**RAJALAKSHMI ENGINEERING COLLEGE**

**RAJALAKSHMI NAGAR, THANDALAM – 602 105**

****

**Laboratory Record NoteBook**

**CS23333 DATABASE MANAGEMENT SYSTEMS LAB**

Name : HEMASHREE R

Year / Branch : II/CSE

University Register No. : 2116240701190

College Roll No. : 240701190

Semester :  III

Academic Year : 2024-2025



****

**EXERCISE-1**

**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’.

**The CREATE TABLE Statement**

**Table:** Basic unit of storage; composed of rows and columns

**Syntax: 1** Create table table\_name (column\_name1 data\_ type (size) column\_name2 data\_ type (size)….);

**Syntax: 2** Create table table\_name (column\_name1 data\_ type (size) constraints, column\_name2 data\_ type constraints …);

**Example:**

Create table employlees ( employee\_id number(6), first\_name varchar2(20), ..job\_id varchar2(10), CONSTRAINT emp\_emp\_id\_pk PRIMARY KEY (employlee\_id));

**Tables Used in this course**

**Creating a table by using a Sub query**

**SYNTAX**

// CREATE TABLE table\_name(column\_name type(size)…);

Create table table\_name **as** select column\_name1,column\_name2,……colmn\_namen from

table\_name where predicate;

**AS Subquery**

Subquery is the select statement that defines the set of rows to be inserted into the new table.

**Example**

Create table dept80 as select employee\_id, last\_name, salary\*12 Annsal, hire\_date

from employees where dept\_id=80;

**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.

**SYNTAX**

ALTER TABLE table\_name ADD /MODIFY(Column\_name type(size));

ALTER TABLE table\_name DROP COLUMN (Column\_nname);

*ALTER TABLE ADD CONSTRAINT Constraint\_name PRIMARY KEY (Colum\_Name);*

**Example:**

Alter table dept80 add (jod\_id varchar2(9));

Alter table dept80 modify (last\_name varchar2(30));

Alter table dept80 drop column job\_id;

**NOTE:** Once the column is dropped it cannot be recovered.

**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.

**Syntax:**

**Drop table *tablename;***

**Example:**

Drop table dept80;

**RENAMING A TABLE**

To rename a table or view.

**Syntax**

RENAME old\_name to new\_name

**Example:**

Rename dept to detail\_dept;

**TRUNCATING A TABLE**

Removes all rows from the table.

Releases the storage space used by that table.

**Syntax**

TRUNCATE TABLE *table\_name*;

E**xample:**

TRUNCATE TABLE copy\_emp;

**Find the Solution for the following:**

**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) |

**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.

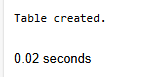
|  |  |  |
| --- | --- | --- |
| **Column name** | ID | NAME |
| **Key Type** |  |  |
| **Nulls/Unique** |  |  |
| **FK table** |  |  |
| **FK column** |  |  |
| **Data Type** | Number | Varchar2 |
| **Length** | 7 | 25 |

CREATE TABLE Department(

ID Number(7),

NAME Varchar(25)

);



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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

CREATE TABLE EMP1(

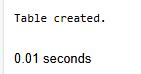
ID Number(7),

LAST\_NAME Varchar2(25),

FIRST\_NAME Varchar2(25),

DEPT\_ID Number(7)

);



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

ALTER table EMP1

MODIFY LAST\_NAME Varchar(50);



1. 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(

Id Number(6),

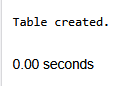
First\_name Varchar(20),

Last\_name Varchar(25),

salary Number(8,2),

Dept\_id Number(4)

);



1. Drop the EMP table.

DROP TABLE EMP1;

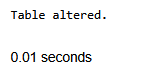


1. Rename the EMPLOYEES2 table as EMP.

E

ALTER TABLE EMPLOYEES2

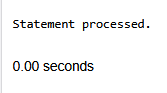
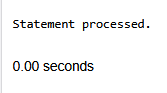
RENAME TO EMP1;



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

COMMENT ON TABLE Department IS 'This is a Comment';

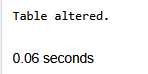
COMMENT ON TABLE EMP1 IS 'This is a Comment';



1. Drop the First\_name column from the EMP table and confirm it.

ALTER TABLE EMP1

DROP COLUMN First\_NAME;



