

ALL

```
expression operator ALL (subquery)
```

Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=).
Usedto compare a single test value to a column of data values produced by a subquery.

NOT IN is equivalent to <> ALL.

List the names of all students enrolled in ‘CS9202’.

```
SELECT sname FROM   student
WHERE  sid IN
           (SELECT sid FROM crsenrl
            WHERE crsnbr = ‘CS9202’);
```

List the names of the students enrolled in ‘Database management Systems’ and received a ‘S’ grade in the course.

```
SELECT sname FROM   student
WHERE  sid IN
      (SELECT sid FROM crsenrl
       WHERE grade = ‘S’ AND crsnbr =
            SELECT crsnbr FROM course
            WHERE cname = ‘Database management Systems’);
```

List the names of the faculty members who work in the same department as Mr.X

```
SELECT fname FROM   faculty
WHERE  dept =
      (SELECT dept FROM faculty
       WHERE fname = ‘Mr.A’);
```

Retrieve a list of student names who have registered for courses.

```
SELECT sname FROM   student
WHERE  EXISTS
      (SELECT * FROM crsenrl
       WHERE student.sid = crsenrl.sid );
```

Retrieve a list of faculty names who are currently taking courses.

```
SELECT fname FROM   faculty
WHERE  NOT EXISTS
      (SELECT * FROM faculty
       WHERE faculty.fid = course.fid);
```

Display the names of the faculty members who salary is greater than the salary of all the faculties in IT department.

```
SELECT fname FROM   faculty
WHERE  salary > ALL
      (SELECT salary FROM faculty
       WHERE dept = ‘IT’);
```

Correlated sub-query

It is a sub-query that is evaluated once for each row processed by the outer query.

Week VI

(Practice Set operators/Views /Sub queries in DDL and DML)

SQL SET OPERATORS

UNIONS CLAUSE

The SQL **UNION** clause/operator is used to return all distinct rows selected by either query.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

UNION [ALL]

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

Display the sid's of all CSE students and other branch students who are taking computer science courses.

```
SELECT sid FROM student WHERE branch = 'CSE'
UNION
SELECT DISTINCT sid FROM crsenrl WHERE crsnbr LIKE 'CS%';
```

INTERSECT Clause

The SQL **INTERSECT** clause/operator is used to return all distinct rows selected by both queries.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

INTERSECT [ALL]

SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

EXCEPT Clause

The SQL **EXCEPT** clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]

EXCEPT
```

```
SELECT column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE condition]
```

The UNION/ INTERSECT/ EXCEPT **ALL** operator is used to combine the results of two SELECT statements including duplicate rows.

Views

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

Any relation that is not part of the logical model, but is made visible to a user as a virtual relation is called a view.

Views are created for security reasons or to hide information or to collect information from more than one relation into a single relation.

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

Updating a View:

A view can be updated under certain conditions:

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

Create a view for computer science faculty with columns fid, fname and rank.

```
CREATE VIEW csfac AS SELECT fid,fname,salary FROM faculty
WHERE dept ='CSE';
```

Create a view called Class-Roster with fields crsnbr, cname, fname , sid and sname .

```
CREATE VIEW Class-Roster
AS SELECT course.crsnbr, cname, fname, student.sid, sname FROM
faculty, course, crsenrl, student
```

```
WHERE faculty.fid = course.fid AND course.crsnbr = crsenrl. Crsnbr
AND crsenrl.sid = student.sid
```

WITH CHECK OPTION:

When views are updateable, it is possible that when we insert a new record or update an existing record, the record added or modified doesn't logical belong to the view any longer.

WITH CHECK OPTION forces all data modification statements executed against the view to check for integrity between source table and views when inserts or updates are done.

```
SQL> CREATE VIEW csstu AS SELECT sid,sname,gender,age,branch,cgpa
FROM student WHERE branch ='CSE' WITH CHECK OPTION;
```

View created.

