

RAJALAKSHMI ENGINEERING COLLEGE
RAJALAKSHMI NAGAR, THANDALAM – 602 105



RAJALAKSHMI
ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

**CS23332 DATABASE MANAGEMENT
SYSTEMS LAB**

Laboratory Record Note Book

Name:.... Lakshaya S.....

Year/Branch / Section : IInd B.E.Computer Science & Engineering –‘C’

University Register No. :2116230701160.....

College Roll No. :230701160.....

Semester :III.....

Academic Year :2024-25.....

CS23332 DATABASE MANAGEMENT SYSTEMS

NAME	Lakshaya S
ROLL NO	230701160
DEPT	B.E.Computer Science & Engineering
SEC	‘C’

SQL Statements

1. Data Retrieval(DR)
2. Data Manipulation Language(DML)
3. Data Definition Language(DDL)
4. Data Control Language(DCL)
5. Transaction Control Language(TCL)

TYPE	STATEMENT	DESCRIPTION
DR	SELECT	Retrieves the data from the database
DML	1.INSERT 2.UPDATE 3.DELETE 4.MERGE	Enter new rows, changes existing rows, removes unwanted rows from tables in the database respectively.
DDL	1.CREATE 2.ALTER 3.DROP 4.RENAME 5.TRUNCATE	Sets up, changes and removes data structures from tables.
TCL	1.COMMIT 2.ROLLBACK 3.SAVEPOINT	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions.
DCL	1.GRANT 2.RREVOKE	Gives or removes access rights to both the oracle database and the structures within it.

DATA TYPES

1. Character Data types:

- Char – fixed length character string that can varies between 1-2000 bytes
- Varchar / Varchar2 – variable length character string, size ranges from 1-4000 bytes.it saves the disk space(only length of the entered value will be assigned as the size of column)
- Long - variable length character string, maximum size is 2 GB

2. Number Data types : Can store +ve, -ve, zero, fixed point, floating point with 38 precision.

- Number – {p=38,s=0}
- Number(p) - fixed point
- Number(p,s) –floating point (p=1 to 38,s= -84 to 127)

3. **Date Time Data type:** used to store date and time in the table.
- DB uses its own format of storing in fixed length of 7 bytes for century, date, month, year, hour, minutes, and seconds.
 - Default data type is —dd-mon-yy||
 - New Date time data types have been introduced. They are
TIMESTAMP-Date with fractional seconds
INTERVAL YEAR TO MONTH-stored as an interval of years and months

INTERVAL DAY TO SECOND-stored as o interval of days to hour's minutes and seconds
4. **Raw Data type:** used to store byte oriented data like binary data and byte string.
5. **Other :**
- CLOB – stores character object with single byte character.
 - BLOB – stores large binary objects such as graphics, video, sounds.
 - BFILE – stores file pointers to the LOB's.

Creating and Managing Tables

OBJECTIVE

After the completion of this exercise, students should be able to do the following:

- ☐ Create tables
- ☐ Describing the data types that can be used when specifying column definition
- ☐ Alter table definitions
- ☐ Drop, rename, and truncate tables

NAMING RULES

Table names and column names:

- Must begin with a letter
- Must be 1-30 characters long
- Must contain only A-Z, a-z, 0-9, _, \$, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an oracle server reserve words
- 2 different tables should not have same name.
- Should specify a unique column name.
- Should specify proper data type along with width
- Can include —not null|| condition when needed. By default it is __null'.

OBJECTIVE

After, the completion of this exercise the students will be able to do the following

- Describe each DML statement
- Insert rows into tables
- Update rows into table
- Delete rows from table
- Control Transactions

A DML statement is executed when you:

- Add new rows to a table
- Modify existing rows
- Removing existing rows

A transaction consists of a collection of DML statements that form a logical unit of work.

The ALTER TABLE Statement

The ALTER statement is used to

- Add a new column
- Modify an existing column
- Define a default value to the new column
- Drop a column
- To include or drop integrity constraint.

DROPPING A TABLE

- All data and structure in the table is deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- Cannot roll back the drop table statement

To Add a New Row

INSERT Statement

Syntax

INSERT INTO table_name VALUES (column1 values, column2 values, ..., columnn values);

Example:

INSERT INTO department (70, 'Public relations', 100, 1700);

Inserting rows with null values

Implicit Method: (Omit the column)

INSERT INTO department VALUES (30, 'purchasing');

Explicit Method: (Specify NULL keyword)

```
INSERT INTO department VALUES (100,'finance', NULL, NULL);
```

Inserting Special Values

Example:

Using SYSDATE

```
INSERT INTO employees VALUES (113,'louis', _popp', _lpopp', '5151244567',SYSDATE,
_ac_account', 6900, NULL, 205, 100);
```

Inserting Specific Date Values

Example:

```
INSERT INTO employees VALUES ( 114,'den', _raphealy', _drapheal', _5151274561',
TO_DATE('feb 3,1999','mon, dd ,yyyy'), _ac_account', 11000,100,30);
```

To Insert Multiple Rows

& is the placeholder for the variable value

Example:

```
INSERT INTO department VALUES (&dept_id, &dept_name, &location);
```

Copying Rows from another table

☐ Using Subquery

Example:

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, Last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE _%REP');
```

CHANGING DATA IN A TABLE

UPDATE Statement

Syntax1: (to update specific rows)

```
UPDATE table_name SET column=value WHERE condition;
```

Syntax 2: (To updae all rows)

```
UPDATE table_name SET column=value;
```

Updating columns with a subquery

```
UPDATE employees  
SET job_id= (SELECT job_id  
FROM employees  
WHERE employee_id=205)  
WHERE employee_id=114;
```

REMOVING A ROW FROM A TABLE

DELETE STATEMENT

Syntax

```
DELETE FROM table_name WHERE conditions;
```

Example:

```
DELETE FROM department WHERE dept_name='finance';
```

Ex.No.: 1		CREATION OF BASE TABLE AND DML OPERATIONS
Date:	1.8.24	

AIM:

ALGORITHM:

STEP-1: Start.

STEP-2: Create a base Table

Syntax:

CREATE TABLE <table name> (column1 type, column2 type, ...);

STEP-3: Describe the Table structure

Syntax:

DESC <table name>

STEP-4: Add a new row to a Table using INSERT statement.

Syntax:

- INSERT INTO <table name> VALUES (value1, value2..);
- INSERT INTO <table name> (column1, column2..) VALUES (value1, value2..);
- INSERT INTO <table name>VALUES (&column1,'&column');

STEP-5: Modify the existing rows in the base Table with UPDATE statement.

Syntax:

UPDATE <table name> SET column1=value, column2 = 'value'
WHERE (condition);

STEP-6: Remove the existing rows from the Table using DELETE statement.

Syntax:

DELETE FROM <table name> WHERE <condition>;

STEP-7: Perform a Query using SELECT statement.

Syntax:

SELECT [DISTINCT] {*,<column1,...>} FROM <table name>
WHERE <condition>;

STEP-8: The truncate command deletes all rows from the table. Only the structure of the table remains.

Syntax:

```
TRUNCATE TABLE <table name>;
```

STEP-9: Alter the existing table using ALTER statement.

Syntax:

Add Column:

```
ALTER TABLE <table name> ADD (column data type  
[DEFAULTexpr][,column data type]);
```

Modify Column:

```
ALTER TABLE <table name> MODIFY (column data type  
[DEFAULT expr], [,column data type]);
```

Drop Column:

```
ALTER TABLE <table name> DROP COLUMN <column name>;
```

STEP-10: To drop the entire table using DROP statement.

Syntax:

```
DROP TABLE <table name>;
```

STEP-11: Exit.

Create MY_EMPLOYEE table with the following structure

```
CREATE TABLE MY_EMPLOYEE (  
  ID NUMBER(4) NOT NULL,  
  Last_name VARCHAR2(25),  
  First_name VARCHAR2(25),  
  Userid VARCHAR2(25),  
  Salary NUMBER(9,2),  
  CONSTRAINT pk_employee PRIMARY KEY (ID)  
);
```

NAME	NULL?	TYPE
ID	Not null	Number(4)
Last_name		Varchar(25)
First_name		Varchar(25)
Userid		Varchar(25)
Salary		Number(9,2)

1. Add the first and second rows data to MY_EMPLOYEE table from the following sample data.

ID	Last_name	First_name	Userid	salary
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	Cnewman	750
5	Ropebur	Audrey	aropebur	1550

```

INSERT INTO MY_EMPLOYEE (ID, Last_name, First_name, Userid, Salary)
VALUES (1, 'Patel', 'Ralph', 'rpatel', 895);
INSERT INTO MY_EMPLOYEE (ID, Last_name, First_name, Userid, Salary)
VALUES (2, 'Dancs', 'Betty', 'bdancs', 860);

```

2. Display the table with values.
SELECT * FROM MY_EMPLOYEE;

3. Populate the next two rows of data from the sample data. Concatenate the first letter of the first_name with the first seven characters of the last_name to produce Userid.

```

INSERT INTO MY_EMPLOYEE (ID, Last_name, First_name, Userid, Salary)
VALUES (3, 'Biri', 'Ben', NULL, 1100);
INSERT INTO MY_EMPLOYEE (ID, Last_name, First_name, Userid, Salary)
VALUES (4, 'Newman', 'Chad', NULL, 750);
UPDATE MY_EMPLOYEE
SET Userid = LOWER(CONCAT(SUBSTR(First_name, 1, 1), SUBSTR(Last_name, 1, 7)))
WHERE ID = 3 OR ID = 4;

```

4. Delete Betty dancs from MY_EMPLOYEE table.

```
DELETE FROM MY_EMPLOYEE  
WHERE First_name = 'Betty' AND Last_name = 'Dancs';
```

Empty the fourth row of the emp table.

```
UPDATE MY_EMPLOYEE  
SET Last_name = NULL, First_name = NULL, Userid = NULL, Salary = NULL  
WHERE ID = 4;
```

1. Make the data additions permanent.

```
COMMIT;
```

Change the last name of employee 3 to Drexler.

```
UPDATE MY_EMPLOYEE  
SET Last_name = 'Drexler'  
WHERE ID = 3;
```

1. Change the salary to 1000 for all the employees with a salary less than 900.

```
UPDATE MY_EMPLOYEE  
SET Salary = 1000  
WHERE Salary < 900;
```

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 2		DATA MANIPULATIONS
Date:	10.8.24	

Create the following tables with the given structure.

EMPLOYEES TABLE

NAME	NULL?	TYPE
Employee_id	Not null	Number(6)
First_Name		Varchar(20)
Last_Name	Not null	Varchar(25)
Email	Not null	Varchar(25)
Phone_Number		Varchar(20)
Hire_date	Not null	Date
Job_id	Not null	Varchar(10)
Salary		Number(8,2)
Commission_pct		Number(2,2)
Manager_id		Number(6)
Department_id		Number(4)

(a) Find out the employee id, names, salaries of all the employees

```
SELECT Employee_id, First_Name, Last_Name, Salary
FROM EMPLOYEES;
```

(b) List out the employees who works under manager 100

```
SELECT Employee_id, First_Name, Last_Name
FROM EMPLOYEES
WHERE Manager_id = 100;
```

(c) Find the names of the employees who have a salary greater than or equal to 4800

```
SELECT First_Name, Last_Name
```

FROM EMPLOYEES

WHERE Salary >= 4800;

(d) List out the employees whose last name is _AUSTIN‘

SELECT First_Name, Last_Name

FROM EMPLOYEES

WHERE Last_Name = 'AUSTIN';

(e) Find the names of the employees who works in departments 60,70 and 80

SELECT First_Name, Last_Name

FROM EMPLOYEES

WHERE Department_id IN (60, 70, 80);

(f) Display the unique Manager_Id.

SELECT DISTINCT Manager_id

FROM EMPLOYEES;

Create an Emp table with the following fields: (EmpNo, EmpName, Job,Basic, DA, HRA,PF, GrossPay, NetPay) (Calculate DA as 30% of Basic and HRA as 40% of Basic)

(a) Insert Five Records and calculate GrossPay and NetPay.

INSERT INTO EMP (EmpNo, EmpName, Job, Basic, Department_id)

VALUES (1, 'John Doe', 'Manager', 5000, 60);

INSERT INTO EMP (EmpNo, EmpName, Job, Basic, Department_id)

VALUES (2, 'Jane Austin', 'Clerk', 4000, 70);

INSERT INTO EMP (EmpNo, EmpName, Job, Basic, Department_id)

VALUES (3, 'Mark Smith', 'Sales', 3500, 80);

INSERT INTO EMP (EmpNo, EmpName, Job, Basic, Department_id)

VALUES (4, 'Chad Newman', 'Manager', 6000, 60);

INSERT INTO EMP (EmpNo, EmpName, Job, Basic, Department_id)

VALUES (5, 'Betty Austin', 'HR', 3900, 70);

UPDATE EMP

SET

DA = 0.30 * Basic,

```
HRA = 0.40 * Basic,  
PF = 0.12 * Basic;  
UPDATE EMP  
SET  
GrossPay = Basic + DA + HRA;  
UPDATE EMP  
SET  
NetPay = GrossPay - PF;
```

(b) Display the employees whose Basic is lowest in each department.

```
SELECT *  
FROM EMP e  
WHERE Basic = (  
SELECT MIN(Basic)  
FROM EMP  
WHERE Department_id = e.Department_id  
);
```

(c) If Net Pay is less than

```
SELECT EmpName, NetPay  
FROM EMP  
WHERE NetPay < 7500;
```

DEPARTMENT TABLE

NAME	NULL?	TYPE
Dept_id	Not null	Number(6)
Dept_name	Not null	Varchar(20)
Manager_id		Number(6)
Location_id		Number(4)

JOB_GRADE TABLE

NAME	NULL?	TYPE
Grade_level		Varchar(2)
Lowest_sal		Number
Highest_sal		Number

LOCATION TABLE

NAME	NULL?	TYPE
Location_id	Not null	Number(4)
St_addr		Varchar(40)
Postal_code		Varchar(12)
City	Not null	Varchar(30)
State_province		Varchar(25)
Country_id		Char(2)

1. Create the DEPT table based on the DEPARTMENT following the table instance chart below. Confirm that the table is created.

Column name	ID	NAME
Key Type		
Nulls/Unique		
FK table		
FK column		
Data Type	Number	Varchar2
Length	7	25


```
CREATE TABLE DEPT (
  ID NUMBER(7),
  NAME VARCHAR2(25),
  CONSTRAINT pk_dept PRIMARY KEY (ID)
);
```

2. Create the EMP table based on the following instance chart. Confirm that the table is created.

```
CREATE TABLE EMP (
  ID NUMBER(7),
  LAST_NAME VARCHAR2(25),
  FIRST_NAME VARCHAR2(25),
  DEPT_ID NUMBER(7),
  CONSTRAINT pk_emp PRIMARY KEY (ID)
);
```

Column name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK table				
FK column				
Data Type	Number	Varchar2	Varchar2	Number
Length	7	25	25	7

- 3 Modify the EMP table to allow for longer employee last names. Confirm the modification.(Hint: Increase the size to 50)

```
ALTER TABLE EMP
MODIFY LAST_NAME VARCHAR2(50);
```

- 4 Create the EMPLOYEES2 table based on the structure of EMPLOYEES table. Include Only the Employee_id, First_name, Last_name, Salary and Dept_id coloumns. Name the columns Id, First_name, Last_name, salary and Dept_id respectively.

```
CREATE TABLE EMPLOYEES2 AS
SELECT Employee_id AS Id, First_Name, Last_Name, Salary, Department_id AS Dept_id
FROM EMPLOYEES;
```

- 5 Drop the EMP table.

```
DROP TABLE EMP;
```

- 6 Rename the EMPLOYEES2 table as EMP.

```
ALTER TABLE EMPLOYEES2  
RENAME TO EMP;
```

- 7 Add a comment on DEPT and EMP tables. Confirm the modification by describing the table.

```
COMMENT ON TABLE DEPT IS 'Department Table';  
COMMENT ON TABLE EMP IS 'Employees Table';  
DESC DEPT;  
DESC EMP;
```

- 8 Drop the First_name column from the EMP table and confirm it.