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

**EXERCISE-2**

**MANIPULATING DATA**

**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.

**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:**

INSER INTO sales\_reps(id, name, salary, commission\_pct)

SELECt employee\_id, Last\_name, salary, commission\_pct

FROM employees

WHERE jod\_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’’;

**Find the Solution for the following:**

1. Create MY\_EMPLOYEE table with the following structure

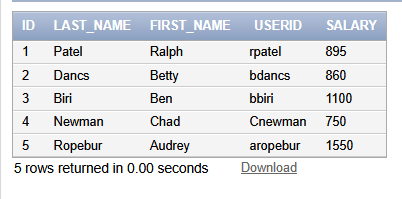
|  |  |  |
| --- | --- | --- |
| NAME | NULL? | TYPE |
| ID | Not null | Number(4) |
| Last\_name |  | Varchar(25) |
| First\_name |  | Varchar(25) |
| Userid |  | Varchar(25) |
| Salary |  | Number(9,2) |

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 |

1. Display the table with values.

Select \* from MY\_EMPLOYEE



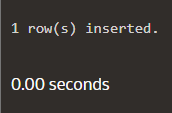
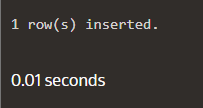
1. Populate the next two rows of data from the sample data. Concatenate the first letter of the first\_name with the first seven characters of the last\_name to produce Userid.

INSERT INTO MY\_EMPLOYEE

VALUES

(6,'Jason','Chadwick','cjason', 350),

(7,'Dan','Kishore','kdan',950) ;



1. Make the data additions permanent.

commit;

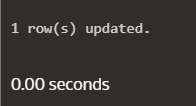


1. Change the last name of employee 3 to Drexler.

update MY\_EMPLOYEE

SET last\_name='Drexler'

Where ID=3;

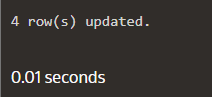


1. Change the salary to 1000 for all the employees with a salary less than 900.

update MY\_EMPLOYEE

SET salary=1000

Where salary<900;

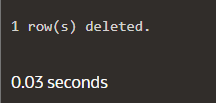


1. Delete Betty dancs from MY \_EMPLOYEE table.

DELETE from MY\_EMPLOYEE

where first\_name='Betty'

AND last\_name='Dancs';



1. Empty the fourth row of the emp table.

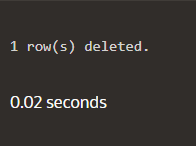
DELETE FROM MY\_EMPLOYEE

where id = ( SELECT ID FROM MY\_EMPLOYEE

ORDER BY ID

offset 3 rows

fetch next 1 rows only)



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

**EXERCISE-3**

**INCLUDING CONSTRAINTS**

**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**

1. **Domain Integrity**

* NOT NULL
* CHECK

1. **Entity Integrity**

* UNIQUE
* PRIMARY KEY

1. **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 employlees ( employee\_id number(6), first\_name varchar2(20), ..job\_id varchar2 (10), CONSTRAINT emp\_emp\_id\_pk PRIMARY KEY (employlee\_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’ COSTRAINT 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 deparments(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, CONTRAINT 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

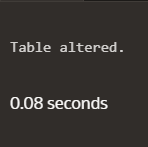
WHERE table\_name=’employees’;

**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 MY\_EMPLOYEE

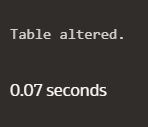
ADD constraint my\_emp\_id\_pk PRIMARY KEY (ID)



1. 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 DEPT

Add CONSTRAINT my\_dept\_id\_pk PRIMARY KEY (ID)



1. 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.

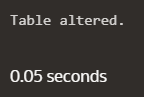
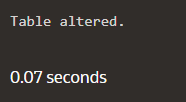
Alter table MY\_EMPLOYEE

ADD DEPT\_ID Number(22);

Alter table MY\_EMPLOYEE

ADD CONSTRAINT my\_emp\_dept\_id\_fk

FOREIGN KEY (DEPT\_ID) references DEPT(ID);



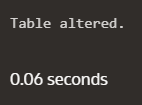
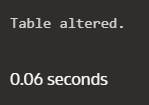
1. Modify the EMP table. Add a COMMISSION column of NUMBER data type, precision 2, scale 2. Add a constraint to the commission column that ensures that a commission value is greater than zero.