```
SQL> insert into csstu values (5, 'eee', 'f', 18, 'cse', 9 );
insert into csstu values (5, 'eee', 'f', 18, 'cse', 9 )
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Dropping a view

```
DROP VIEW view-name;
```

```
DROP VIEW csfac;
```

DDL Statements

SQL UPDATE Syntax

```
UPDATE table_name SET column1=value, column2=value2,... WHERE
some_column=some_value
```

Change grade of Student-10 from B to A in CS9205.

```
UPDATE Crsenrl SET grade = 'A' WHERE sid = 10 AND crsnbr = 'CS9205';
```

SQL DELETE Syntax

```
DELETE FROM table_name WHERE some_column=some_value
```

Delete sid-106 from the enrollment of CS9205.

```
DELETE FROM crsenrl WHERE sid = 106 and crsnbr = 'CS9205' ;
```

Delete All Rows

```
DELETE * FROM table_name    (OR)

DELETE FROM table_name
```

Subqueries in DML and DDL commands

Subqueries can also be used with DML commands. WHERE clause of UPDATE and DELETE can always contain a subquery.

Delete the students who have got a U-grade in CS9205.

```
DELETE FROM student where sid IN
(SELECT sid from crsenrl WHERE grade = 'U' AND crsnbr = 'CS9205');
```

Update the salary of faculty by 5% who are taking the course CS9205.

```
UPDATE faculty SET salary = salary + salary * 0.05 WHERE fid IN
(SELECT fid FROM course WHERE crsnbr = 'CS9205');
```

Populate one table using another table:

We can populate data into a table through select statement over another table provided another table has a set of fields which are required to populate first table. Here is the syntax:

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

Week VII

(Practice Simple PL/SQL)

What is PL/SQL?

PL/SQL stands for Procedural Language extension of SQL.

The basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks.

With PL/SQL, we can use SQL statements to manipulate data and flow-of-control statements to process the data. Moreover, we can declare constants and variables, define procedures and functions, and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

A PL/SQL block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception (or Error) Handling section (optional).

```
DECLARE
   Variable declaration
BEGIN
   Program Execution
EXCEPTION
   Exception handling
END;
```

PL/SQL Comments

Single-line comments start with a double-dash (--). Multiline comments start with a slash and asterisk (/*) and end with an asterisk and slash (*/)

PL/SQL Variables

```
variable_name datatype [NOT NULL := value ];
```

Declaring Variables

```
sid NUMBER(10);
```

Assigning Values to a Variable

There are two ways to assign values to variables.

1.Using assignment operator :=

```
bonus := salary * 0.10;
```

2. By selecting or fetching database values into a variable.

```
SELECT salary * 0.10 INTO bonus FROM faculty WHERE fid = fac_id;
```

Declaring Constants

Use the keyword `CONSTANT to declare constants.`

```
credit_limit CONSTANT REAL := 5000.00;
```

PL/SQL Operators.

Operator	Description
`+ - / *`	arithmetic
=	equality
!= or <>	inequality
`||`	string concatenation
:=	assignment

Some Oracle built-in functions.

Function	Description
String Functions	
`upper(s), lower(s)`	convert string `s` to upper/lower-case
`initcap(s)`	capitalize first letter of each word
`ltrim(s), rtrim(s)`	remove blank char. from left/right
`substr(s,start,len)`	sub-string of length `len` from position `start`
`length(s)`	length of `s`
Date Functions	
`sysdate`	current date (on Oracle server)
`to_date(date,format)`	date formatting to_date('2003/07/09', 'yyyy/mm/dd') would return a date value of July 9, 2003. to_date('070903', 'MMDDYY') would return a date value of July 9, 2003. to_date('20020315', 'yyyymmdd') would return a date value of Mar 15, 2002.

Number Functions	
`round(x)`	round real number `x` to integer
`mod(n,p)`	`n` modulus `p`
`abs(x)`	absolute value of `x`
`dbms_random.random()`	generate a random integer
Type Conversion Functions	
`to_char()`	convert to string
`to_date()`	convert to date
`to_number()`	convert to number
Miscellaneous Functions	
`user`	current Oracle user

ANONYMOUS BLOCK

Anonymous blocks are generally constructed dynamically and executed only once by the user. It is sort of a complex SQL statement.

To write a PL/SQL *anonymous block,*

```
SQL> SET SERVEROUTPUT ON
```

SET SERVEROUTPUT ON is the SQL*Plus command1 to activate the console output.