```
ALTER TABLE EMP  
DROP COLUMN First_Name;  
DESC EMP;
```

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 3		WRITING BASIC SQL SELECT STATEMENTS
Date:	17.8.24	

OBJECTIVES

After the completion of this exercise, the students will be able to do the following:

- List the capabilities of SQL SELECT Statement
- Execute a basic SELECT statement

Capabilities of SQL SELECT statement

A SELECT statement retrieves information from the database. Using a select statement, we can perform

- ✓ Projection: To choose the columns in a table
- ✓ Selection: To choose the rows in a table
- ✓ Joining: To bring together the data that is stored in different tables

Basic SELECT Statement

Syntax

```
SELECT *|DISTINCT Column_ name| alias
`      FROM table_name;
```

NOTE:

DISTINCT—Suppress the duplicates.

Alias—gives selected columns different

headings. **Example: 1**

```
SELECT * FROM departments;
```

Example: 2

```
SELECT location_id, department_id FROM departments;
```

Writing SQL Statements

- SQL statements are not case sensitive
- SQL statements can be on one or more lines.

- Keywords cannot be abbreviated or split across lines
- Clauses are usually placed on separate lines
- Indents are used to enhance readability

Using Arithmetic Expressions

Basic Arithmetic operators like *, /, +, - can be used

Example:1

```
SELECT last_name, salary, salary+300 FROM employees;
```

Example:2

```
SELECT last_name, salary, 12*salary+100 FROM employees;
```

The statement is not same as

```
SELECT last_name, salary, 12*(salary+100) FROM employees;
```

Example:3

```
SELECT last_name, job_id, salary, commission_pct FROM employees;
```

Example:4

```
SELECT last_name, job_id, salary, 12*salary*commission_pct FROM employees;
```

Using Column Alias

- To rename a column heading with or without AS keyword.

Example:1

```
SELECT last_name AS Name
FROM employees;
```

Example: 2

```
SELECT last_name —Name|| salary*12 —Annual Salary —
FROM employees;
```

Concatenation Operator

- Concatenates columns or character strings to other columns
- Represented by two vertical bars (||)
- Creates a resultant column that is a character expression

Example:

```
SELECT last_name||job_id AS —EMPLOYEES JOB|| FROM employees;
```

Using Literal Character String

- A literal is a character, a number, or a date included in the SELECT list.
- Date and character literal values must be enclosed within single quotation marks.

Example:

```
SELECT last_name||'is a'||job_id AS —EMPLOYEES JOB|| FROM employees;
```

Eliminating Duplicate Rows

- Using DISTINCT keyword.

Example:

```
SELECT DISTINCT department_id FROM employees;
```

Displaying Table Structure

- Using DESC keyword.

Syntax

```
DESC table_name;
```

Example:

```
DESC employees;
```

Find the Solution for the following:

True OR False

1. The following statement executes successfully.

Identify the Errors

```
SELECT employee_id, last_name  
      sal*12 ANNUAL SALARY  
FROM employees;
```

```
SELECT Employee_id, Last_Name, Salary * 12 AS "ANNUAL SALARY"  
FROM EMPLOYEES;
```

Queries

2. Show the structure of departments the table. Select all the data from it.

```
DESC departments;
```

```
SELECT * FROM departments;
```

3. Create a query to display the last name, job code, hire date, and employee number for each employee, with employee number appearing first.

```
SELECT employee_id, last_name, job_id, hire_date  
FROM employees;
```

4. Provide an alias STARTDATE for the hire date.

```
SELECT employee_id, last_name, job_id, hire_date AS "STARTDATE"  
FROM employees;
```

5. Create a query to display unique job codes from the employee table.

```
SELECT DISTINCT job_id  
FROM employees;
```

6. Display the last name concatenated with the job ID , separated by a comma and space, and name the column EMPLOYEE and TITLE.

```
SELECT last_name || ', ' || job_id AS "EMPLOYEE and TITLE"  
FROM employees;
```

7. Create a query to display all the data from the employees table. Separate each column by a comma. Name the column THE_OUTPUT.

```
SELECT employee_id || ', ' || first_name || ', ' || last_name || ', ' || email || ', ' ||  
phone_number || ', ' || hire_date || ', ' || job_id || ', ' || salary || ', ' || commission_pct || ', ' ||  
manager_id || ', ' || department_id AS "THE_OUTPUT"  
FROM employees
```

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 4		WORKING WITH CONSTRAINTS
Date:	19.8.24	

OBJECTIVE

After the completion of this exercise the students should be able to do the following

- Describe the constraints
- Create and maintain the constraints

What are Integrity constraints?

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies

The following types of integrity constraints are valid

a) Domain Integrity

- ✓ NOT NULL
- ✓ CHECK

b) Entity Integrity

- ✓ UNIQUE
- ✓ PRIMARY KEY

c) Referential Integrity

- ✓ FOREIGN KEY

Constraints can be created in either of two ways

1. At the same time as the table is created
2. After the table has been created.

Defining Constraints

Create table tablename (column_name1 data_type constraints, column_name2 data_type constraints ...);

Example:

Create table employees (employee_id number(6), first_name varchar2(20), ..job_id varchar2(10), CONSTRAINT emp_emp_id_pk PRIMARY KEY (employee_id));

Domain Integrity

This constraint sets a range and any violations that takes place will prevent the user from performing the manipulation that caused the breach. It includes:

NOT NULL Constraint

While creating tables, by default the rows can have null value. the enforcement of not null constraint in a table ensure that the table contains values.

Principle of null values:

- Setting null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
- A null value is not equivalent to a value of zero.
- A null value will always evaluate to null in any expression.
- When a column name is defined as not null, that column becomes a mandatory i.e., the user has to enter data into it.
- Not null Integrity constraint cannot be defined using the alter table command when the table contain rows.

Example

```
CREATE TABLE employees (employee_id number (6), last_name varchar2(25) NOT NULL, salary number(8,2), commission_pct number(2,2), hire_date date constraint emp_hire_date_nn NOT NULL '...');
```

CHECK

Check constraint can be defined to allow only a particular range of values. when the manipulation violates this constraint, the record will be rejected. Check condition cannot contain sub queries.

```
CREATE TABLE employees (employee_id number (6), last_name varchar2 (25) NOT NULL, salary number(8,2), commission_pct number(2,2), hire_date date constraint emp_hire_date_nn NOT NULL '...', CONSTRAINT emp_salary_mi CHECK(salary > 0));
```

Entity Integrity

Maintains uniqueness in a record. An entity represents a table and each row of a table represents an instance of that entity. To identify each row in a table uniquely we need to use this constraint. There are 2 entity constraints:

a) Unique key constraint

It is used to ensure that information in the column for each record is unique, as with telephone or driver's license numbers. It prevents the duplication of value with rows of a specified column in a set of column. A column defined with the constraint can allow null value.

If unique key constraint is defined in more than one column i.e., combination of column cannot be specified. Maximum combination of columns that a composite unique key can contain is 16.

Example:

```
CREATE TABLE employees (employee_id number(6), last_name varchar2(25) NOT NULL, email varchar2(25), salary number(8,2), commission_pct number(2,2), hire_date date constraint emp_hire_date_nn NOT NULL' CONSTRAINT emp_email_uk UNIQUE(email));
```

PRIMARY KEY CONSTRAINT

A primary key avoids duplication of rows and does not allow null values. Can be defined on one or more columns in a table and is used to uniquely identify each row in a table. These values should never be changed and should never be null.

A table should have only one primary key. If a primary key constraint is assigned to more than one column or combination of column is said to be composite primary key, which can contain 16 columns.

Example:

```
CREATE TABLE employees (employee_id number(6) , last_name varchar2(25) NOT NULL, email varchar2(25), salary number(8,2), commission_pct number(2,2), hire_date date constraint emp_hire_date_nn NOT NULL, Constraint emp_id pk PRIMARY KEY (employee_id),CONSTRAINT emp_email_uk UNIQUE(email));
```

c) Referential Integrity

It enforces relationship between tables. To establish parent-child relationship between 2 tables having a common column definition, we make use of this constraint. To implement this, we should define the column in the parent table as primary key and same column in the child table as foreign key referring to the corresponding parent entry.

Foreign key

A column or combination of column included in the definition of referential integrity, which would refer to a referenced key.

Referenced key

It is a unique or primary key upon which is defined on a column belonging to the parent table.
Keywords:

FOREIGN KEY: Defines the column in the child table at the table level constraint.

REFERENCES: Identifies the table and column in the parent table.

ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted.

ON DELETE SET NULL: converts dependent foreign key values to null when the parent value is removed.

```
CREATE TABLE employees (employee_id number(6), last_name varchar2(25) NOT
NULL,email varchar2(25), salary number(8,2), commission_pct number(2,2), hire_date date
constraint emp_hire_date_nn NOT NULL, Constraint emp_id pk PRIMARY KEY
(employee_id), CONSTRAINT emp_email_uk UNIQUE(email), CONSTRAINT emp_dept_fk
FOREIGN KEY (department_id) references departments(dept_id));
```

ADDING A CONSTRAINT

Use the ALTER to

- Add or Drop a constraint, but not modify the structure
- Enable or Disable the constraints
- Add a not null constraint by using the Modify clause

Syntax

```
ALTER TABLE table name ADD CONSTRAINT Cons_name type(column name);
```

Example:

```
ALTER TABLE employees ADD CONSTRAINT emp_manager_fk FOREIGN KEY
(manager_id) REFERENCES employees (employee_id);
```

DROPPING A CONSTRAINT

Example:

```
ALTER TABLE employees DROP CONSTRAINT emp_manager_fk;
```

CASCADE IN DROP

- The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

Syntax

```
ALTER TABLE departments DROP PRIMARY KEY|UNIQUE (column)| CONSTRAINT
constraint_name CASCADE;
```

DISABLING CONSTRAINTS

- Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint
- Apply the CASCADE option to disable dependent integrity constraints.

Example

ALTER TABLE employees DISABLE CONSTRAINT emp_emp_id_pk CASCADE;

ENABLING CONSTRAINTS

- Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.

Example

ALTER TABLE employees ENABLE CONSTRAINT emp_emp_id_pk CASCADE;

CASCADING CONSTRAINTS

The CASCADE CONSTRAINTS clause is used along with the DROP column clause. It drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped Columns.

This clause also drops all multicolumn constraints defined on the dropped column.

Example:

Assume table TEST1 with the following structure

CREATE TABLE test1 (pk number PRIMARY KEY, fk number, col1 number,col2 number, CONSTRAINT fk_constraint FOREIGN KEY(fk) references test1, CONSTRAINT ck1 CHECK (pk>0 and col1>0), CONSTRAINT ck2 CHECK (col2>0));

An error is returned for the following statements

ALTER TABLE test1 DROP (pk);

ALTER TABLE test1 DROP (col1);

The above statement can be written with CASCADE CONSTRAINT

ALTER TABLE test 1 DROP(pk) CASCADE CONSTRAINTS;

(OR)

ALTER TABLE test 1 DROP(pk, fk, col1) CASCADE CONSTRAINTS;

VIEWING CONSTRAINTS

Query the USER_CONSTRAINTS table to view all the constraints definition and names.

Example:

```
SELECT constraint_name, constraint_type, search_condition FROM user_constraints  
WHERE table_name='employees';
```

Viewing the columns associated with constraints

```
SELECT constraint_name, constraint_type, FROM user_cons_columns  
WHERE table_name='employees';
```

Find the Solution for the following:

1. Add a table-level PRIMARY KEY constraint to the EMP table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

```
ALTER TABLE EMP  
ADD CONSTRAINT my_emp_id_pk PRIMARY KEY (Employee_id);
```

2. Create a PRIMAY KEY constraint to the DEPT table using the ID colum. The constraint should be named at creation. Name the constraint my_dept_id_pk.

```
ALTER TABLE DEPARTMENTS  
ADD CONSTRAINT my_dept_id_pk PRIMARY KEY (dept_id);
```

3. Add a column DEPT_ID to the EMP table. Add a foreign key reference on the EMP table that ensures that the employee is not assigned to nonexistent deparment. Name the constraint my_emp_dept_id_fk.

```
ALTER TABLE EMP
```

```
ADD DEPT_ID NUMBER(4);  
ALTER TABLE EMP  
ADD CONSTRAINT my_emp_dept_id_fk FOREIGN KEY (DEPT_ID)  
REFERENCES DEPARTMENTS(dept_id)
```

4. Modify the EMP table. Add a COMMISSION column of NUMBER data type, precision 2, scale 2. Add a constraint to the commission column that ensures that a commission value is greater than zero.

```
ALTER TABLE EMP
```

```
ADD COMMISSION NUMBER(2,2);
```

```
ALTER TABLE EMP
```

```
ADD CONSTRAINT chk_commission_gt_zero CHECK (COMMISSION > 0);
```

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 5		CREATING VIEWS
Date:	22.8.24	

After the completion of this exercise, students will be able to do the following:

- Describe a view
- Create, alter the definition of, and drop a view
- Retrieve data through a view
- Insert, update, and delete data through a view
- Create and use an inline view

View

A view is a logical table based on a table or another view. A view contains no data but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables.

Advantages of Views

- To restrict data access
- To make complex queries easy
- To provide data independence
- To present different views of the same data

Classification of views

1. Simple view
2. Complex view

Feature	Simple	Complex
No. of tables	One	One or more
Contains functions	No	Yes
Contains groups of data	No	Yes
DML operations thr' view	Yes	Not always

Creating a view

Syntax

CREATE OR REPLACE FORCE/NOFORCE VIEW view_name AS Subquery WITH CHECK OPTION CONSTRAINT constraint WITH READ ONLY CONSTRAINT constraint;

FORCE - Creates the view regardless of whether or not the base tables exist.

NOFORCE - Creates the view only if the base table exist.

WITH CHECK OPTION CONSTRAINT-specifies that only rows accessible to the view can be inserted or updated.

WITH READ ONLY CONSTRAINT-ensures that no DML operations can be performed on the view.

Example: 1 (Without using Column aliases)

Create a view EMPVU80 that contains details of employees in department80.

Example 2:

```
CREATE VIEW empvu80 AS SELECT employee_id, last_name, salary FROM employees
WHERE department_id=80;
```

Example:1 (Using column aliases)

```
CREATE VIEW salvu50
AS SELECT employee_id,id_number, last_name NAME, salary *12 ANN_SALARY
FROM employees
WHERE department_id=50;
```

Retrieving data from a view

Example:

```
SELECT * from salvu50;
```

Modifying a view

A view can be altered without dropping, re-creating.

Example: (Simple view)

Modify the EMPVU80 view by using CREATE OR REPLACE.

```
CREATE OR REPLACE VIEW empvu80 (id_number, name, sal, department_id)
AS SELECT employee_id,first_name, last_name, salary, department_id
FROM employees
WHERE department_id=80;
```

Example: (complex view)

```
CREATE VIEW dept_sum_vu (name, minsal, maxsal,avgsal)
AS    SELECT d.department_name, MIN(e.salary), MAX(e.salary), AVG(e.salary)
FROM employees e, department d
WHERE e.deparment_id=d.deparment_id
GROUP BY d.department_name;
```

Rules for performing DML operations on view

- Can perform operations on simple views
- Cannot remove a row if the view contains the following:
 - Group functions
 - Group By clause
 - Distinct keyword
- Cannot modify data in a view if it contains
 - Group functions
 - Group By clause
 - Distinct keyword
 - Columns contain by expressions
- Cannot add data thr' a view if it contains
 - Group functions
 - Group By clause
 - Distinct keyword
 - Columns contain by expressions
 - NOT NULL columns in the base table that are not selected by the view

Example: (Using the WITH CHECK OPTION clause)

```
CREATE OR REPLACE VIEW empvu20
AS    SELECT *
FROM employees
WHERE department_id=20
WITH CHECK OPTION CONSTRAINT empvu20_ck;
```

Note: Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

Example – (Execute this and note the error)

```
UPDATE empvu20 SET department_id=10 WHERE employee_id=201;
```

Denying DML operations

Use of WITH READ ONLY option.

Any attempt to perform a DML on any row in the view results in an oracle server error.

Try this code:

```
CREATE OR REPLACE VIEW empvu10(employee_number, employee_name, job_title)
AS SELECT employee_id, last_name, job_id
FROM employees
WHERE department_id=10
WITH READ ONLY;
```

Find the Solution for the following:

1. Create a view called EMPLOYEE_VU based on the employee numbers, employee names and department numbers from the EMPLOYEES table. Change the heading for the employee name to EMPLOYEE.

```
CREATE VIEW EMPLOYEE_VU AS
```

```
SELECT Employee_id,
```

```
First_Name || ' ' || Last_Name AS EMPLOYEE,
```

```
Dept_ID
```

```
FROM EMPLOYEES;
```

2. Display the contents of the EMPLOYEES_VU view.

```
SELECT * FROM EMPLOYEE_VU;
```

3. Select the view name and text from the USER_VIEWS data dictionary views.

```
SELECT VIEW_NAME, TEXT
FROM USER_VIEWS
WHERE VIEW_NAME = 'EMPLOYEE_VU';
```

4. Using your EMPLOYEES_VU view, enter a query to display all employees names and department.

```
SELECT EMPLOYEE, Dept_ID  
FROM EMPLOYEE_VU
```

Create a view named DEPT50 that contains the employee number, employee last names and department numbers for all employees in department 50. Label the view columns EMPNO, EMPLOYEE and DEPTNO. Do not allow an employee to be reassigned to another department through the view.

```
CREATE VIEW DEPT50 AS
SELECT Employee_id AS EMPNO,
       Last_Name AS EMPLOYEE,
       Dept_ID AS DEPTNO
FROM EMPLOYEES
WHERE Dept_ID = 50;
```

5. Display the structure and contents of the DEPT50 view.

```
DESC DEPT50;
SELECT * FROM DEPT50;
```

6. Attempt to reassign Matos to department 80.

```
UPDATE EMPLOYEES
SET Dept_ID = 80
WHERE Last_Name = 'Matos';
```

7. Create a view called SALARY_VU based on the employee last names, department names, salaries, and salary grades for all employees. Use the Employees, DEPARTMENTS and JOB_GRADE tables. Label the column Employee, Department, salary, and Grade respectively.

```
CREATE VIEW SALARY_VU AS
SELECT E.Last_Name AS Employee,
       D.dept_name AS Department,
       E.Salary AS Salary,
       J.Grade_level AS Grade
FROM EMPLOYEES E
JOIN DEPARTMENTS D ON E.Dept_ID = D.dept_id
JOIN JOB_GRADE J ON E.Salary BETWEEN J.Lowest_sal AND J.Highest_sal;
```

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 6		RESTRICTING AND SORTING DATA
Date:	29.8.24	

After the completion of this exercise, the students will be able to do the following:

- Limit the rows retrieved by the queries
- Sort the rows retrieved by the queries
-

Limiting the Rows selected

- Using WHERE clause
- Alias cannot be used in WHERE clause

Syntax

SELECT-----
FROM-----
WHERE condition;

Example:

```
SELECT employee_id,last_name, job_id, department_id FROM employees WHERE
department_id=90;
```

Character strings and Dates

Character strings and date values are enclosed in single quotation marks.

Character values are case sensitive and date values are format sensitive.

Example:

```
SELECT employee_id,last_name, job_id, department_id FROM employees
WHERE last_name='WHALEN';
```

Comparison Conditions

All relational operators can be used. (=, >, >=, <, <=, <>, !=)

Example:

```
SELECT last_name, salary
```



```
FROM employees
WHERE salary<=3000;
```

Other comparison conditions

Operator	Meaning
BETWEEN ...AND...	Between two values
IN	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null values

Example:1

```
SELECT last_name, salary
FROM employees
WHERE salary BETWEEN 2500 AND 3500;
```

Example:2

```
SELECT employee_id, last_name, salary , manager_id
FROM employees
WHERE manager_id IN (101, 100,201);
```

Example:3

- Use the LIKE condition to perform wildcard searches of valid string values.
- Two symbols can be used to construct the search string
- % denotes zero or more characters
- _ denotes one character

```
SELECT first_name, salary
FROM employees
WHERE first_name LIKE _%s‘;
```

Example:4

```
SELECT last_name, salary
FROM employees
WHERE last_name LIKE __o%‘;
```

Example:5

ESCAPE option-To have an exact match for the actual % and_ characters
To search for the string that contain _SA_‘

```
SELECT employee_id, first_name, salary, job_id
FROM employees
WHERE job_id LIKE _%sa\_%'ESCAPE'\';
```

Test for NULL

- Using IS NULL operator

Example:

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

Logical Conditions

All logical operators can be used.(AND,OR,NOT)

Example:1

```
SELECT employee_id, last_name, salary , job_id
FROM employees
WHERE salary >= 10000
AND job_id LIKE _%MAN%';
```

Example:2

```
SELECT employee_id, last_name, salary , job_id
FROM employees
WHERE salary >= 10000
OR job_id LIKE _%MAN%';
```

Example:3

```
SELECT employee_id, last_name, salary , job_id
FROM employees
WHERE job_id NOT IN (_it_prog', st_clerk', sa_rep');
```

Rules of Precedence

Order Evaluated	Operator
1	Arithmetic

2	Concatenation
3	Comparison
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	Logical NOT
7	Logical AND
8	Logical OR

Example:1

```
SELECT employee_id, last_name, salary , job_id
FROM employees
WHERE job_id ='sa_rep'
OR job_id ='ad_pres'
AND salary > 15000;
```

Example:2

```
SELECT employee_id, last_name, salary , job_id
FROM employees
WHERE (job_id ='sa_rep'
OR job_id ='ad_pres')
AND salary > 15000;
```

Sorting the rows

Using ORDER BY Clause

ASC-Ascending Order,Default

DESC-Descending order

Example:1

```
SELECT last_name, salary , job_id, department_id, hire_date
FROM employees
ORDER BY hire_date;
```

Example:2

```
SELECT last_name, salary , job_id, department_id, hire_date
FROM employees
ORDER BY hire_date DESC;
```

Example:3

Sorting by column alias

```
SELECT last_name, salary*12 annsal , job_id,department_id,hire_date
FROM employees
ORDER BY annsal;
```

Example:4

Sorting by Multiple columns

```
SELECT last_name, salary , job_id,department_id,hire_date
FROM employees
ORDER BY department_id, salary DESC;
```

Find the Solution for the following:

1. Create a query to display the last name and salary of employees earning more than 12000.

Query : select last_name,salary from Employee where salary>12000;

2. Create a query to display the employee last name and department number for employee number 176.

Query : select last_name,dept_id from Employee where emp_id = 176;

3. Create a query to display the last name and salary of employees whose salary is not in the range of 5000 and 12000. (hints: not between)

Query : select last_name,salary from Employee where salary not between 5000 and 12000;

4. Display the employee last name, job ID, and start date of employees hired between February 20,1998 and May 1,1998.order the query in ascending order by start date.(hints: between)

Query : select last_name,salary,job_id,hire_date from Employee where hire_date between '02-20-1998' and '05-1-1998' order by hire_date asc

5. Display the last name and department number of all employees in departments 20 and 50 in alphabetical order by name. (hints: in, orderby)

Query : select last_name , dept_id from Employee where dept_id in (20,50) order by last_name asc;

5. Display the last name and salary of all employees who earn between 5000 and 12000 and are in departments 20 and 50 in alphabetical order by name. Label the columns EMPLOYEE, MONTHLY SALARY respectively. (hints: between, in)

Query : select last_name as "Employee", salary as "Monthly Salary"
from Employee
where (salary between 5000 and 12000) and (dept_id in (20,50))
order by last_name;

6. Display the last name and hire date of every employee who was hired in 1994. (hints: like)

Query : select last_name, hire_date from Employee where hire_date like "1994%";

7. Display the last name and job title of all employees who do not have a manager. (hints: is null)

Query : select last_name, job_id from Employee where Manager is null;

8. Display the last name, salary, and commission for all employees who earn commissions. Sort data in descending order of salary and commissions.(hints: is not nul,orderby)

Query : select last_name,salary,commission from Employee where commission is not null
order by salary,commission desc;

9. Display the last name of all employees where the third letter of the name is *a*.(hints:like)

Query : SELECT last_name
FROM Employee
WHERE last_name LIKE '__a%';

10. Display the last name of all employees who have an *a* and an *e* in their last name.(hints: like)

Query : SELECT last_name
FROM Employee
WHERE last_name LIKE '%a%' and last_name like '%e%';

11. Display the last name and job and salary for all employees whose job is sales representative or stock clerk and whose salary is not equal to 2500 ,3500 or 7000.(hints:in,not in)

Query : select last_name , job , salary from Employee where job in (' sales representative','Stock clerk') and (salary not in (2500,3500,7000))

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 7		USING SET OPERATORS
Date:	2.9.24	

Objectives

After the completion this exercise, the students should be able to do the following:

- Describe set operators
- Use a set operator to combine multiple queries into a single query
- Control the order of rows returned

The set operators combine the results of two or more component queries into one result.

Queries containing set operators are called *compound queries*.

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and not selected in the second SELECT statement

The tables used in this lesson are:

- EMPLOYEES: Provides details regarding all current employees
- JOB_HISTORY: Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs

UNION Operator

Guidelines

- The number of columns and the data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- The IN operator has a higher precedence than the UNION operator.

- By default, the output is sorted in ascending order of the first column of the SELECT clause.

Example:

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT employee_id, job_id FROM employees UNION SELECT employee_id, job_id  
FROM job_history;
```

Example:

```
SELECT employee_id, job_id, department_id  
FROM employees  
UNION  
SELECT employee_id, job_id, department_id  
FROM job_history;
```

UNION ALL Operator

Guidelines

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL:

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

Example:

Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id  
FROM employees  
UNION ALL  
SELECT employee_id, job_id, department_id  
FROM job_history  
ORDER BY employee_id;
```

INTERSECT Operator

Guidelines

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- INTERSECT does not ignore NULL values.

Example:

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired (that is, they changed jobs but have now gone back to doing their original job).

```
SELECT employee_id, job_id FROM employees
INTERSECT
SELECT employee_id, job_id
FROM job_history;
```

Example

```
SELECT employee_id, job_id, department_id
FROM employees
INTERSECT
SELECT employee_id, job_id, department_id
FROM job_history;
```

MINUS Operator

Guidelines

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

Example:

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id, job_id
FROM employees
MINUS
SELECT employee_id, job_id
FROM job_history;
```

Find the Solution for the following:

1. The HR department needs a list of department IDs for departments that do not contain the job ID ST_CLERK. Use set operators to create this report.

Query : select dept_id from DEPT_W7 minus select dept_id from EMPLOYEE7 where job_title = 'stock clerk'

2. The HR department needs a list of countries that have no departments located in them. Display the country ID and the name of the countries. Use set operators to create this report.

Query :

```
SELECT c.c_id, c.c_name
FROM COUNTRY c
WHERE c.c_id IN (
    SELECT c.c_id
    FROM COUNTRY c
```

```
MINUS
SELECT d.c_id
```

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID using set operators.

```
Query : SELECT j.job_id, j.dept_id
FROM EMPLOYEE7 j
WHERE j.dept_id = 10
UNION
SELECT j.job_id, j.dept_id
FROM EMPLOYEE7 j
WHERE j.dept_id = 50
UNION
SELECT j.job_id, j.dept_id
FROM EMPLOYEE7 j
WHERE j.dept_id = 20
```

4. Create a report that lists the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired by the company (that is, they changed jobs but have now gone back to doing their original job).

```
Query : SELECT e.emp_id, e.job_id
FROM EMPLOYEE7 e
intersect
SELECT j.emp_id, j.job_id
FROM job_history7 j
```

5. The HR department needs a report with the following specifications:

- Last name and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to a department.
- Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them Write a compound query to

accomplish this.

- Query : SELECT lname, dept_id
- FROM EMPLOYEE7
- UNION
- SELECT DEPT_NAME, DEPT_ID
- FROM DEPT_W7;

Ex.No.: 8	WORKING WITH MULTIPLE TABLES
Date: 9.9.24	

Objective

After the completion of this exercise, the students will be able to do the following:

- Write SELECT statements to access data from more than one table using equality and nonequality joins
 - View data that generally does not meet a join condition by using outer joins
 - Join a table to itself by using a self join
- Sometimes you need to use data from more than one table.

Cartesian Products

- A Cartesian product is formed when:
 - A join condition is omitted
 - A join condition is invalid
 - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

A Cartesian product tends to generate a large number of rows, and the result is rarely useful. You should always include a valid join condition in a WHERE clause, unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

Example:

To displays employee last name and department name from the EMPLOYEES and DEPARTMENTS tables.

```
SELECT last_name, department_name dept_name
FROM employees, departments;
```

Types of Joins

- Equijoin
- Non-equijoin
- Outer join
- Self join
- Cross joins
- Natural joins
- Using clause
- Full or two sided outer joins
- Arbitrary join conditions for outer joins

Joining Tables Using Oracle Syntax

```
SELECT table1.column, table2.column
```

```
FROM table1, table2
WHERE table1.column1 = table2.column2;
```

Write the join condition in the WHERE clause.

- Prefix the column name with the table name when the same column name appears in more than one table.

Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join n tables together, you need a minimum of n-1 join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row

What is an Equijoin?

To determine an employee's department name, you compare the value in the DEPARTMENT_ID

column in the EMPLOYEES table with the DEPARTMENT_ID values in the DEPARTMENTS table.

The relationship between the EMPLOYEES and DEPARTMENTS tables is an equijoin—that is, values

in the DEPARTMENT_ID column on both tables must be equal. Frequently, this type of join involves

primary and foreign key complements.

Note: Equijoins are also called simple joins or inner joins

```
SELECT employees.employee_id, employees.last_name, employees.department_id,
```

```
departments.department_id, departments.location_id
```

```
FROM employees, departments
```

```
WHERE employees.department_id = departments.department_id;
```

Additional Search Conditions

Using the AND Operator

Example:

To display employee Matos' department number and department name, you need an additional condition in the WHERE clause.

```
SELECT last_name, employees.department_id,
department_name
```

```
FROM employees, departments
```

```
WHERE employees.department_id = departments.department_id AND last_name = 'Matos';
```

Qualifying Ambiguous

Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

Using Table Aliases

- Simplify queries by using table aliases.
- Improve performance by using table prefixes

Example:

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e , departments d  
WHERE  e.department_id = d.department_id;
```

Joining More than Two Tables

To join n tables together, you need a minimum of n-1 join conditions. For example, to join three tables, a minimum of two joins is required.

Example:

To display the last name, the department name, and the city for each employee, you have to join the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables.

```
SELECT e.last_name, d.department_name, l.city  
FROM   employees e, departments d, locations l  
WHERE  e.department_id = d.department_id  
AND    d.location_id = l.location_id;
```

Non-Equi Joins

A non-equijoin is a join condition containing something other than an equality operator. The relationship between the EMPLOYEES table and the JOB_GRADES table has an example of a non-equijoin. A relationship between the two tables is that the SALARY column in the EMPLOYEES table must be between the values in the LOWEST_SALARY and HIGHEST_SALARY columns of the JOB_GRADES table. The relationship is obtained using an operator other than equals (=).

Example:

```
SELECT e.last_name, e.salary, j.grade_level  
FROM   employees e, job_grades j  
WHERE  e.salary  
       BETWEEN j.lowest_sal AND j.highest_sal;
```

Outer Joins

Syntax

- You use an outer join to also see rows that do not meet the join condition.
- The Outer join operator is the plus sign (+).

```
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column(+) = table2.column;
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.column = table2.column(+);
```

The missing rows can be returned if an outer join operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is placed on the —side of the join that is deficient in information. This operator has the effect of creating one or more null rows, to which one or more rows from the nondeficient table can be joined.

Example:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e, departments d
WHERE e.department_id(+) = d.department_id ;
```

Outer Join Restrictions

- The outer join operator can appear on only one side of the expression—the side that has information missing. It returns those rows from one table that have no direct match in the other table.
- A condition involving an outer join cannot use the IN operator or be linked to another condition by the OR operator

Self Join

Sometimes you need to join a table to itself.

Example:

To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self join.

```
SELECT worker.last_name || ' works for '
|| manager.last_name
FROM employees worker, employees manager
WHERE worker.manager_id = manager.employee_id ;
```

Use a join to query data from more than one table.

```
SELECT table1.column, table2.column
```



```
FROM table1  
[CROSS JOIN table2] |  
[NATURAL JOIN table2] |  
[JOIN table2 USING (column_name)] |  
[JOIN table2  
ON(table1.column_name = table2.column_name)] |  
[LEFT|RIGHT|FULL OUTER JOIN table2  
ON (table1.column_name = table2.column_name)];
```

In the syntax:

table1.column Denotes the table and column from which data is retrieved

CROSS JOIN Returns a Cartesian product from the two tables

NATURAL JOIN Joins two tables based on the same column name

JOIN table USING column_name Performs an equijoin based on the column name

JOIN table ON table1.column_name Performs an equijoin based on the condition in the ON clause

= table2.column_name

LEFT/RIGHT/FULL OUTER

Creating Cross Joins

- The CROSS JOIN clause produces the crossproduct of two tables.
- This is the same as a Cartesian product between the two tables.

Example:

```
SELECT last_name, department_name  
FROM employees  
CROSS JOIN departments ;  
SELECT last_name, department_name  
FROM employees, departments;
```

Creating Natural Joins

- The NATURAL JOIN clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

Example:

```
SELECT department_id, department_name,  
location_id, city  
FROM departments  
NATURAL JOIN locations ;
```

LOCATIONS table is joined to the DEPARTMENT table by the LOCATION_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

Example:

```
SELECT department_id, department_name,  
location_id, city  
FROM departments  
NATURAL JOIN locations  
WHERE department_id IN (20, 50);
```

Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with the USING clause to specify the columns that should be used for an equijoin.
- Use the USING clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The NATURAL JOIN and USING clauses are mutually exclusive.

Example:

```
SELECT l.city, d.department_name  
FROM locations l JOIN departments d USING (location_id)  
WHERE location_id = 1400;  
EXAMPLE:
```

```
SELECT e.employee_id, e.last_name, d.location_id  
FROM employees e JOIN departments d  
USING (department_id) ;
```

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the ON clause is used.
- The join condition is separated from other searchconditions.
- The ON clause makes code easy to understand.

Example:

```
SELECT e.employee_id, e.last_name, e.department_id,  
d.department_id, d.location_id  
FROM employees e JOIN departments d  
ON (e.department_id = d.department_id);  
EXAMPLE:
```

```
SELECT e.last_name emp, m.last_name mgr  
FROM employees e JOIN employees m
```

ON (e.manager_id = m.employee_id);
INNER Versus OUTER Joins

- A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is a left (or right) outer join.
- A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.