Note: *This need to be issued once in a SQL*Plus session.*

```
SQL> BEGIN
     dbms_output.put_line('Welcome to PL/SQL');
     END;
     /
```

The simplest way to run a function (e.g. `sysdate`) is to call it from within an SQL statement:

```
SQL> SELECT sysdate FROM DUAL
  2    /

SYSDATE
---------
28-AUG-12
```

Usage of the utility function dbms_output.put_line

```
DECLARE
    name          VARCHAR2(30);
    desig         VARCHAR2(20);
BEGIN
    name  := 'XXX';
```

```
      desig := ‘Prof’;
    DBMS_OUTPUT.PUT_LINE(name||' works as a '|| desig);
END;
/
```

Control Flow in PL/SQL

Conditional structures:

IF statement:

```
IF <condition>
THEN <statement_list>
ELSE <statement_list>                --optional.
END IF;
```

For a multiway branch, we can use:

```
IF <condition_1> THEN ...

ELSIF <condition_2> THEN ...

... ...

ELSIF <condition_n> THEN ...

ELSE ...

END IF;
```

```
DECLARE
 var1 INTEGER:= 10;
BEGIN
 IF var1 = 10
 THEN
 DBMS_OUTPUT.put_line ('var1 value is equal to 10');
 ELSE
 DBMS_OUTPUT.put_line ('var1 value is not equal to 10');
 END IF;
END;
/
var1 value is equal to 10

PL/SQL procedure successfully completed.
```

CASE statement

```
CASE [ expression ]
 WHEN condition_1 THEN result_1
 WHEN condition_2 THEN result_2
 WHEN condition_n THEN result_n
 ELSE result
END CASE
```

```
DECLARE
```

```
 var1 INTEGER := 5;
BEGIN
 case
 when var1 < 7 then DBMS_OUTPUT.put_line ('var1 value is less than 7');
 when var1 = 7 then DBMS_OUTPUT.put_line ('var1 value is equal to 7');
 when var1 > 7 then DBMS_OUTPUT.put_line ('var1 value is greater than
7');
 else DBMS_OUTPUT.put_line ('var1 value is unknown');
 END CASE;
END;
/
var1 value is less than 7

PL/SQL procedure successfully completed.

DECLARE
 var1 INTEGER := 5;
BEGIN
 case 5
 when 5 then DBMS_OUTPUT.put_line ('var1 value is less than 7');
 when 7 then DBMS_OUTPUT.put_line ('var1 value is equal to 7');
 when 10 then DBMS_OUTPUT.put_line ('var1 value is greater than 7');
 else DBMS_OUTPUT.put_line ('var1 value is unknown');
 END CASE;
END;
/
```

Iterative structures

Unconstrained LOOP statement

```
LOOP

    <loop_body> /* A list of statements. */

END LOOP;
```

EXIT statement

EXIT by itself is an unconditional loop break.
At least one of the statements in `<loop_body>` should be an `EXIT` statement of the form `EXIT WHEN <condition>;`

The loop breaks if <condition> is true.

```
DECLARE
 i INTEGER := 0;
BEGIN
 LOOP
 i := i + 1;
 DBMS_OUTPUT.put_line ('The index value is ' || i);
 EXIT WHEN i >= 10;
 END LOOP;
END;
/
The index value is 1
The index value is 2
```

```
The index value is 3
The index value is 4
The index value is 5
The index value is 6
The index value is 7
The index value is 8
The index value is 9
The index value is 10
```

WHILE loop

```
          WHILE <condition> LOOP

              <loop_body>

          END LOOP;
```

```
DECLARE
 i INTEGER := 1;
BEGIN
 WHILE i <= 10
 LOOP
 DBMS_OUTPUT.put_line ('The index value is ' || i);
 i := i + 1;
 END LOOP;
END;
```

FOR loop

```
          FOR <var> IN [REVERSE]<start>..<finish> LOOP

              <loop_body>

          END LOOP;
```

Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start> and <finish> are constants.

```
BEGIN
 FOR i IN REVERSE 1 .. 10
 LOOP
 DBMS_OUTPUT.put_line ('The index value is ' || i);
 END LOOP;
END;
/
The index value is 10
The index value is 9
The index value is 8
The index value is 7
The index value is 6
The index value is 5
The index value is 4
The index value is 3
The index value is 2
The index value is 1
```

```
        FOR i IN (select_statement)
```

```
        LOOP
                statement;
        END LOOP;
```

```
BEGIN
 FOR i IN (SELECT fname FROM faculty
 WHERE dept = 'ECE')
 LOOP
 DBMS_OUTPUT.put_line ('Faculty name: '|| i.fname);
 END LOOP;
END;
```

Week VIII

(Practice PL/SQL Procedures and Functions)

NAMED BLOCKS:

Named blocks are one that have a name associated with them, are stored in the database, and can be executed again and again, can take in parameters, and can modify and existing database.

Stored procedures, functions and triggers comes under the category of Named blocks.

PROCEDURES

A procedure is created by the keywords `CREATE PROCEDURE` followed by the procedure name and its parameters. An option is to follow `CREATE OR REPLACE`.

The possible modes of passing parameters to a procedure are `IN` (read-only), `OUT` (write-only), and `INOUT` (read and write).

Following the arguments is the keyword **`IS`** (**`AS`** is a synonym). Then comes the body, which is essentially a PL/SQL block. However, the `DECLARE` section should *not* start with the keyword `DECLARE`. Rather, following **`IS`** we have:

```
 CREATE OR REPLACE PROCEDURE proc_name
 IS

<local_var_declarations>

BEGIN

    <procedure_body>

END;
.
run;
```

To write a procedure,

```
CREATE OR REPLACE PROCEDURE first_proc
IS
user_name VARCHAR2(8) := user;
BEGIN
dbms_output.put_line('Welcome to PL/SQL, '
      || user_name || '!');
END;
  /

Procedure created.

SQL> EXEC first_proc;
Welcome to PL/SQL, SYSTEM!
```

```
PL/SQL procedure successfully completed.

CREATE OR REPLACE
PROCEDURE HELLO IS
BEGIN
DBMS_OUTPUT.PUT_LINE ('Hello World');
END;
 /

Procedure created.

SQL> call hello()
  2  /
Hello World

Call completed.

We can also execute a procedure with an anonymous block.

SQL> begin
  2  hello();
  3  end;
  4  /
Hello World

PL/SQL procedure successfully completed.

CREATE OR REPLACE
PROCEDURE DISP_AB (A INT, B INT) IS
BEGIN
DBMS_OUTPUT.PUT_LINE ('A + B = ' || (A + B));
DBMS_OUTPUT.PUT_LINE ('A * B =' || (A * B));
END;

SQL> call disp_ab(5, 3 );
A + B = 8
A * B =15

Call completed.

SQL> exec disp_ab(5,3);
A + B = 8
A * B =15

PL/SQL procedure successfully completed.

CREATE OR REPLACE
PROCEDURE SUM_AB (A INT, B INT, C OUT INT) IS
BEGIN
C := A + B;
END;

DECLARE
R INT;
BEGIN
SUM_AB(23,27,R);
DBMS_OUTPUT.PUT_LINE ( 'SUM IS: ' || R);
END;

/
```

```
SUM IS: 50
```

FUNCTIONS

Like procedures we can also write functions. **It may take in one or more parameters and RETURNs only one value back to the calling application.**

CREATE FUNCTION <func_name>(<param_list>) RETURN <return_type> IS ...

```
CREATE OR REPLACE FUNCTION Ask_CGPA(stuid IN NUMBER)
RETURN REAL
IS
pcgpa REAL;
BEGIN
SELECT cgpa into pcgpa FROM student WHERE sid = stuid ;
Return (pcgpa);
END;
  /
```

```
To execute this function,

SQL> SELECT Ask_CGPA(1) from DUAL;
ASK_CGPA(1)
-----------
       9.3
```

```
CREATE OR REPLACE FUNCTION CalcTax (salary IN  faculty.salary%TYPE)
RETURN NUMBER
AS
BEGIN
    RETURN (salary * 0.028);
END CalcTax;
/
```

```
SQL> SELECT fname AS "FACULTY", CalcTax(salary) AS "TAX"
  2  FROM faculty;

FACULTY                       TAX
-------------------- ----------
aaa                         2800
bbb                         5600
ccc                         8400
ddd                         2800
```

A function can be executed in the following ways.