LEFT OUTER JOIN

Example:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e
LEFT OUTER JOIN departments d
ON (e.department_id = d.department_id) ;
```

Example of LEFT OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, which is the left table even if there is no match in the DEPARTMENTS table.

This query was completed in earlier releases as follows:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e, departments d
WHERE d.department_id (+) = e.department_id;
```

RIGHT OUTER JOIN

Example:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e
RIGHT OUTER JOIN departments d
ON (e.department_id = d.department_id) ;
```

This query retrieves all rows in the DEPARTMENTS table, which is the right table even if there is no match in the EMPLOYEES table.

This query was completed in earlier releases as follows:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e, departments d
WHERE d.department_id = e.department_id (+);
```

FULL OUTER JOIN

Example:

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e
FULL OUTER JOIN departments d
ON (e.department_id = d.department_id) ;
```

This query retrieves all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

Find the Solution for the following:

1. Write a query to display the last name, department number, and department name for all employees.

Query : SELECT e.ename, dept_id, d.dept_name from EMPLOYEE8 e JOIN DEPT_W8 d using (dept_id)

2. Create a unique listing of all jobs that are in department 80. Include the location of the department in the output.

```
Query : SELECT j.job_id, j.job_name
FROM JOB_8 j
JOIN DEPT_W8 d ON (j.dept_id = d.dept_id)
JOIN COUNTRY8 c ON (d.c_id = c.c_id)
WHERE d.dept_id = 80;
```

3. Write a query to display the employee last name, department name, location ID, and city of all employees who earn a commission

```
Query : SELECT e.LNAME, d.DEPT_NAME, c.C_ID, c.CITY_NAME
FROM EMPLOYEE8 e
JOIN DEPT_W8 d ON (e.DEPT_ID = d.DEPT_ID)
JOIN COUNTRY8 c ON (d.C_ID = c.C_ID)
WHERE e.COMMISSION IS NOT NULL
```

3. Display the employee last name and department name for all employees who have an a(lowercase) in their last names. P

```
Query ; SELECT e.LNAME ,DEPT_ID
FROM EMPLOYEE8 e NATURAL JOIN DEPT_W8 d
WHERE e.LNAME LIKE '%a%'
```

5. Write a query to display the last name, job, department number, and department name for all employees who work in Toronto.

```
Query : SELECT DISTINCT e.LNAME , j.JOB_NAME , d.DEPT_ID
FROM EMPLOYEE8 e
JOIN JOB_8 j ON (j.JOB_ID = e.JOB_ID)
JOIN DEPT_W8 d ON (d.DEPT_ID = e.DEPT_ID)
JOIN COUNTRY8 c ON (c.C_ID=d.C_ID)
WHERE c.CITY_NAME = 'TORONTO'
```

6. Display the employee last name and employee number along with their manager's last name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, Respectively
Query : select e.emp_id as emp#,e.lname as Employee,m.m_id as Mgr# ,m.M_name as manager
from EMPLOYEE8 e left outer join MANAGER m on(m.M_ID=e.manager_id)

7. Modify lab4_6.sql to display all employees including King, who has no manager. Order the results by the employee number.

```
Query : SELECT e.fname || ' ' || e.lname AS Name
FROM EMPLOYEE8 e
LEFT JOIN EMPLOYEE8 m ON e.MANAGER_ID = m.EMP_ID
WHERE m.EMP_ID IS NULL
ORDER BY e.EMP_ID;
```

8. Create a query that displays employee last names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label

```
Query : SELECT e1.LNAME AS NAME , e1.DEPT_ID AS DEPARTMENT_NUMBER FROM
EMPLOYEE8 e1
join EMPLOYEE8 e2 on e1.dept_id=e2.dept_id
WHERE e2.emp_id = 2
```

9. Show the structure of the JOB_GRADES table. Create a query that displays the name, job, department name, salary, and grade for all employees

Query : DESC JOB_8

```
SELECT DISTINCT e.LNAME,e.salary, j.JOB_NAME, d.DEPT_NAME,JOB_ID FROM  
EMPLOYEE8 e  
INNER JOIN  DEPT_W8 d USING(DEPT_ID)  
INNER JOIN JOB_8 j USING(JOB_ID)
```

10. Create a query to display the name and hire date of any employee hired after employee Davies.

Query : select e1.lname,e1.hire_date from EMPLOYEE8 e1 join EMPLOYEE8 e2 on e1.hire_date>
e2.hire_date
where e2.lname = 'Davies'

11. Display the names and hire dates for all employees who were hired before their managers, along with their manager's names and hire dates. Label the columns Employee, Emp Hired, Manager, and Mgr Hired, respectively.

Query : SELECT e.fname || ' ' || e.lname AS employee, e.hire_date AS EMP_HIRED ,m.lname as
Manager , m.hire_date as mgr_hired
FROM Employee8 e
JOIN Employee8 m ON e.manager_id = m.emp_id
WHERE e.hire_date < m.hire_date;

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 9		SUB QUERIES
Date:	10.10.24	

Objectives

After completing this lesson, you should be able to do the following:

- Define subqueries
- Describe the types of problems that subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

Using a Subquery to Solve a Problem

Who has a salary greater than Abel's?

Main query:

Which employees have salaries greater than Abel's salary?

Subquery:

What is Abel's salary?

Subquery Syntax

SELECT select_list FROM table WHERE expr operator (SELECT select_list FROM table);

- The subquery (inner query) executes once before the main query (outer query).
- The result of the subquery is used by the main query.

A subquery is a **SELECT** statement that is embedded in a clause of another **SELECT** statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

operator includes a comparison condition such as **>**, **=**, or **IN**

Note: Comparison conditions fall into two classes: single-row operators

(>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL). statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query

Using a Subquery

```
SELECT last_name FROM employees WHERE salary > (SELECT salary FROM employees  
WHERE last_name = 'Abel');
```

The inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than this amount.

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The ORDER BY clause in the subquery is not needed unless you are performing Top-N analysis.
- Use single-row operators with single-row

subqueries, and use multiple-row operators with multiple-row subqueries.

Types of Subqueries

- Single-row subqueries: Queries that return only one row from the inner SELECT statement.
- Multiple-row subqueries: Queries that return more than one row from the inner SELECT statement.

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

Example

Display the employees whose job ID is the same as that of employee 141:

```
SELECT last_name, job_id FROM employees WHERE job_id = (SELECT job_id FROM  
employees  
WHERE employee_id = 141);
```

Displays employees whose job ID is the same as that of employee 141 and whose salary is greater than that of employee 143.

```
SELECT last_name, job_id, salary FROM employees WHERE job_id =(SELECT job_id FROM employees WHERE employee_id = 141) AND salary > (SELECT salary FROM employees WHERE employee_id = 143);
```

Using Group Functions in a Subquery

Displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.

```
SELECT last_name, job_id, salary FROM employees WHERE salary = (SELECT MIN(salary) FROM employees);
```

The HAVING Clause with Subqueries

- The Oracle server executes subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

Displays all the departments that have a minimum salary greater than that of department 50.

```
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) >
(SELECT MIN(salary)
FROM employees
WHERE department_id = 50);
```

Example

Find the job with the lowest average salary.

```
SELECT job_id, AVG(salary)
FROM employees
GROUP BY job_id
HAVING AVG(salary) = (SELECT MIN(AVG(salary))
FROM employees
GROUP BY job_id);
```

What Is Wrong in this Statements?

```
SELECT employee_id, last_name
FROM employees
```

```
WHERE salary =(SELECT MIN(salary) FROM employees GROUP BY department_id);
Will This Statement Return Rows?
SELECT last_name, job_id
FROM employees
WHERE job_id =(SELECT job_id FROM employees WHERE last_name = 'Haas');
```

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Example

Find the employees who earn the same salary as the minimum salary for each department.

```
SELECT last_name, salary, department_id FROM employees WHERE salary IN (SELECT
MIN(salary)
FROM employees GROUP BY department_id);
```

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary FROM employees WHERE salary < ANY
(SELECT salary FROM employees WHERE job_id = 'IT_PROG') AND job_id <> 'IT_PROG';
```

Displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is \$9,000.

< ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ALL (SELECT salary FROM employees WHERE job_id = 'IT_PROG')
AND job_id <> 'IT_PROG';
```

Displays employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

- ALL means more than the maximum, and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.

Null Values in a Subquery

```
SELECT emp.last_name FROM employees emp
```

```
WHERE emp.employee_id NOT IN (SELECT mgr.manager_id FROM employees mgr);
```

Notice that the null value as part of the results set of a subquery is not a problem if you use the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT emp.last_name  
FROM employees emp  
WHERE emp.employee_id IN (SELECT mgr.manager_id FROM employees mgr);
```

Display all employees who do not have any subordinates:

```
SELECT last_name FROM employees  
WHERE employee_id NOT IN (SELECT manager_id FROM employees WHERE manager_id  
IS NOT NULL);
```

Find the Solution for the following:

1. The HR department needs a query that prompts the user for an employee last name. The query then displays the last name and hire date of any employee in the same department as the employee whose name they supply (excluding that employee). For example, if the user enters Zlotkey, find all employees who work with Zlotkey (excluding Zlotkey).

Query : `SELECT LNAME,HIRE_DATE FROM EMPLOYEE7 WHERE DEPT_ID =
(SELECT DEPT_ID FROM EMPLOYEE7 WHERE FNAME = 'Zlotkey') and fname <>
'Zlotkey';`

2. Create a report that displays the employee number, last name, and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary.

Query : `SELECT EMP_ID ,LNAME,SALARY
FROM EMPLOYEE7
WHERE SALARY>(SELECT AVG(SALARY) FROM EMPLOYEE7)
ORDER BY SALARY`

3. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains a *u*.

Query : `SELECT EMP_ID ,LNAME FROM EMPLOYEE7 WHERE DEPT_ID IN (SELECT
DEPT_ID FROM EMPLOYEE7 WHERE LNAME LIKE '%u%')`

4. The HR department needs a report that displays the last name, department number, and job ID of all employees whose department location ID is 1700.

Query : SELECT LNAME,DEPT_ID,JOB_ID FROM EMPLOYEE7 WHERE DEPT_ID IN (SELECT DEPT_ID FROM DEPT_W7 WHERE C_ID = 1700)

5. Create a report for HR that displays the last name and salary of every employee who reports to King.

Query : SELECT LNAME,SALARY FROM EMPLOYEE7 WHERE EMP_ID IN (SELECT emp_id FROM EMPLOYEE7 WHERE REPORTING ='KING')

6. Create a report for HR that displays the department number, last name, and job ID for every employee in the Executive department.

Query : SELECT DEPT_ID , LNAME,JOB_ID FROM EMPLOYEE7 WHERE DEPT_ID = (SELECT DEPT_ID FROM DEPT_W7 WHERE DEPT_NAME = 'EXECUTIVE')

7. Modify the query 3 to display the employee number, last name, and salary of all employees who earn more than the average salary and who work in a department with any employee whose last name contains a *u*.

Query : SELECT EMP_ID ,LNAME FROM EMPLOYEE7 WHERE DEPT_ID IN (SELECT DEPT_ID FROM EMPLOYEE7 WHERE LNAME LIKE '%u%') AND SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE7)

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 10		AGGREGATING DATA USING GROUP FUNCTIONS
Date:	17.10.24	

Objectives

After the completion of this exercise, the students be will be able to do the following:

- Identify the available group functions
- Describe the use of group functions
- Group data by using the GROUP BY clause
- Include or exclude grouped rows by using the HAVING clause

What Are Group Functions?

Group functions operate on sets of rows to give one result per group

Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

Function	Description
AVG ([DISTINCT <u>ALL</u>] <i>n</i>)	Average value of <i>n</i> , ignoring null values
COUNT ({ * [DISTINCT <u>ALL</u>] <i>expr</i> })	Number of rows, where <i>expr</i> evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX ([DISTINCT <u>ALL</u>] <i>expr</i>)	Maximum value of <i>expr</i> , ignoring null values
MIN ([DISTINCT <u>ALL</u>] <i>expr</i>)	Minimum value of <i>expr</i> , ignoring null values
STDDEV ([DISTINCT <u>ALL</u>] <i>x</i>)	Standard deviation of <i>n</i> , ignoring null values
SUM ([DISTINCT <u>ALL</u>] <i>n</i>)	Sum values of <i>n</i> , ignoring null values
VARIANCE ([DISTINCT <u>ALL</u>] <i>x</i>)	Variance of <i>n</i> , ignoring null values

Group Functions: Syntax

```
SELECT [column,] group_function(column), ...
FROM table
[WHERE condition]
```

[GROUP BY *column*]

[ORDER BY *column*];

Guidelines for Using Group Functions

- DISTINCT makes the function consider only nonduplicate values; ALL makes it consider every value, including duplicates. The default is ALL and therefore does not need to be specified.
- The data types for the functions with an expr argument may be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions ignore null values.

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),  
MIN(salary), SUM(salary)  
FROM employees  
WHERE job_id LIKE '%REP%';
```

Using the MIN and MAX Functions

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT MIN(hire_date), MAX(hire_date)  
FROM employees;
```

You can use the MAX and MIN functions for numeric, character, and date data types. example displays the most junior and most senior employees.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetized list of all employees:

```
SELECT MIN(last_name), MAX(last_name)  
FROM employees;
```

Note: The AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric data types. MAX and MIN cannot be used with LOB or LONG data types.

Using the COUNT Function

COUNT(*) returns the number of rows in a table:

```
SELECT COUNT(*)  
FROM employees
```



```
WHERE department_id = 50;  
COUNT(expr) returns the number of rows with nonnull  
values for the expr:  
SELECT COUNT(commission_pct)  
FROM employees  
WHERE department_id = 80;
```

Using the DISTINCT Keyword

- COUNT(DISTINCT *expr*) returns the number of distinct non-null values of the *expr*.
- To display the number of distinct department values in the EMPLOYEES table:

```
SELECT COUNT(DISTINCT department_id) FROM employees;
```

Use the DISTINCT keyword to suppress the counting of any duplicate values in a column.

Group Functions and Null Values

Group functions ignore null values in the column:

```
SELECT AVG(commission_pct)  
FROM employees;
```

The NVL function forces group functions to include null values:

```
SELECT AVG(NVL(commission_pct, 0))  
FROM employees;
```

Creating Groups of Data

To divide the table of information into smaller groups. This can be done by using the GROUP BY clause.

GROUP BY Clause Syntax

```
SELECT column, group_function(column)  
FROM table  
[WHERE condition]  
[GROUP BY group_by_expression]  
[ORDER BY column];
```

In the syntax:

group_by_expression specifies columns whose values determine the basis for grouping rows

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well, *unless* the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.

Using the GROUP BY Clause

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id ;
```

The GROUP BY column does not have to be in the SELECT list.

```
SELECT AVG(salary) FROM employees GROUP BY department_id ;
```

You can use the group function in the ORDER BY clause:

```
SELECT department_id, AVG(salary) FROM employees GROUP BY department_id ORDER
BY AVG(salary);
```

Grouping by More Than One Column

```
SELECT department_id dept_id, job_id, SUM(salary) FROM employees
GROUP BY department_id, job_id ;
```

Illegal Queries Using Group Functions

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP

BY clause:

```
SELECT department_id, COUNT(last_name) FROM employees;
```

You can correct the error by adding the GROUP BY clause:

```
SELECT department_id, count(last_name) FROM employees GROUP BY department_id;
```

You cannot use the WHERE clause to restrict groups.

- You use the HAVING clause to restrict groups.
- You cannot use group functions in the WHERE clause.

```
SELECT department_id, AVG(salary) FROM employees WHERE AVG(salary) > 8000  
GROUP BY department_id;
```

You can correct the error in the example by using the HAVING clause to restrict groups:

```
SELECT department_id, AVG(salary) FROM employees  
HAVING AVG(salary) > 8000 GROUP BY department_id;
```

Restricting Group Results

With the HAVING Clause .When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

Using the HAVING Clause

```
SELECT department_id, MAX(salary) FROM employees  
GROUP BY department_id HAVING MAX(salary) > 10000 ;
```

The following example displays the department numbers and average salaries for those departments with a maximum salary that is greater than \$10,000:

```
SELECT department_id, AVG(salary) FROM employees GROUP BY department_id  
HAVING max(salary) > 10000;
```

Example displays the job ID and total monthly salary for each job that has a total payroll exceeding \$13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

```
SELECT job_id, SUM(salary) PAYROLL FROM employees WHERE job_id NOT LIKE  
'%REP%'  
GROUP BY job_id HAVING SUM(salary) > 13000 ORDER BY SUM(salary);
```

Nesting Group Functions

Display the maximum average salary:

Group functions can be nested to a depth of two. The slide example displays the maximum average salary.

```
SELECT MAX(AVG(salary)) FROM employees GROUP BY department_id;
```

Summary

In this exercise, students should have learned how to:

- Use the group functions COUNT, MAX, MIN, and AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

```
SELECT column, group_function
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING group_condition]
[ORDER BY column];
```

Find the Solution for the following:

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result per group.

True/False

2. Group functions include nulls in calculations.

True/False

3. The WHERE clause restricts rows prior to inclusion in a group calculation.

True/False

The HR department needs the following reports:

4. Find the highest, lowest, sum, and average salary of all employees. Label the columns Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number

Query : SELECT

MIN(SALARY) AS MINIMUM,

MAX(SALARY) AS MAXIMUM,

SUM(SALARY) AS SUM,

ROUND(AVG(SALARY),0) AS AVERAGE

FROM EMPLOYEE10;

Modify the above query to display the minimum, maximum, sum, and average salary for each job type.

```
Query : SELECT
    MIN(SALARY) AS MINIMUM,
    MAX(SALARY) AS MAXIMUM,
    SUM(SALARY) AS SUM,
    ROUND(AVG(SALARY)) AS AVERAGE,
    JOB_TITLE
FROM EMPLOYEE10
GROUP BY JOB_TITLE;
```

Write a query to display the number of people with the same job. Generalize the query so that the user in the HR department is prompted for a job title.

Query : SELECT COUNT(*) ,JOB_TITLE
FROM EMPLOYEE10
GROUP BY JOB_TITLE

Determine the number of managers without listing them. Label the column Number of Managers. *Hint: Use the MANAGER_ID column to determine the number of managers.*

Query : SELECT COUNT() AS ['NUMBER OF MANAGERS']
FROM EMPLOYEE10
WHERE MANAGER_ID IS NOT NULL*

5. Find the difference between the highest and lowest salaries. Label the column DIFFERENCE.

Query : SELECT MAX(SALARY) - MIN(SALARY) AS DIFFERENCE
FROM EMPLOYEE10

6. Create a report to display the manager number and the salary of the lowest-paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is \$6,000 or less. Sort the output in descending order of salary.

Query : SELECT MIN(SALARY) AS SALARY,MANAGER_ID
FROM EMPLOYEE10
WHERE MANAGER_ID IS NOT NULL
GROUP BY MANAGER_ID
HAVING MIN(SALARY)>6000
ORDER BY SALARY DESC

7. Create a query to display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

```
Query : SELECT COUNT(EMP_ID) AS TOTAL ,EXTRACT(YEAR FROM HIRE_DATE) AS  
YEAR  
FROM EMPLOYEE10 e1  
WHERE EXTRACT(YEAR FROM HIRE_DATE) IN (1995,1996,1997,1998)  
GROUP BY EXTRACT(YEAR FROM HIRE_DATE)  
UNION  
SELECT COUNT(EMP_ID),NULL  
FROM EMPLOYEE
```

Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

Query : SELECT
 DEPT_ID,
 SUM(CASE WHEN JOB_TITLE='manager' THEN SALARY ELSE 0 END) AS
 MANAGER_SALARY,
 SUM(CASE WHEN JOB_TITLE='Buisness Analyst' THEN SALARY ELSE 0 END)
 AS ANALYST_SALARY,
 SUM(CASE WHEN JOB_TITLE='STOCK-CLERK' THEN SALARY ELSE 0 END)
 AS CLERK_SALARY
 FROM EMPLOYEE10
 GROUP BY DEPT_ID

Write a query to display each department's name, location, number of employees, and the average salary for all the employees in that department. Label the column name-Location, Number of people, and salary respectively. Round the average salary to two decimal places.

Query : SELECT DEPT_NAME AS DEPARTMENT ,C_ID AS LOCATION,
 COUNT(EMP_ID) AS NO_OF_PEOPLE, ROUND(AVG(SALARY),2) AS SALARY
 FROM EMPLOYEE10
 GROUP BY DEPT_NAME ,C_ID

Evaluation Procedure	Marks awarded
Query(5)	
Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 11		PL SQL PROGRAMS
Date:	24.10.24	

PROGRAMS

TO DISPLAY HELLO MESSAGE

```
SQL> set serveroutput on;
SQL> declare
  2 a varchar2(20);
  3 begin
  4 a:='Hello';
  5 dbms_output.put_line(a);
  6 end;
  7 /
Hello
```

PL/SQL procedure successfully completed.

TO INPUT A VALUE FROM THE USER AND DISPLAY IT

```
SQL> set serveroutput on;
SQL> declare
  2 a varchar2(20);
  3 begin
  4 a:=&a;
  5 dbms_output.put_line(a);
  6 end;
  7 /
Enter value for a: 5
old 4: a:=&a;
new 4: a:=5;
5
```

PL/SQL procedure successfully completed.

GREATEST OF TWO NUMBERS

```
SQL> set serveroutput on;
```

```
SQL> declare
  2 a number(7);
```

```

3 b number(7);
4 begin
5 a:=&a;
6 b:=&b;
7 if(a>b) then
8 dbms_output.put_line (' The grerater of the two is'|| a);
9 else
10 dbms_output.put_line (' The grerater of the two is'|| b);
11 end if;
12 end;
13 /

```

Enter value for a: 5

old 5: a:=&a;

new 5: a:=5;

Enter value for b: 9

old 6: b:=&b;

new 6: b:=9;

The grerater of the two is9

PL/SQL procedure successfully completed.

GREATEST OF THREE NUMBERS

SQL> set serveroutput on;

SQL> declare

```

2 a number(7);
3 b number(7);
4 c number(7);
5 begin
6 a:=&a;
7 b:=&b;
8 c:=&c;
9 if(a>b and a>c) then
10 dbms_output.put_line (' The greatest of the three is ' || a);
11 else if (b>c) then
12 dbms_output.put_line (' The greatest of the three is ' || b);
13 else
14 dbms_output.put_line (' The greatest of the three is ' || c);
15 end if;
16 end if;
17 end;
18 /

```

Enter value for a: 5

old 6: a:=&a;

new 6: a:=5;

```
Enter value for b: 7
old 7: b:=&b;
new 7: b:=7;
Enter value for c: 1
old 8: c:=&c;
new 8: c:=1;
The greatest of the three is 7
```

PL/SQL procedure successfully completed.

PRINT NUMBERS FROM 1 TO 5 USING SIMPLE LOOP

```
SQL> set serveroutput on;
```

```
SQL> declare
  2 a number:=1;
  3 begin
  4 loop
  5 dbms_output.put_line (a);
  6 a:=a+1;
  7 exit when a>5;
  8 end loop;
  9 end;
 10 /
1
2
3
4
5
```

PL/SQL procedure successfully completed.

PRINT NUMBERS FROM 1 TO 4 USING WHILE LOOP

```
SQL> set serveroutput on;
```

```
SQL> declare
  2 a number:=1;
  3 begin
  4 while(a<5)
  5 loop
  6 dbms_output.put_line (a);
  7 a:=a+1;
  8 end loop;
```

```
9 end;
10 /
```

```
1
2
3
4
```

PL/SQL procedure successfully completed.

PRINT NUMBERS FROM 1 TO 5 USING FOR LOOP

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 a number:=1;
3 begin
4 for a in 1..5
5 loop
6 dbms_output.put_line (a);
7 end loop;
8 end;
9 /
```

```
1
2
3
4
5
```

PL/SQL procedure successfully completed.

PRINT NUMBERS FROM 1 TO 5 IN REVERSE ORDER USING FOR LOOP

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 a number:=1;
3 begin
4 for a in reverse 1..5
5 loop
6 dbms_output.put_line (a);
7 end loop;
8 end;
9 /
```

```
5
4
3
2
1
```

PL/SQL procedure successfully completed.

TO CALCULATE AREA OF CIRCLE

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 pi constant number(4,2):=3.14;
```

```

3 a number(20);
4 r number(20);
5 begin
6 r:=&r;
7 a:= pi* power(r,2);
8 dbms_output.put_line (' The area of circle is ' || a);
9 end;
10 /

```

Enter value for r: 2

old 6: r:=&r;

new 6: r:=2;

The area of circle is 13

PL/SQL procedure successfully completed.

TO CREATE SACCOUNT TABLE

```
SQL> create table saccount ( accno number(5), name varchar2(20), bal number(10));
```

Table created.

```
SQL> insert into saccount values ( 1,'mala',20000);
```

1 row created.

```
SQL> insert into saccount values (2,'kala',30000);
```

1 row created.

```
SQL> select * from saccount;
```

ACCNO	NAME	BAL
1	mala	20000
2	kala	30000

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 a_bal number(7);
```

```
3 a_no varchar2(20);
```

```
4 debit number(7):=2000;
```

```
5 minamt number(7):=500;
```

```
6 begin
```

```
7 a_no:=&a_no;
```

```
8 select bal into a_bal from saccount where accno= a_no;
```

```
9 a_bal:= a_bal-debit;
```

```
10 if (a_bal > minamt) then
```

```
11 update saccount set bal=bal-debit where accno=a_no;
```

```
12 end if;
```

```
13 end;
```

```
14
```

```
15 /
```

Enter value for a_no: 1

old 7: a_no:=&a_no;

new 7: a_no:=1;

PL/SQL procedure successfully completed.

SQL> select * from saccount;

ACCNO	NAME	BAL
1	mala	18000
2	kala	30000

TO CREATE TABLE SROUTES

SQL> create table sroutes (rno number(5), origin varchar2(20), destination varchar2(20), fare number

r(10), distance number(10));

Table created.

SQL> insert into sroutes values (2, 'chennai', 'dindugal', 400,230);

1 row created.

SQL> insert into sroutes values (3, 'chennai', 'madurai', 250,300);

1 row created.

SQL> insert into sroutes values (6, 'thanjavur', 'palani', 350,370);

1 row created.

SQL> select * from sroutes;

RNO	ORIGIN	DESTINATION	FARE	DISTANCE
2	chennai	dindugal	400	230
3	chennai	madurai	250	300
6	thanjavur	palani	350	370

SQL> set serveroutput on;

SQL> declare

2 route sroutes.rno % type;

3 fares sroutes.fare % type;

4 dist sroutes.distance % type;

5 begin

6 route:=&route;

7 select fare, distance into fares , dist from sroutes where rno=route;

8 if (dist < 250) then

9 update sroutes set fare=300 where rno=route;

10 else if dist between 250 and 370 then

11 update sroutes set fare=400 where rno=route;

12 else if (dist > 400) then

13 dbms_output.put_line('Sorry');

14 end if;

15 end if;

16 end if;

17 end;

18 /

Enter value for route: 3

```
old 6: route:=&route;
new 6: route:=3;
```

PL/SQL procedure successfully completed.

```
SQL> select * from sroutes;
```

RNO	ORIGIN	DESTINATION	FARE	DISTANCE
2	chennai	dindugal	400	230
3	chennai	madurai	400	300
6	thanjavur	palani	350	370

TO CREATE SCALCULATE TABLE

```
SQL> create table scalculate ( radius number(3), area number(5,2));
```

Table created.

```
SQL> desc scalculate;
```

Name	Null?	Type
RADIUS		NUMBER(3)
AREA		NUMBER(5,2)

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 pi constant number(4,2):=3.14;
3 area number(5,2);
4 radius number(3);
5 begin
6 radius:=3;
7 while (radius <=7)
8 loop
9 area:= pi* power(radius,2);
10 insert into scalculate values (radius,area);
11 radius:=radius+1;
12 end loop;
13 end;
14 /
```

PL/SQL procedure successfully completed.

```
SQL> select * from scalculate;
```

```
RADIUS    AREA
```

```
-----  
3    28.26  
4    50.24  
5    78.5  
6   113.04  
7   153.86
```

TO CALCULATE FACTORIAL OF A GIVEN NUMBER

```
SQL> set serveroutput on;
```

```
SQL> declare
```

```
2 f number(4):=1;
```

```
3 i number(4);
```

```
4 begin
```

```
5 i:=&i;
```

```
6 while(i>=1)
```

```
7 loop
```

```
8 f:=f*i;
```

```
9 i:=i-1;
```

```
10 end loop;
```

```
11 dbms_output.put_line('The value is ' || f);
```

```
12 end;
```

```
13 /
```

```
Enter value for i: 5
```

```
old 5: i:=&i;
```

```
new 5: i:=5;
```

```
The value is 120
```

PL/SQL procedure successfully completed.

PROGRAM 1

Write a PL/SQL block to calculate the incentive of an employee whose ID is 110.

```
create or replace function cal_incentive(emp_id int)
return int
is
    emp_salary int;
    incentive int;
BEGIN
    SELECT SALARY into emp_salary FROM employee11 WHERE ID = emp_id;
    incentive := emp_salary*0.1;
    return incentive;
end cal_incentive

DECLARE
    incentive_amt NUMBER;
BEGIN
    incentive_amt := cal_incentive(110);
    DBMS_OUTPUT.PUT_LINE('Incentive for employee 110: ' || incentive_amt);
END;
```

PROGRAM 2

Write a PL/SQL block to show an invalid case-insensitive reference to a quoted and without quoted user-defined identifier.

```
DECLARE
  "EmployeeName" VARCHAR2(50) := 'Alice';
  employee_id NUMBER := 110;
BEGIN
  --DBMS_OUTPUT.PUT_LINE(EmployeeName);--
  DBMS_OUTPUT.PUT_LINE(employee_id);
  DBMS_OUTPUT.PUT_LINE("EmployeeName");
END;
```

PROGRAM 3

Write a PL/SQL block to adjust the salary of the employee whose ID 122.

Sample table: employees

```
DEClare
  old_salary int;
  new_salary int;
BEGIN
  select salary into old_salary
  From employee11
  where id = 122;
  new_salary := old_salary +
old_salary*0.1;
  update employee11 set salary =
NEW_salary where id = 122;

end
```

PROGRAM 4

Write a PL/SQL block to create a procedure using the "IS [NOT] NULL Operator" and show AND operator returns TRUE if and only if both operands are TRUE.

```
CREATE OR REPLACE PROCEDURE check_null_conditions (  
    param1 IN VARCHAR2,  
    param2 IN VARCHAR2  
) IS  
BEGIN  
    -- Check if both parameters are NOT NULL  
    IF param1 IS NOT NULL AND param2 IS NOT NULL THEN  
        DBMS_OUTPUT.PUT_LINE('Both parameters are NOT NULL.');    ELSE  
        DBMS_OUTPUT.PUT_LINE('At least one parameter is NULL.');    END IF;  
END check_null_conditions;  
  
BEGIN  
    check_null_conditions('Hello', 'World');  
    check_null_conditions('Hello', NULL);  
    check_null_conditions(NULL, NULL);  
END;
```

PROGRAM 5

Write a PL/SQL block to describe the usage of LIKE operator including wildcard characters and escape character.

```
DECLARE
    emp_name VARCHAR2(50) := 'John_Doe';
    pattern1 VARCHAR2(20) := '%Doe';
    pattern2 VARCHAR2(20) := 'Jo%';
    pattern3 VARCHAR2(20) := 'John\_Doe';
BEGIN
    IF emp_name LIKE pattern1 THEN
        DBMS_OUTPUT.PUT_LINE('Employee name ends with "Doe"');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee name does not end with "Doe"');
    END IF;
    IF emp_name LIKE pattern2 THEN
        DBMS_OUTPUT.PUT_LINE('Employee name starts with "Jo"');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee name does not start with "Jo"');
    END IF;
    IF emp_name LIKE pattern3 ESCAPE '\' THEN
        DBMS_OUTPUT.PUT_LINE('Employee name matches "John_Doe" with literal underscore');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee name does not match "John_Doe" exactly');
    END IF;
END;
```

PROGRAM 6

Write a PL/SQL program to arrange the number of two variable in such a way that the small number will store in num_small variable and large number will store in num_large variable.

```
CREATE OR REPLACE PROCEDURE ordering(param1 NUMBER, param2 NUMBER)
IS
```

```
    num_small NUMBER;
    num_large NUMBER;
BEGIN
    IF param1 < param2 THEN
        num_small := param1;
        num_large := param2;
    ELSE
        num_small := param2;
        num_large := param1;
    END IF;
```

```

    DBMS_OUTPUT.PUT_LINE('Smaller number: ' || num_small);
    DBMS_OUTPUT.PUT_LINE('Larger number: ' || num_large);
END ordering;
```

```

begin
    ordering(12,45);
    ordering(23,12);
end;
```

PROGRAM 7

Write a PL/SQL procedure to calculate the incentive on a target achieved and display the message either the record updated or not.

```
create or replace procedure update_incentive(emp_id
int,target_achieved number)
is
    amt number;
    row_no number;
begin
    amt:= target_achieved *0.5;
    update employee11
    set incentive = amt
    where id = emp_id;
    row_no := SQL%ROWCOUNT;
    IF row_NO > 0 THEN
        DBMS_OUTPUT.PUT_LINE('Record updated
successfully. Incentive is: ' || amt);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Record not updated.
Employee ID may not exist.');
```

```
    END IF;
    END update_incentive;

begin
    update_incentive(110,40000);
end
```