1) Since a function returns a value we can assign it to a variable.

```
employee_name :=  employer_details_func;
```

If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

3) In a PL/SQL Statements like,

```
dbms_output.put_line(employer_details_func);
```
This line displays the value returned by the function.

To find out what procedures and functions you have created, use the following SQL query:

```
SELECT OBJECT_TYPE, OBJECT_NAME FROM USER_OBJECTS
WHERE  OBJECT_TYPE IN ('FUNCTION', 'PROCEDURE');
```

To drop a stored procedure/function:

```
drop procedure <procedure_name>;
```

```
drop function <function_name>;
```

Difference between Functions and procedures

	Function	**Procedure**
Parameters	input, output	input, output
Returns value	yes	no
Can be called within SQL	yes	no

Week IX

(Practice PL/SQL Cursors)

Embedding SQL in PL/SQL

Working with Cursors.

A cursor is used to process a single row 'at a time' from multiple rows retrieved from the database. A cursor is a temporary work area created in the system memory when a SQL statement is executed.
Cursors are declared in the Declaration Section.

CURSOR c1 IS SELECT fname, salary FROM faculty;

There are two types of cursors in PL/SQL:

Implicit cursors:

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

For example,

```
SELECT selectfields INTO declared_variables FROM table_list WHERE
search_criteria;
```

%TYPE or %ROWTYPE

Variables can be associated with a table structure.

%TYPE

This corresponds to the data type of the column in the table.

To display the name and cgpa for student 5. (using %TYPE)

```
DECLARE
 stu_name student.sname%TYPE;
 stu_cgpa student.cgpa%TYPE;

BEGIN
 SELECT sname, cgpa
 INTO stu_name, stu_cgpa
 FROM student
 WHERE sid = 5;
 DBMS_OUTPUT.put_line (‘student name is  ’ || stu_name );
 DBMS_OUTPUT.put_line (‘student cgpa is  ’ || stu_cgpa );
END;
```

%ROWTYPE

This corresponds to the data type of the row in the table.

To display the name, branch and cgpa for student 5. (using %ROWTYPE)

```
DECLARE
 stu_rec student%ROWTYPE;
BEGIN
 SELECT *
 INTO stu_rec
 FROM student
 WHERE sid = 5;
 DBMS_OUTPUT.put_line ('name: ' || stu_rec.sname);
 DBMS_OUTPUT.put_line ('branch: ' || stu_rec.branch);
 DBMS_OUTPUT.put_line ('cgpa: ' || stu_rec.cgpa);
END;
```

Explicit cursors:

Explicit cursors must be created when executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When we fetch a row the cursor position moves to next row.

Cursor Attributes

The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

SQL%FOUND - TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row.

SQL%NOTFOUND - TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row.

SQL%ROWCOUNT - Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT

SQL%ISOPEN - TRUE, if the cursor is already open in the program

To raise the salary of all faculties.

```
DECLARE
norows number(5);
BEGIN
  UPDATE faculty
  SET salary = salary + 5000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('None of the salaries where updated');
```

```
  ELSIF SQL%FOUND THEN
    norows := SQL%ROWCOUNT;
    dbms_output.put_line('Salaries for ' || norows || 'faculties are
updated');
  END IF;
END;
```

The General Syntax for creating a cursor is as given below:

```
CURSOR cursor_name IS select_statement;
```

The FETCH statement is used to retrieve the output of a single record from the CURSOR SELECT statement INTO associate variables.