PROGRAM 8

Write a PL/SQL procedure to calculate incentive achieved according to the specific sale limit.

```
CREATE OR REPLACE PROCEDURE incentive_saleslimit(EMP_ID INT)
IS
    amount NUMBER;
    AMT NUMBER;
BEGIN
    SELECT SALES_AMT INTO amount FROM employee11 where id = emp_id;
    IF amount >= 100000 THEN
        amt := amount * 0.10;
    ELSIF amount >= 50000 THEN
        amt := amount * 0.07;
    ELSIF amount >= 10000 THEN
        amt := amount * 0.05;
    ELSE
        amt := 0;
    END IF;
    update employee11 set incentive = amt where id = emp_id;
    dbms_output.put_line('incentive updated');
end incentive_saleslimit;

begin
    incentive_saleslimit(110);
end;
```


PROGRAM 9

Write a PL/SQL program to count number of employees in department 50 and check whether this department have any vacancies or not. There are 45 vacancies in this department.

```
declare
    count_dept50 number;
begin
    select count(*) into count_dept50 from employee11 where dept = 50;

    if count_dept50<45 then dbms_output.put_line('there are'|| (45 - count_dept50)|| ' vacancies in dept
50. ');
    else dbms_output.put_line('There is no vacancies. ');
    end if;
end
```

PROGRAM 10

Write a PL/SQL program to count number of employees in a specific department and check whether this department have any vacancies or not. If any vacancies, how many vacancies are in that department.

```
CREATE OR REPLACE PROCEDURE COUNT_DEPT(DEPARTMENT NUMBER,
COUNT_EMPLOYEE NUMBER)
IS
    count_dept50 number;
begin
    select count(*) into count_dept50 from employee11 where dept =DEPARTMENT;

    if count_dept50<45 then dbms_output.put_line('there are '|| (COUNT_EMPLOYEE - count_DEPT50)||
vacancies in dept 50. ');
    else dbms_output.put_line('There is no vacancies. ');
    end if;
end COUNT_DEPT;

BEGIN
    COUNT_DEPT(80 ,2);
END;
```

PROGRAM 11

Write a PL/SQL program to display the employee IDs, names, job titles, hire dates, and salaries of all employees.

```
DECLARE
CURSOR emp_cursor IS
    SELECT emp_id, Fname,LNAME, job_title, hire_date, salary
    FROM EMPLOYEE10;

emp_record emp_cursor%ROWTYPE;
BEGIN

FOR emp_record IN emp_cursor LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.emp_id ||
        ', Name: ' || emp_record.Fname || ' ' || EMP_RECORD.LNAME||
        ', Job Title: ' || emp_record.job_title ||
        ', Hire Date: ' || TO_CHAR(emp_record.hire_date, 'DD-MON-YYYY') ||
        ', Salary: ' || emp_record.salary);
END LOOP;
END;
```

PROGRAM 12

Write a PL/SQL program to display the employee IDs, names, and department names of all employees.

```
DECLARE
  EMP_ID NUMBER;
  EMP_NAME EMPLOYEE10.FNAME%TYPE;
  DEPARTMENT EMPLOYEE10.DEPT_NAME%TYPE;
  CURSOR EMPLOYEE IS
    SELECT EMP_ID,FNAME,DEPT_NAME FROM EMPLOYEE10;
BEGIN
  OPEN EMPLOYEE;
  LOOP
    FETCH EMPLOYEE INTO EMP_ID,EMP_NAME,DEPARTMENT;
    EXIT WHEN EMPLOYEE%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(EMP_ID||' '||EMP_NAME||
'||DEPARTMENT);
  END LOOP;
  CLOSE EMPLOYEE;
END
```

PROGRAM 13

Write a PL/SQL program to display the job IDs, titles, and minimum salaries of all jobs.

```
DECLARE
JOB_ID NUMBER;
JOB_TITLE EMPLOYEE10.JOB_TITLE%TYPE;
MIN_SALARY EMPLOYEE10.SALARY%TYPE;
CURSOR JOB IS
SELECT JOB_ID , JOB_TITLE,MIN(SALARY) FROM EMPLOYEE10 GROUP BY
JOB_ID,JOB_TITLE;
BEGIN
OPEN JOB;
LOOP
FETCH JOB INTO JOB_ID,JOB_TITLE,MIN_SALARY;
EXIT WHEN JOB%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(JOB_ID||'- '||JOB_TITLE||'- '||MIN_SALARY);
END LOOP;
CLOSE JOB;
END
```

PROGRAM 14

Write a PL/SQL program to display the employee IDs, names, and job history start dates of all employees.

```
DECLARE
CURSOR emp_job_history_cur IS
SELECT e.ID AS employee_id, e.NAME AS employee_name,
j.start_date
FROM employee11 e
JOIN job_history11 j ON e.ID = j.employee_id
ORDER BY e.ID, j.start_date;
BEGIN
FOR emp_record IN emp_job_history_cur LOOP
DBMS_OUTPUT.PUT_LINE('Employee ID: ' ||
emp_record.employee_id ||
', Name: ' || emp_record.employee_name ||
', Job Start Date: ' || TO_CHAR(emp_record.start_date,
'DD-MM-YYYY'));
END LOOP;
END;
```

PROGRAM 15

Write a PL/SQL program to display the employee IDs, names, and job history end dates of all employees.

```
DECLARE
CURSOR emp_job_history_cur IS
  SELECT e.ID AS employee_id, e.NAME AS employee_name, j.end_date
  FROM employee11 e
  JOIN job_history11 j ON e.ID = j.employee_id
  ORDER BY e.ID, j.end_date;
BEGIN
  FOR emp_record IN emp_job_history_cur LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.employee_id ||
                          ', Name: ' || emp_record.employee_name ||
                          ', Job End Date: ' || TO_CHAR(emp_record.end_date, 'DD-MM-
YYYYY'));
  END LOOP;
END;
```

Ex.No.: 12		WORKING WITH CURSOR, PROCEDURES AND FUNCTIONS
Date:	24.10.24	

AIM:

Create PL/SQL Blocks to perform the Item Transaction Operations using CURSOR, FUNCTION and PROCEDURE.

ALGORITHM:

STEP-1: Start.

STEP-2: Create two tables Item Master and Item Trans.

itemmaster(itemid , itemname, stockonhand)

itemtrans(itemid ,itemname ,dateofpurchase ,quantity)

STEP-3: Create a PROCEDURE with id, name and quantity as parameters which make a call to the FUNCTION by passing id, name, dop, and quantity as parameters dop is set as sysdate.

STEP-4: Using FUNCTION fetch each record from the table Item Master using CURSOR inside a Loop statement,
If Item Master's ItemId is equal to the entered ID value then exit the loop otherwise fetch the next record.

loop

fetch master into masterrec

exit when master%notfound

if masterrec.itemid=id then

exit;

end if;

end loop;

STEP-5: If Itemmaster's itemid = id then,

Add the Itemmaster's stockonhand with the given quantity and update the ItemMaster table and insert the Item information into the ItemTrans table.

STEP-6: Else, if the inputted item is not present in the ItemMaster table then insert t new Item in both the tables.

STEP-7: Call the Procedure by passing the Item informations which calls the Function.

STEP-8: Exit.

PROCEDURES – SYNTAX

```
create or replace procedure <procedure name> (argument {in, out, inout} datatype ) {is,as}
variable declaration;
constant declaration;
begin
PL/SQL subprogram body;
exception
exception PL/SQL block;
end;
```

FUNCTIONS – SYNTAX

```
create or replace function <function name> (argument in datatype,.....) return datatype {is,as}
variable declaration;
constant declaration;
begin
PL/SQL subprogram body;
exception
exception PL/SQL block;
end;
```

CREATING THE TABLE 'ITITEMS' AND DISPLAYING THE CONTENTS

```
SQL> create table ititems(itemid number(3), actualprice number(5), ordid number(4), prodid
number(4));
Table created.
```

```
SQL> insert into ititems values(101, 2000, 500, 201);
1 row created.
```


SQL> insert into ititems values(102, 3000, 1600, 202);
1 row created.

SQL> insert into ititems values(103, 4000, 600, 202);
1 row created.

SQL> select * from ititems;

ITEMID	ACTUALPRICE	ORDID	PRODID
101	2000	500	201
102	3000	1600	202
103	4000	600	202

PROGRAM FOR GENERAL PROCEDURE – SELECTED RECORD’S PRICE IS INCREMENTED BY 500 , EXECUTING THE PROCEDURE CREATED AND DISPLAYING THE UPDATED TABLE

SQL> create procedure itsum(identity number, total number) is price number;
2 null_price exception;
3 begin
4 select actualprice into price from ititems where itemid=identity;
5 if price is null then
6 raise null_price;
7 else
8 update ititems set actualprice=actualprice+total where itemid=identity;
9 end if;
10 exception
11 when null_price then
12 dbms_output.put_line('price is null');
13 end;
14 /
Procedure created.

SQL> exec itsum(101, 500);
PL/SQL procedure successfully completed.

SQL> select * from ititems;

ITEMID	ACTUALPRICE	ORDID	PRODID
101	2500	500	201
102	3000	1600	202
103	4000	600	202

PROCEDURE FOR ‘IN’ PARAMETER – CREATION, EXECUTION

SQL> set serveroutput on;

```
SQL> create procedure yyy (a IN number) is price number;
  2 begin
  3 select actualprice into price from ititems where itemid=a;
  4 dbms_output.put_line('Actual price is ' || price);
  5 if price is null then
  6 dbms_output.put_line('price is null');
  7 end if;
  8 end;
  9 /
```

Procedure created.

```
SQL> exec yyy(103);
Actual price is 4000
PL/SQL procedure successfully completed.
```

PROCEDURE FOR 'OUT' PARAMETER – CREATION, EXECUTION

```
SQL> set serveroutput on;
```

```
SQL> create procedure zzz (a in number, b out number) is identity number;
  2 begin
  3 select ordid into identity from ititems where itemid=a;
  4 if identity<1000 then
  5 b:=100;
  6 end if;
  7 end;
  8 /
```

Procedure created.

```
SQL> declare
  2 a number;
  3 b number;
  4 begin
  5 zzz(101,b);
  6 dbms_output.put_line('The value of b is '|| b);
  7 end;
  8 /
```

The value of b is 100

PL/SQL procedure successfully completed.

PROCEDURE FOR 'INOUT' PARAMETER – CREATION, EXECUTION

```
SQL> create procedure itit ( a in out number) is
  2 begin
  3 a:=a+1;
```

```
4 end;
5 /
Procedure created.
```

```
SQL> declare
2 a number:=7;
3 begin
4 itit(a);
5 dbms_output.put_line('The updated value is '||a);
6 end;
7 /
```

The updated value is 8

PL/SQL procedure successfully completed.

CREATE THE TABLE 'ITTRAIN' TO BE USED FOR FUNCTIONS

```
SQL>create table ittrain ( tno number(10), tfare number(10));
Table created.
```

```
SQL>insert into ittrain values (1001, 550);
1 row created.
```

```
SQL>insert into ittrain values (1002, 600);
1 row created.
```

```
SQL>select * from ittrain;
```

TNO	TFARE
1001	550
1002	600

PROGRAM FOR FUNCTION AND IT'S EXECUTION

```
SQL> create function aaa (trainnumber number) return number is
2 trainfunction ittrain.tfare % type;
3 begin
4 select tfare into trainfunction from ittrain where tno=trainnumber;
5 return(trainfunction);
6 end;
7 /
```

Function created.

```
SQL> set serveroutput on;
```

```

SQL> declare
2 total number;
3 begin
4 total:=aaa (1001);
5 dbms_output.put_line('Train fare is Rs. '||total);
6 end;
7 /

```

Train fare is Rs.550

PL/SQL procedure successfully completed.

FACTORIAL OF A NUMBER USING FUNCTION — PROGRAM AND EXECUTION

```

SQL> create function itfact (a number) return number is
2 fact number:=1;
3 b number;
4 begin
5 b:=a;
6 while b>0
7 loop
8 fact:=fact*b;
9 b:=b-1;
10 end loop;
11 return(fact);
12 end;
13 /

```

Function created.

```
SQL> set serveroutput on;
```

```

SQL> declare
2 a number:=7;
3 f number(10);
4 begin
5 f:=itfact(a);
6 dbms_output.put_line('The factorial of the given number is'||f);
7 end;
8 /

```

The factorial of the given number is 5040

PL/SQL procedure successfully completed.

Program 1

FACTORIAL OF A NUMBER USING FUNCTION

```
create or replace function factorial(a number)
return number
is
  fact number:=1;
  b number ;
begin
  b := a;
  while b>0
  loop
    fact := fact*b;
    b := b-1;
  end loop;
  return fact;
end;
declare
  factorial_5 number;
begin
  factorial_5 := factorial(5);
  dbms_output.put_line('result :'||factorial_5);
end;
```

Program 2

Write a PL/SQL program using Procedures IN,INOUT,OUT parameters to retrieve the corresponding book information in library

```
CREATE OR REPLACE PROCEDURE GetInfo(
input_book_id IN INT,
output_title OUT VARCHAR2,
output_author OUT VARCHAR2,
output_publication_year OUT INT,
output_available_copies OUT INT
) AS
BEGIN
-- Select book information based on the input book ID
SELECT
    title,
    author,
    publication_year,
    available_copies
INTO
    output_title,
    output_author,
    output_publication_year,
    output_available_copies
FROM
    Books
WHERE
    id = input_book_id;
END GetInfo;

DECLARE
    book_title VARCHAR2(255);
    book_author VARCHAR2(255);
    book_pub_year INT;
    book_available_copies INT;
BEGIN
-- Call the procedure
GetInfo(1, book_title, book_author, book_pub_year, book_available_copies);

-- Display the results
DBMS_OUTPUT.PUT_LINE('Title: ' || book_title);
DBMS_OUTPUT.PUT_LINE('Author: ' || book_author);
DBMS_OUTPUT.PUT_LINE('Publication Year: ' || book_pub_year);
DBMS_OUTPUT.PUT_LINE('Available Copies: ' || book_available_copies);
END;
```

TO WRITE A PL/SQL BLOCK TO DISPLAY THE EMPLOYEE ID AND EMPLOYEE NAME WHERE DEPARTMENT NUMBER IS 11 USING EXPLICIT CURSORS

```
1 declare
2 cursor cen1 is select eid,sal from ssempp where dno=11;
3 ecode ssempp.eid%type;
4 esal empp.sal%type;
5 begin
6 open cen1;
7 loop
8 fetch cen1 into ecode,esal;
9 exit when cen1%notfound;
10 dbms_output.put_line(' Employee code and employee salary are' || ecode _and' || esal);
```

```

11 end loop;
12 close cenl;
13* end;

```

SQL> /

Employee code and employee salary are 1 and 39000

Employee code and employee salary are 5 and 35000

Employee code and employee salary are 6 and 23000

PL/SQL procedure successfully completed.

TO WRITE A PL/SQL BLOCK TO UPDATE THE SALARY BY 5000 WHERE THE JOB IS LECTURER , TO CHECK IF UPDATES ARE MADE USING IMPLICIT CURSORS AND TO DISPLAY THE UPDATED TABLE

SQL> declare

```

2 county number;
3 begin
4 update ssempp set sal=sal+10000 where job='lecturer';
5 county:= sql%rowcount;
6 if county > 0 then
7 dbms_output.put_line('The number of rows are '|| county);
8 end if;
9 if sql %found then
10 dbms_output.put_line('Employee record modification successful');
11 else if sql%notfound then
12 dbms_output.put_line('Employee record is not found');
13 end if;
14 end if;
15 end;
16 /

```

The number of rows are 3

Employee record modification successful

PL/SQL procedure successfully completed.

SQL> select * from ssempp;

EID	ENAME	JOB	SAL	DNO
1	nala	lecturer	44000	11
2	kala	seniorlecturer	20000	12
5	ajay	lecturer	40000	11
6	vijay	lecturer	28000	11
3	nila	professor	60000	12

Evaluation Procedure	Marks awarded
PL/SQL Procedure(5)	
Program/Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 13		WORKING WITH TRIGGER <u>TRIGGER</u>
Date:	26.10.24	

DEFINITION

A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database. The parts of a trigger are,

- **Trigger statement:** Specifies the DML statements and fires the trigger body. It also specifies the table to which the trigger is associated.
- **Trigger body or trigger action:** It is a PL/SQL block that is executed when the triggering statement is used.
- **Trigger restriction:** Restrictions on the trigger can be achieved

The different uses of triggers are as follows,

- *To generate data automatically*
- *To enforce complex integrity constraints*
- *To customize complex securing authorizations*
- *To maintain the replicate table*
- *To audit data modifications*

TYPES OF TRIGGERS

The various types of triggers are as follows,

- **Before:** It fires the trigger before executing the trigger statement.
- **After:** It fires the trigger after executing the trigger statement
- .
- **For each row:** It specifies that the trigger fires once per row
- .
- **For each statement:** This is the default trigger that is invoked. It specifies that the trigger fires once per statement.