FETCH c_query INTO v_last_name, v_salary;

There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

Step 1:

```
   DECLARE CURSOR c1  IS
   SELECT * FROM student  WHERE cgpa > 9;
```

Step 2:

```
OPEN cursor_name;
OPEN c1;
```

Step 3:

```
FETCH cursor_name INTO record_name;
```

OR

```
FETCH cursor_name INTO variable_list;

FETCH c1 INTO stu_rec;
```

Step 4:

```
CLOSE cursor_name;
CLOSE c1;
```

To display the name and salary of all faculties.

```
DECLARE
  v_name         VARCHAR2(15);
  v_salary         faculty.salary%TYPE;
  CURSOR c1   IS
    SELECT fname, salary FROM faculty;
BEGIN
  OPEN c1;
```

```
  LOOP
    FETCH c1 INTO v_name, v_salary;
    IF v_salary >= 40000 THEN
      DBMS_OUTPUT.PUT_LINE (v_name||' '||v_salary);
    END IF;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
END;
/
```

Cursor FOR Loop

This automatically opens the cursor, fetches the records, then closes the cursor. The general syntax is,

FOR variable_name(s) IN cursor_name LOOP
processing commands
END LOOP;

Cursor variables cannot be used outside loop.

```
DECLARE
s student%ROWTYPE;
CURSOR c1 IS
SELECT * FROM student;
BEGIN
FOR s IN c1 LOOP
DBMS_OUTPUT.PUT_LINE('Student ' || s.sname|| ' and ' || s.cgpa);
END LOOP;
END;
```

Week X

(Practice PL/SQL Exceptions)

EXCEPTION HANDLING

Handling Runtime Errors in PL/SQL Programs

- Runtime errors cause exceptions
- Exception handlers exist to deal with different error situations
- Exceptions cause program control to fall to exception section where exception is handled

It has the following structure

```
EXCEPTION
      WHEN EXCEPTION_NAME
      THEN
            ERROR-PROCESSING STATEMENTS;
```

BUILT-IN EXCEPTIONS

- When a built-in exception occurs, it is said to be raised implicitly.

```
DECLARE
        n1 integer := &s1;
        n2 integer := &s2;
  res number;
BEGIN
  res := n1 / n2;
  DBMS_OUTPUT.PUT_LINE ('result: '|| res);
EXCEPTION
  WHEN ZERO_DIVIDE
  THEN
    DBMS_OUTPUT.PUT_LINE
     ('A number cannot be divided by zero.');
END;
```

some common Oracle runtime errors are predefined in the PL/SQL as exceptions.
For e.g. **ORA-01476: divisor is equal to zero**

NO_DATA_FOUND exception is raised when a SELECT INTO statement does not return any rows.
TOO_MANY_ROWS exception is raised when a SELECT INTO statement returns more than one row
OTHERS All predefined Oracle errors (exceptions) can be handled with the help of the this handler.

```
DECLARE
  Stuname student.sname%TYPE;
BEGIN
 SELECT sname
 INTO stuname
```

```
 FROM student
 WHERE sid = 101;
   DBMS_OUTPUT.PUT_LINE
   (‘Student name is’|| stuname);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
     DBMS_OUTPUT.PUT_LINE (‘There is no such student’);
END;


DECLARE
   fac_number   Faculty.fid%TYPE:= &f;
   fac_name     Faculty.fname%TYPE;
BEGIN
   SELECT fname INTO fac_name FROM Faculty
      WHERE fid = fac_number;
   DBMS_OUTPUT.PUT_LINE('Faculty name is ' || fac_name);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No such faculty: ' || fac_number);
END;
```

Exceptions in functions

```
CREATE OR REPLACE FUNCTION FindCourse(name IN Course.cname%TYPE)
   RETURN Course.courseid%TYPE
IS
  cid Course.courseid%TYPE;
 BEGIN
 select courseid into cid from course  where cname = name;

RETURN cid;

EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE(‘No Such course exists’);
RETURN NULL;
END;
```

Handling Multiple Exceptions

The following program contains two exceptions in the single exception handling section.

```
DECLARE
  stuid crsenrl.sid%TYPE := &stid;
  enrolled VARCHAR2(3) := 'NO';
BEGIN
   DBMS_OUTPUT.PUT_LINE
   (‘Check if the student is enrolled’);
SELECT ‘YES’
INTO enrolled
FROM crsenrl
 WHERE sid = stuid;
    DBMS_OUTPUT.PUT_LINE
```

```
    ('The student is enrolled into one course');
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE('The student is not enrolled');
  WHEN TOO_MANY_ROWS
  THEN
    DBMS_OUTPUT.PUT_LINE
   ('The student is enrolled into many courses');
END;
```

User Defined Exceptions

- It is defined by the programmer.
- Before the exception can be used, it must be declared.
- A user-defined exception must be raised explicitly.