VARIABLES USED IN TRIGGERS

- :new
- :old

These two variables retain the new and old values of the column updated in the database. The values in these variables can be used in the database triggers for data manipulation

SYNTAX

```
create or replace trigger triggername [before/after] {DML statements}
on [tablename] [for each row/statement]
begin
-----
-----
-----
exception
end;
```

USER DEFINED ERROR MESSAGE

The package —raise_application_error is used to issue the user defined error messages

Syntax: raise_application_error(error number, _error message_);

The error number can lie between -20000 and -20999.

The error message should be a character string.

TABLE CREATION:

```
create table employeebonus(empno number(5)constraint emppk primary key, empname
varchar2(25)not null, experience number(2)not null, bonus number(7,2));
```

Table created.

TRIGGER CREATION FOR BONUS CALCULATION:

```
SQL> set serveroutput on
```

```
SQL> create or replace trigger employeebonus_tgr
```

```
after insert on employeebonus
```

```
declare
```

```
cursor emp is select * from employeebonus;
```

```
emprec employeebonus%rowtype;
```

```
begin
```

```

open emp;
loop
fetch emp into emprec;
exit when emp%notfound;
if(emprec.experience<5) then
emprec.bonus:=5000;
elsif(emprec.experience>=5 and emprec.experience<8) then
emprec.bonus:=8000;
else
emprec.bonus:=10000;
end if;
update employeebonus set bonus=emprec.bonus where empno=emprec.empno;
end loop;
close emp;

dbms_output.put_line('Bonus calculated and Updated Sucessfully');
end;

/

```

Trigger created.

TABLE DESCRIPTION:

SQL> desc employeebonus;

Name Null? Type

```

-----
EMPNO NOT NULL NUMBER(5)
EMPNAME NOT NULL VARCHAR2(25)
EXPERIENCE NOT NULL NUMBER(2)
BONUS NUMBER(7,2)

```

RECORD INSERTION:

```

SQL> insert into employeebonus(empno,empname,experience)
values(&empno,&empname,&experience);

```

Enter value for empno: 101

Enter value for empname: murugan

Enter value for experience: 25

old 1: insert into employeebonus(empno,empname,experience)

```
values(&empno,&empname,&experience)
```

```
new 1: insert into employeebonus(empno,empname,experience)
```

```
values(101,'murugan',25)
```

Bonus calculated and Updated Sucessfully

1 row created.

RECORD SELECTION:

```
SQL> select * from employeebonus;
```

```
EMPNO EMPNAME EXPERIENCE BONUS
```

```
-----
```

```
101 murugan 25 10000
```

```
102 suresh 3 5000
```

```
103 akash 7 8000
```

```
104 mahesh 2 5000
```

RESULT:

Thus, the above program was Created and Executed Successfully.

Program 1

Write a code in PL/SQL to develop a trigger that enforces referential integrity by preventing the deletion of a parent record if child records exist.

```
CREATE OR REPLACE TRIGGER on_delete
BEFORE DELETE ON employee13
FOR EACH ROW
DECLARE
    row_count NUMBER;
BEGIN

    SELECT COUNT(*)
    INTO row_count
    FROM details
    WHERE details.emp_id = :OLD.emp_id;
    IF row_count > 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Cannot delete parent record as child records exist.');
```

Program 2

Write a code in PL/SQL to create a trigger that checks for duplicate values in a specific column and raises an exception if found.

```
create or replace trigger check_duplicate
before insert OR UPDATE ON employee13
FOR EACH ROW
DECLARE
    duplicate_count NUMBER;
BEGIN
    select count(*) into duplicate_count
    from employee13
    where emp_name =: new.emp_name
    AND ( :OLD.emp_id IS NULL OR emp_id != :OLD.emp_id );
    if duplicate_count > 0 THEN RAISE_APPLICATION_ERROR(-
20002, 'Duplicate name found. Each name must be unique.');
```

Program 3

Write a code in PL/SQL to create a trigger that restricts the insertion of new rows if the total of a column's values exceeds a certain threshold.

```
create or replace trigger set_max
before insert or update on employee13
for each row
declare
    threshold constant number := 1000000;
    curr_total number;
begin
    select sum(salary) into curr_total from employee13;
    if curr_total+ :new.salary > threshold then
        RAISE_APPLICATION_ERROR(-20003, 'Cannot insert row: Total salary amount exceeds
the allowed threshold.');
```

Program 4

Write a code in PL/SQL to design a trigger that captures changes made to specific columns and logs them in an audit table.

```
CREATE OR REPLACE TRIGGER audit_employee_changes
AFTER UPDATE ON employee13
FOR EACH ROW
BEGIN

    IF :OLD.salary != :NEW.salary THEN
        INSERT INTO employee_audit (emp_id, changed_column, old_value,
new_value, changed_by)
        VALUES (:OLD.emp_id, 'salary', TO_CHAR(:OLD.salary),
TO_CHAR(:NEW.salary), USER);
    END IF;
    IF :OLD.emp_name != :NEW.emp_name THEN
        INSERT INTO employee_audit (emp_id, changed_column, old_value,
new_value, changed_by)
        VALUES (:OLD.emp_id, 'name', :OLD.emp_name, :NEW.emp_name,
USER);
    END IF;
END;
```

Program 5

Write a code in PL/SQL to implement a trigger that records user activity (inserts, updates, deletes) in an audit log for a given set of tables.

```
CREATE OR REPLACE TRIGGER audit_employees
AFTER INSERT OR UPDATE OR DELETE ON employee13
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        insert into audit_log (operation_type,table_name ,old_value ,new_value ,changed_by)
values('INSERT', 'employees', NULL, :NEW.emp_id || ', ' || :NEW.emp_name || ', ' || :NEW.salary,
USER);
    ELSIF UPDATING THEN
        insert into audit_log (operation_type,table_name ,old_value ,new_value ,changed_by)
values('UPDATE', 'employees',
        :OLD.emp_id || ', ' || :OLD.emp_name || ', ' || :OLD.salary,
        :NEW.emp_id || ', ' || :NEW.emp_name || ', ' || :NEW.salary, USER);
    ELSIF DELETING THEN
        insert into audit_log (operation_type,table_name ,old_value ,new_value ,changed_by)
values('DELETE', 'employees', :OLD.emp_id || ', ' || :OLD.emp_name || ', ' || :OLD.salary, NULL,
USER);
    END IF;
END;
```

Program 7

Write a code in PL/SQL to implement a trigger that automatically calculates and updates a running total column for a table whenever new rows are inserted.


```
CREATE OR REPLACE TRIGGER update_running_total  
FOR INSERT ON employee13  
COMPOUND TRIGGER
```

```
    total_running_total NUMBER := 0;
```

```
    BEFORE EACH ROW IS  
    BEGIN
```

```
        INTO total_running_total  
        FROM sales;  
        total_running_total := total_running_total + :NEW.sales_amt;  
    END BEFORE EACH ROW;
```

```
    AFTER EACH ROW IS  
    BEGIN
```

```
        INSERT INTO sales (sale_amount, running_total)  
        VALUES (:NEW.sales_amt, total_running_total);  
    END AFTER EACH ROW;
```

```
END update_running_total;
```

Program 8

Write a code in PL/SQL to create a trigger that validates the availability of items before allowing an order to be placed, considering stock levels and pending orders.

```
CREATE TRIGGER validate_stock_before_order
BEFORE INSERT ON Orders
FOR EACH ROW

DECLARE
    total_reserved INT;
    available_stock INT;
BEGIN
    SELECT SUM(quantity) INTO total_reserved
    FROM Pending_Orders
    WHERE item_id = :NEW.item_id;

    SELECT stock_quantity - total_reserved INTO available_stock
    FROM Items
    WHERE item_id = :NEW.item_id;

    IF available_stock < :NEW.quantity THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient stock available for this order.');
```

```
    END IF;
END ;
```

Evaluation Procedure	Marks awarded
PL/SQL Procedure(5)	
Program/Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	

Ex.No.: 14		MONGO DB
Date:	7.11.24	

MongoDB is a free and open-source cross-platform document-oriented database. Classified as a NoSQL database, MongoDB avoids the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas, making the integration of data in certain types of applications easier and faster.

Create Database using mongosh

After connecting to your database using mongosh, you can see which database you are using by typing db in your terminal.

If you have used the connection string provided from the MongoDB Atlas dashboard, you should be connected to the myFirstDatabase database.

Show all databases

To see all available databases, in your terminal type show dbs.

Notice that myFirstDatabase is not listed. This is because the database is empty. An empty database is essentially non-existent.

Change or Create a Database

You can change or create a new database by typing use then the name of the database.

Create Collection using mongosh

You can create a collection using the createCollection() database method.

Insert Documents

insertOne()

db.posts.insertOne({

title: "Post Title 1",

body: "Body of post.",

category: "News",

likes: 1,

tags: ["news", "events"],

```
date: Date()

})
```

EXERCISE 18

Structure of 'restaurants' collection:

```
{
  "address": {
    "building": "1007",
    "coord": [ -73.856077, 40.848447 ],
    "street": "Morris Park Ave",
    "zipcode": "10462"
  },
  "borough": "Bronx",
  "cuisine": "Bakery",

  "grades": [
    { "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },
    { "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },
    { "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },
    { "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },
    { "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }
  ],
  "name": "Morris Park Bake Shop",
  "restaurant_id": "30075445"
}
```

1. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which prepared dish except 'American' and 'Chinees' or restaurant's name begins with letter 'Wil'.

```
db.restaurants.find(
{
  $or: [
    { cuisine: { $nin: ["American", "Chinese"] } },
    { name: /^Wil/ }
  ]
},
{ restaurant_id: 1, name: 1, borough: 1, cuisine: 1 }
)
```

2. Write a MongoDB query to find the restaurant Id, name, and grades for those restaurants which achieved a grade of "A" and scored 11 on an ISODate "2014-08-11T00:00:00Z" among many of survey dates..

```
db.restaurants.find(  
  {  
    grades: {  
      $elemMatch: { grade: "A", score: 11,  
        date: ISODate("2014-08-11T00:00:00Z")  
      }  
    }  
  },  
  { restaurant_id: 1, name: 1, grades: 1 }  
)
```

3. Write a MongoDB query to find the restaurant Id, name and grades for those restaurants where the 2nd element of grades array contains a grade of "A" and score 9 on an ISODate "2014-08- 11T00:00:00Z".

```
db.restaurants.find(  
{  
  "grades.1.grade": "A",  
  "grades.1.score": 9,  
  "grades.1.date": ISODate("2014-08-11T00:00:00Z")  
},  
{ restaurant_id: 1, name: 1, grades: 1 }  
)
```

4. Write a MongoDB query to find the restaurant Id, name, address and geographical location for those restaurants where 2nd element of coord array contains a value which is more than 42 and upto 52..

```
db.restaurants.find(  
{ "address.coord.1": { $gt: 42, $lte: 52 } },  
{ restaurant_id: 1, name: 1, address: 1, "address.coord": 1 }  
)
```

5. Write a MongoDB query to arrange the name of the restaurants in ascending order along with all the columns.

```
db.restaurants.find().sort({ name: 1 })
```

6. Write a MongoDB query to arrange the name of the restaurants in descending along with all the columns.

```
db.restaurants.find().sort({ name: -1 })
```

7. Write a MongoDB query to arranged the name of the cuisine in ascending order and for that same cuisine borough should be in descending order.

```
db.restaurants.find().sort({ cuisine: 1, borough: -1 })
```

8. Write a MongoDB query to know whether all the addresses contains the street or not.

```
db.restaurants.find({ "address.street": { $exists: true } })
```


9. Write a MongoDB query which will select all documents in the restaurants collection where the coord field value is Double.

```
db.restaurants.find({ "address.coord": { $type: "double" } })
```

10. Write a MongoDB query which will select the restaurant Id, name and grades for those restaurants which returns 0 as a remainder after dividing the score by 7.

```
db.restaurants.find(  
  { "grades.score": { $mod: [7, 0] } },  
  { restaurant_id: 1, name: 1, grades: 1 }  
)
```

11. Write a MongoDB query to find the restaurant name, borough, longitude and attitude and cuisine for those restaurants which contains 'mon' as three letters somewhere in its name.

```
db.restaurants.find(  
  { name: /mon/i },  
  { name: 1, borough: 1, "address.coord": 1, cuisine: 1 }  
)
```

12. Write a MongoDB query to find the restaurant name, borough, longitude and latitude and cuisine for those restaurants which contain 'Mad' as first three letters of its name.

```
db.restaurants.find(  
  { name: /^Mad/ },  
  { name: 1, borough: 1, "address.coord": 1, cuisine: 1 }  
)
```

13. Write a MongoDB query to find the restaurants that have at least one grade with a

score of less than 5.

```
db.restaurants.find({ "grades.score": { $lt: 5 } })
```

14. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5 and that are located in the borough of Manhattan.

```
db.restaurants.find({ "grades.score": { $lt: 5 }, borough: "Manhattan" })
```

15. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5 and that are located in the borough of Manhattan or Brooklyn.

```
db.restaurants.find({ "grades.score": { $lt: 5 }, borough: { $in: ["Manhattan", "Brooklyn"] } })
```

16. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5 and that are located in the borough of Manhattan or Brooklyn, and their cuisine is not American.

```
db.restaurants.find(
{ "grades.score": { $lt: 5 }, borough: { $in: ["Manhattan", "Brooklyn"] }, cuisine: {
$ne: "American" } }
)
```

17. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5 and that are located in the borough of Manhattan or Brooklyn, and their cuisine is not American or Chinese.

```
db.restaurants.find(
{ "grades.score": { $lt: 5 }, borough: { $in: ["Manhattan", "Brooklyn"] }, cuisine: {
$nin: ["American", "Chinese"] } }
)
```

18. Write a MongoDB query to find the restaurants that have a grade with a score of 2 and a grade with a score of 6.

```
db.restaurants.find({ grades: {
  $all: [
    { $elemMatch: { score: 2 } },
    { $elemMatch: { score: 6 } }
  ]
}})
```

19. Write a MongoDB query to find the restaurants that have a grade with a score of 2 and a grade with a score of 6 and are located in the borough of Manhattan.

```
db.restaurants.find({ grades: {
  $all: [
    { $elemMatch: { score: 2 } },
    { $elemMatch: { score: 6 } }
  ]
},
borough: "Manhattan"
})
```

20. Write a MongoDB query to find the restaurants that have a grade with a score of 2 and a grade with a score of 6 and are located in the borough of Manhattan or Brooklyn.

```
db.restaurants.find({ grades: {
  $all: [
    { $elemMatch: { score: 2 } },
    { $elemMatch: { score: 6 } }
  ]
}})
```

```
},  
borough: { $in: ["Manhattan", "Brooklyn"] }  
})
```

21. Write a MongoDB query to find the restaurants that have a grade with a score of 2 and a grade with a score of 6 and are located in the borough of Manhattan or Brooklyn, and their cuisine is not American.

```

db.restaurants.find({ grades: {
  $all: [
    { $elemMatch: { score: 2 } },
    { $elemMatch: { score: 6 } }
  ]
},
borough: { $in: ["Manhattan", "Brooklyn"] }, cuisine: { $ne: "American" }
}))

```

22. Write a MongoDB query to find the restaurants that have a grade with a score of 2 and a grade with a score of 6 and are located in the borough of Manhattan or Brooklyn, and their cuisine is not American or Chinese.

```

db.restaurants.find({ grades: {
  $all: [
    { $elemMatch: { score: 2 } },
    { $elemMatch: { score: 6 } }
  ]
},
borough: { $in: ["Manhattan", "Brooklyn"] }, cuisine: { $nin: ["American", "Chinese"] }
}))

```

23. Write a MongoDB query to find the restaurants that have a grade with a score of 2 or a grade with a score of 6.

```

db.restaurants.find({
  grades: { $elemMatch: { score: { $in: [2, 6] } } }
})

```

Sample document of 'movies' collection

```
{
  _id: ObjectId("573a1390f29313caabcd42e8"),
  plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on
  their heels.',
  genres: [ 'Short', 'Western' ],
  runtime: 11,
  cast: [
    'A.C. Abadie',
    "Gilbert M. 'Broncho Billy' Anderson",
    'George Barnes',
    'Justus D. Barnes'
  ],
}
```

poster: 'https://m.media-amazon.com/images/M/MV5BMTU3NjE5NzYtYTYyNS00MDVmLWIwYjgtMmYwYWIXZDYyNzU2XkEyXkFqcGdeQXVyNzQzNzQxNzI@._V1_SY1000_SX677_AL_.jpg',
title: 'The Great Train Robbery',

fullplot: "Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",

languages: ['English'],
released: ISODate("1903-12-01T00:00:00.000Z"),
directors: ['Edwin S. Porter'],
rated: 'TV-G',
awards: { wins: 1, nominations: 0, text: '1 win.' },
lastupdated: '2015-08-13 00:27:59.177000000',
year: 1903,
imdb: { rating: 7.4, votes: 9847, id: 439 },
countries: ['USA'],
type: 'movie',
tomatoes: {
viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
fresh: 6,
critic: { rating: 7.6, numReviews: 6, meter: 100 },
rotten: 0,
lastUpdated: ISODate("2015-08-08T19:16:10.000Z")
}

1. Find all movies with full information from the 'movies' collection that released in the year 1893.

```
db.movies.find({ year: 1893 })
```

2. Find all movies with full information from the 'movies' collection that have a runtime greater than 120 minutes.

```
db.movies.find({ runtime: { $gt: 120 } })
```


3. Find all movies with full information from the 'movies' collection that have "Short" genre.

```
db.movies.find({ genres: "Short" })
```

4. Retrieve all movies from the 'movies' collection that were directed by "William K.L. Dickson" and include complete information for each movie.

```
db.movies.find({ directors: "William K.L. Dickson" })
```

6. Retrieve all movies from the 'movies' collection that were released in the USA and include complete information for each movie.

```
db.movies.find({ countries: "USA" })
```

7. Retrieve all movies from the 'movies' collection that have complete information and are rated as "UNRATED".

```
db.movies.find({ countries: "USA" })
```

8. Retrieve all movies from the 'movies' collection that have complete information and have received more than 1000 votes on IMDb.

```
db.movies.find({ countries: "USA" })
```

9. Retrieve all movies from the 'movies' collection that have complete information and have an IMDb rating higher than 7.

```
db.movies.find({ "imdb.rating": { $gt: 7 } })
```

10. Retrieve all movies from the 'movies' collection that have complete information and have a viewer rating higher than 4 on Tomatoes.

```
db.movies.find({ "tomatoes.viewer.rating": { $gt: 4 } })
```

11. Retrieve all movies from the 'movies' collection that have received an award.

```
db.movies.find({ "awards.wins": { $gt: 0 } })
```

12. Find all movies with title, languages, released, directors, writers, awards, year, genres, runtime, cast, countries from the 'movies' collection in MongoDB that have at least one nomination.

```
db.movies.find({
  "awards.nominations": { $gte: 1 }
},
{
  title: 1, languages: 1,
  released: 1, directors: 1,
  writers: 1, awards: 1,
  year: 1, genres: 1,
  runtime: 1, cast: 1,
  countries: 1
})
```

13. Find all movies with title, languages, released, directors, writers, awards, year, genres, runtime, cast, countries from the 'movies' collection in MongoDB with cast including "Charles Kayser".

```
db.movies.find({
  cast: "Charles Kayser"
}, {
  title: 1,
  languages: 1,
  released: 1,
  directors: 1,
  writers: 1,
  awards: 1,
  year: 1,
  genres: 1,
  runtime: 1,
  cast: 1,
  countries: 1
})
```

14. Retrieve all movies with title, languages, released, directors, writers, countries from the 'movies' collection in MongoDB that released on May 9, 1893.

```
db.movies.find({
released: new Date("1893-05-09")
}, {
title: 1,
languages: 1,
released: 1,
directors: 1,
writers: 1,
countries: 1
})
```

14. Retrieve all movies with title, languages, released, directors, writers, countries from the 'movies' collection in MongoDB that have a word "scene" in the title.

```
db.movies.find({ title: /scene/i
}, {
title: 1,
languages: 1,
released: 1,
directors: 1,
writers: 1,
countries: 1
})
```

Ex.No.: 15		OTHER DATABASE OBJECTS
Date:		

OTHER DATABASE OBJECTS

Objectives

After the completion of this exercise, the students will be able to do the following:

- Create, maintain, and use sequences
- Create and maintain indexes

Database Objects

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of some queries, you should consider creating an index. You

can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

What Is a Sequence?

A sequence:

- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

The CREATE SEQUENCE Statement Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE sequence
[INCREMENT BY n]
[START WITH n]
[{MAXVALUE n | NOMAXVALUE}]
[{MINVALUE n | NOMINVALUE}]
[{CYCLE | NOCYCLE}]
[{CACHE n | NOCACHE}];
```

In the syntax:

sequence is the name of the sequence generator

INCREMENT BY *n* specifies the interval between sequence numbers where *n* is an integer (If this clause is omitted, the sequence increments by 1.)

START WITH *n* specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)

MAXVALUE *n* specifies the maximum value the sequence can generate

NOMAXVALUE specifies a maximum value of 10^{27} for an ascending sequence and -1 for a descending sequence (This is the default option.)

MINVALUE *n* specifies the minimum sequence value

NOMINVALUE specifies a minimum value of 1 for an ascending sequence and $-(10^{26})$ for a descending sequence (This is the default option.)

CYCLE | NOCYCLE specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)

CACHE *n* | NOCACHE specifies how many values the Oracle server preallocates and keep in memory (By default, the Oracle server caches 20 values.)

Creating a Sequence

- Create a sequence named DEPT_DEPTID_SEQ to be used for the primary key of the DEPARTMENTS table.
- Do not use the CYCLE option.

EXAMPLE:

```
CREATE SEQUENCE dept_deptid_seq  
INCREMENT BY 10  
START WITH 120  
MAXVALUE 9999  
NOCACHE  
NOCYCLE;
```

Confirming Sequences

- Verify your sequence values in the USER_SEQUENCES data dictionary table.
- The LAST_NUMBER column displays the next available sequence number if NOCACHE is specified.

EXAMPLE:

```
SELECT sequence_name, min_value, max_value, increment_by, last_number
```

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

Using a Sequence

- Insert a new department named —Support in location ID 2500.
- View the current value for the DEPT_DEPTID_SEQ sequence.

EXAMPLE:

```
INSERT INTO departments(department_id, department_name, location_id)
VALUES (dept_deptid_seq.NEXTVAL, 'Support', 2500);
```

```
SELECT dept_deptid_seq.CURRVAL FROM dual;
```

The example inserts a new department in the DEPARTMENTS table. It uses the DEPT_DEPTID_SEQ sequence for generating a new department number as follows:

You can view the current value of the sequence:

```
SELECT dept_deptid_seq.CURRVAL FROM dual;
```

Removing a Sequence

- Remove a sequence from the data dictionary by using the DROP SEQUENCE statement.
- Once removed, the sequence can no longer be referenced.

EXAMPLE:

```
DROP SEQUENCE dept_deptid_seq;
```

What is an Index?

An index:

- Is a schema object
- Is used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table it indexes
- Is used and maintained automatically by the Oracle server

How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a PRIMARY KEY or UNIQUE constraint in a table definition.
- Manually: Users can create nonunique indexes on columns to speed up access to the rows.

Types of Indexes

Two types of indexes can be created. One type is a unique index: the Oracle server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name given to the constraint.

The other type of index is a nonunique index, which a user can create. For example, you can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.

Creating an Index

- Create an index on one or more columns.
- Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table.

```
CREATE INDEX index  
ON table (column [, column]...);
```

EXAMPLE:

```
CREATE INDEX emp_last_name_idx  
ON employees(last_name);
```

In the syntax:

index is the name of the index

table is the name of the table

column is the name of the column in the table to be indexed

When to Create an Index

You should create an index if:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 percent of the rows

When Not to Create an Index

It is usually not worth creating an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table
- The table is updated frequently
- The indexed columns are referenced as part of an Expression

Confirming Indexes

- The USER_INDEXES data dictionary view contains the name of the index and its uniqueness.
- The USER_IND_COLUMNS view contains the index name, the table name, and the column name.

EXAMPLE:

```
SELECT ic.index_name, ic.column_name, ic.column_position col_pos, ix.uniqueness
FROM user_indexes ix, user_ind_columns ic
WHERE ic.index_name = ix.index_name
AND ic.table_name = 'EMPLOYEES';
```

Removing an Index

- Remove an index from the data dictionary by using the DROP INDEX command.
- Remove the UPPER_LAST_NAME_IDX index from the data dictionary.
- To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege.

```
DROP INDEX upper_last_name_idx;
```

```
DROP INDEX index;
```

Find the Solution for the following:

1. Create a sequence to be used with the primary key column of the DEPT table. The sequence should start at 200 and have a maximum value of 1000. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.

```
db.movies.find({ title: /scene/i
}, {
title: 1,
languages: 1,
released: 1,
directors: 1,
writers: 1,
countries: 1
```

```
}}
```

2. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number

```
SELECT sequence_name, max_value, increment_by, last_number FROM user_sequences;
```

3. Write a script to insert two rows into the DEPT table. Name your script lab12_3.sql. Be sure to use the sequence that you created for the ID column. Add two departments named Education and Administration. Confirm your additions. Run the commands in your script.

```
INSERT INTO DEPT VALUES (DEPT_ID_SEQ.NEXTVAL, 'Education'); INSERT INTO  
DEPT VALUES (DEPT_ID_SEQ.NEXTVAL, 'Administration'); SELECT * FROM DEPT;
```

4. Create a nonunique index on the foreign key column (DEPT_ID) in the EMP table.
CREATE INDEX emp_dept_id_idx ON

```
EMP(DEPT_ID);
```

5. Display the indexes and uniqueness that exist in the data dictionary for the EMP table.

```
SELECT ic.index_name, ic.column_name, ic.column_position AS col_pos, ix.uniqueness  
FROM user_indexes ix
```

```
JOIN user_ind_columns ic ON ic.index_name = ix.index_name WHERE ic.table_name = 'EMP'
```

Evaluation Procedure	Marks awarded
PL/SQL Procedure(5)	
Program/Execution (5)	
Viva(5)	
Total (15)	

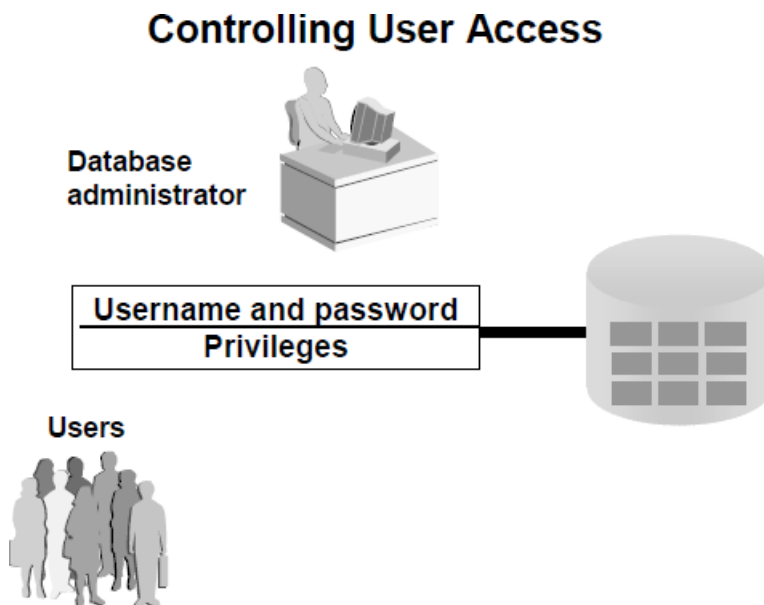
Faculty Signature	
--------------------------	--

Ex.No.: 16		CONTROLLING USER ACCESS
Date:		

Objectives

After the completion of this exercise, the students will be able to do the following:

- Create users
- Create roles to ease setup and maintenance of the security model
- Use the GRANT and REVOKE statements to grant and revoke object privileges
- Create and access database links



Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. With Oracle server database security, you can do the following:

- Control database access
- Give access to specific objects in the database
- Confirm given and received *privileges* with the Oracle data dictionary
- Create synonyms for database objects

Privileges

- Database security:
 - System security
 - Data security

- System privileges: Gaining access to the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collections of objects, such as tables, views, and sequences

System Privileges

- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
 - Creating new users
 - Removing users
 - Removing tables
 - Backing up tables

Typical DBA Privileges

System Privilege	Operations Authorized
CREATE USER	Grantee can create other Oracle users (a privilege required for a DBA role).
DROP USER	Grantee can drop another user.
DROP ANY TABLE	Grantee can drop a table in any schema.
BACKUP ANY TABLE	Grantee can back up any table in any schema with the export utility.
SELECT ANY TABLE	Grantee can query tables, views, or snapshots in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.

Creating Users

The DBA creates users by using the CREATE USER statement.

EXAMPLE:

```
CREATE USER scott IDENTIFIED BY tiger;
```

User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.
- An application developer, for example, may have the following system privileges:
 - CREATE SESSION
 - CREATE TABLE
 - CREATE SEQUENCE
 - CREATE VIEW
 - CREATE PROCEDURE

```
GRANT privilege [, privilege...]
TO user [, user/ role, PUBLIC...];
```

Typical User Privileges

System Privilege	Operations Authorized
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema
CREATE SEQUENCE	Create a sequence in the user's schema
CREATE VIEW	Create a view in the user's schema
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema

In the syntax:

privilege is the system privilege to be granted

user |role|PUBLIC is the name of the user, the name of the role, or PUBLIC designates that every user is granted the privilege

Note: Current system privileges can be found in the dictionary view SESSION_PRIVS.

Granting System Privileges

The DBA can grant a user specific system privileges.

GRANT create session, create table, create sequence, create view TO scott;

What is a Role?

A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

Syntax

CREATE ROLE *role*;

In the syntax:

role is the name of the role to be created

Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.

Creating and Granting Privileges to a Role

CREATE ROLE manager;
Role created.

GRANT create table, create view TO manager;
Grant succeeded.

GRANT manager TO DEHAAN, KOCHHAR;
Grant succeeded.

- Create a role
- Grant privileges to a role
- Grant a role to users

Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the

ALTER USER statement.
ALTER USER scott
IDENTIFIED BY lion;
User altered.

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√	√		
SELECT	√	√	√	
UPDATE	√	√		

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

```
GRANT object_priv [(columns)]
ON object
TO {user|role|PUBLIC}
[WITH GRANT OPTION];
```

In the syntax:

object_priv is an object privilege to be granted

ALL specifies all object privileges

columns specifies the column from a table or view on which privileges are granted

ON *object* is the object on which the privileges are granted

TO identifies to whom the privilege is granted

PUBLIC grants object privileges to all users

WITH GRANT OPTION allows the grantee to grant the object privileges to other users and roles

Granting Object Privileges

- Grant query privileges on the EMPLOYEES table.
- Grant privileges to update specific columns to users and roles.

```
GRANT select
ON employees
TO sue, rich;
```

```
GRANT update (department_name, location_id)
ON departments
TO scott, manager;
```

Using the WITH GRANT OPTION and PUBLIC Keywords

- Give a user authority to pass along privileges.
- Allow all users on the system to query data from Alice's DEPARTMENTS table.

```
GRANT select, insert  
ON departments  
TO scott  
WITH GRANT OPTION;
```

```
.  
GRANT select  
ON alice.departments  
TO PUBLIC;
```

How to Revoke Object Privileges

- You use the REVOKE statement to revoke privileges granted to other users.
- Privileges granted to others through the WITH GRANT OPTION clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}  
ON object  
FROM {user[, user...]|role|PUBLIC}  
[CASCADE CONSTRAINTS];
```

In the syntax:

CASCADE is required to remove any referential integrity constraints made to the CONSTRAINTS object by means of the REFERENCES privilege

Revoking Object Privileges

As user Alice, revoke the SELECT and INSERT privileges given to user Scott on the DEPARTMENTS table.

```
REVOKE select, insert  
ON departments  
FROM scott;
```

Find the Solution for the following:

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

The user should be given the CREATE SESSION privilege. This is a **system privilege**.

2. What privilege should a user be given to create tables?

The user should be given the CREATE TABLE privilege.

3. If you create a table, who can pass along privileges to other users on your table?

Only the owner of the table (the user who created the table) can pass along privileges to other users on that table.

Only the owner of the table (the user who created the table) can pass along privileges to other users on that table.

4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

You should create a **role** with the necessary privileges and then grant this role to each user.

5. What command do you use to change your password?

ALTER USER username IDENTIFIED BY new_password;

6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.

GRANT SELECT ON departments TO other_user; GRANT SELECT ON departments TO original_user;

7. Query all the rows in your DEPARTMENTS table.

SELECT * FROM departments;

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department

number 500. Team 2 should add Human Resources department number 510. Query the other team's table.

```
INSERT INTO departments VALUES (500, 'Education'); INSERT INTO  
departments VALUES (510, 'Human Resources');
```

9. Query the USER_TABLES data dictionary to see information about the tables that you own.

```
SELECT * FROM other_team_user.departments;
```

10. Revoke the SELECT privilege on your table from the other team.

```
REVOKE SELECT ON departments FROM other_team_user;
```

11. Remove the row you inserted into the DEPARTMENTS table in step 8 and save the changes.

```
DELETE FROM departments WHERE department_id IN (500,510); COMMIT
```

Evaluation Procedure	Marks awarded
PL/SQL Procedure(5)	
Program/Execution (5)	
Viva(5)	
Total (15)	
Faculty Signature	