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   …
  IF CONDITION
  THEN
       RAISE exception_name;
   ELSE
        …
    END IF;
EXCEPTION
     WHEN exception_name
      THEN
           ERROR-PROCESSING
           STATEMENTS;
END;
```

Raise the salary of any faculty if salary is not NULL

```
CREATE OR REPLACE PROCEDURE raise_salary (id Faculty.fid%TYPE)
 IS
   cursal NUMBER(6);
   null_salary EXCEPTION;
BEGIN
   SELECT salary INTO cursal FROM Faculty WHERE fid = id;
   IF cursal IS NULL THEN

      raise null_salary;
   ELSE
      UPDATE Faculty SET salary = cursal + 5000 WHERE fid = id;
   END IF;
EXCEPTION
      WHEN null_salary THEN
       DBMS_OUTPUT.PUT_LINE('No salary exists');
END;
```

Pragma EXCEPTION_INIT

When we declare our own exception, we must RAISE it explicitly. All declared exceptions have an error code of 1 and the error message "User-defined exception," unless we use the EXCEPTION_INIT pragma.

We can associate an error number with a declared exception with the PRAGMA EXCEPTION_INIT statement:

```
DECLARE
   exception_name EXCEPTION;
   PRAGMA EXCEPTION_INIT (exception_name, error_number);
```

where *error_number* is a literal value (variable references are not allowed). This number can be an Oracle error, such as -1855, or an error in the user-definable -20000 to -20999 range.

Raise the salary of any faculty if salary is not NULL

```
CREATE OR REPLACE PROCEDURE raise_salary (id Faculty.fid%TYPE)
 IS
   cursal NUMBER(6);
   null_salary EXCEPTION;
   PRAGMA EXCEPTION_INIT(null_salary, -20101);

BEGIN
   SELECT salary INTO cursal FROM Faculty WHERE fid = id;
   IF cursal IS NULL THEN
     raise_application_error(-20101, 'Salary is missing');
   ELSE
      UPDATE Faculty SET salary = cursal + 5000 WHERE fid = id;
   END IF;
EXCEPTION
      WHEN null_salary THEN
       DBMS_OUTPUT.PUT_LINE('No salary exists');
END;

CREATE OR REPLACE PROCEDURE raise_salary (id Faculty.fid%TYPE)
 IS
   cursal NUMBER(6);
   null_salary EXCEPTION;
   PRAGMA EXCEPTION_INIT(null_salary, -20101);

BEGIN
   SELECT salary INTO cursal FROM Faculty WHERE fid = id;
   IF cursal IS NULL THEN
   raise null_salary;

   ELSE
      UPDATE Faculty SET salary = cursal + 5000 WHERE fid = id;
   END IF;
EXCEPTION
      WHEN null_salary THEN
raise_application_error(-20101, 'Salary is missing');
END;
```

RAISE_APPLICATION_ERROR ( )

RAISE_APPLICATION_ERROR is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

PRAGMA refers to a compiler directive or "hint" it is used to provide an instruction to the compiler.

PL/SQL Packages

A package is an encapsulated collection of related program objects (for example, procedures, functions, variables, constants, cursors, and exceptions) stored together in the database.

Create the package header

```
create or replace package package_test IS
   function test_function return varchar2;
   procedure test_procedure (name IN varchar2);
END package_test;
```

Create the package bodies

```
create or replace package body package_test IS

  function test_function return varchar2 IS
  BEGIN
    return 'Function Test!!';
  END;

  procedure test_procedure (name IN varchar2) IS
  BEGIN
    dbms_output.put_line('Procedure Test:' || name );
  END;

END package_test;
/
```

Calling a procedure from a package

```
exec package_test.test_procedure('JNF');
```

Calling a function from a package

```
select package_test.test_function from dual;
```

Remove a package

```
drop package package_test;
```

Week XI

(Practice Triggers)

TRIGGERS

Similar to stored procedures and functions.

- Contains a Declaration, Executable, and Exception sections

Differences

- Triggers are not executed explicitly; they are implicitly executed when a triggering event occurs.
- Triggers do not accept parameters
- Triggering events are fired by DML Statements (INSERTs, UPDATEs, or DELETEs) against tables or views AND certain system events
- Events could be : BEFORE|AFTER INSERT|UPDATE|DELETE ON <tableName>

Row vs Statement Level Trigger

Row level: activated once per modified tuple
Statement level: activate once per SQL statement

Row level triggers can access new data, statement level triggers cannot always do that (depends on DBMS).
Statement level triggers will be more efficient if we do not need to make row-specific decisions

CREATE [OR REPLACE] TRIGGER < *trigger_name*>
{AFTER | BEFORE | INSTEAD OF} {INSERT | DELETE | UPDATE } [OF <*col name*>]
ON <*table_name (or view_name)* >
[REFERENCING [NEW AS < new row name>] [OLD AS <old row name>]]
[FOR EACH ROW [WHEN trigger_condition]]
DECLARE (optional)
BEGIN (mandatory)
Executes only when trigger_condition is TRUE on a ROW LEVEL TRIGGER
EXCEPTION (optional)
Exception Section
END; (mandatory)

New, old → special variables referring to new and old tuples. They are precede by a colon' :' .
Note: *a_trigger_event may be any combination of an INSERT, DELETE, and/or UPDATE on a table or view*

Ensures that any new faculty inserted has salary >= 60000

```
CREATE OR REPLACE TRIGGER SalaryCheck BEFORE INSERT ON Faculty
     FOR EACH ROW
BEGIN
      IF (:new.salary < 60000)
            THEN RAISE_APPLICATION_ERROR (-20004,
      'Violation of Minimum Professor Salary');
      END IF;
END;
```

```
SQL> INSERT INTO FACULTY VALUES ( 9 , 'III' , 'P' , 50000 , 'CSE' );
INSERT INTO FACULTY VALUES ( 9 , 'III' , 'P' , 50000 , 'CSE' )
       *
ERROR at line 1:
ORA-20004: Violation of Minimum Professor Salary
ORA-06512: at "SYSTEM.SALARYCHECK", line 3
ORA-04088: error during execution of trigger 'SYSTEM.SALARYCHECK'
```

:new – refers to the new tuple inserted

Conditions can refer to old/new values of tuples modified by the statement activating the trigger.

Ensure that salary does not decrease

```
CREATE TRIGGER SalaryCheckUP
     BEFORE UPDATE ON Faculty
      REFERENCING OLD AS oldTuple NEW as newTuple
FOR EACH ROW
WHEN (newTuple.salary < oldTuple.salary)
BEGIN
      RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing !!');
END;
.
run;
```

```
SQL> UPDATE FACULTY SET SALARY = 40000 WHERE FID = 8;
UPDATE FACULTY SET SALARY = 40000 WHERE FID = 8
            *
ERROR at line 1:
ORA-20004: Salary Decreasing !!
ORA-06512: at "SYSTEM.SALARYCHECKUP", line 2
ORA-04088: error during execution of trigger 'SYSTEM.SALARYCHECKUP'
```

Combining multiple events into one trigger

```
CREATE TRIGGER salaryCheckCOM
AFTER INSERT OR UPDATE ON Faculty
```

```
FOR EACH ROW
BEGIN

IF (INSERTING AND :new.salary < 60000) THEN
       RAISE_APPLICATION_ERROR (-20004, 'below min salary');
 END IF;

IF (UPDATING AND :new.salary < :old.salary) THEN
      RAISE_APPLICATION_ERROR (-20004, 'Salary Decreasing!!');
END IF;

END;
```

Note:

- **ORA-04091**: mutating relation problem

  In a **row level trigger**, you **cannot** have the body refer to the table specified in the event

- Also INSTEAD OF triggers can be specified on views

To Show Compilation Errors

SQL> SHOW ERRORS **TRIGGER** MY_TRIGGER

To drop triggers

Drop trigger <trigger_name> ;

Alter trigger <trigger_name> {disable| enable}

Select <trigger_name> from user-triggers ;