ALTER TABLE MY\_EMPLOYEE

ADD COMMISSION Number(2,2);

ALTER TABLE MY\_EMPLOYEE

ADD CHECK (COMMISSION>0);



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

**EXERCISE-4**

**Writing Basic SQL SELECT Statements**

**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—Suppr

ess 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 sued 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 deparment\_id FROM employees;

**Displaying Table Structure**

* Using DESC keyword.

**Syntax**

DESC table\_name;

**Example:**

DESC employees;

**Find the Solution for the following:**

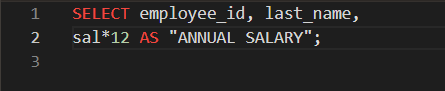
**True OR False**

1. The following statement executes successfully.

**Identify the Errors**

SELECT employee\_id, last\_name

sal\*12 ANNUAL SALARY



FROM employees;

**Queries**

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

CREATE TABLE DEPARTMENTS(

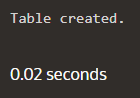
Dept\_id Number(6),

Dept\_name Varchar(20),

Manager\_id Number(6),

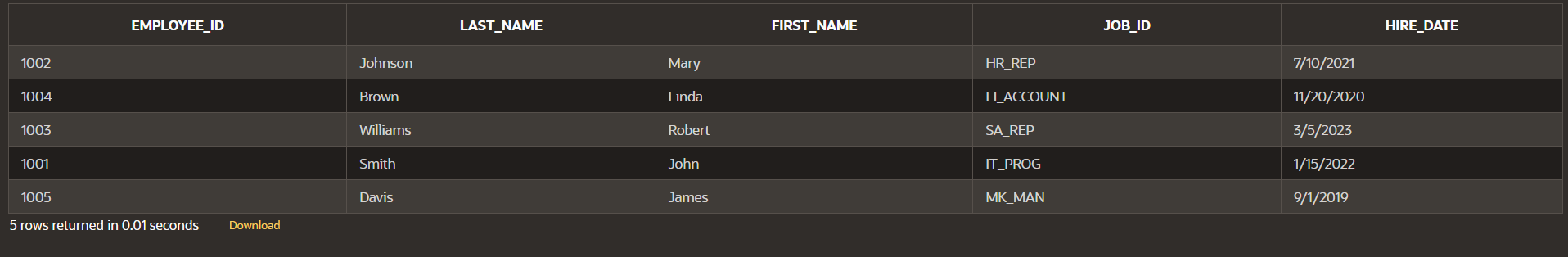
Location\_id Number(4)

);



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

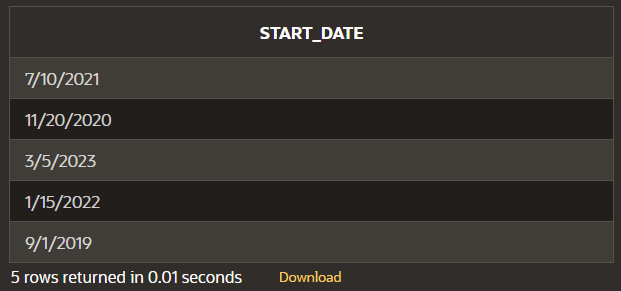
SELECT employee\_id,last\_name,first\_name,job\_id,hire\_date FROM EMPLOYEE;



1. Provide an alias STARTDATE for the hire date.

SELECT hire\_date AS "START\_DATE"

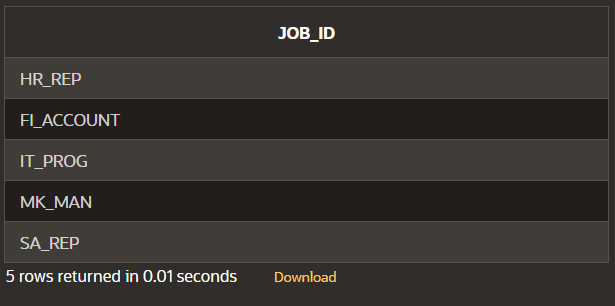
FROM EMPLOYEE;



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

SELECT UNIQUE job\_id

FROM EMPLOYEE;

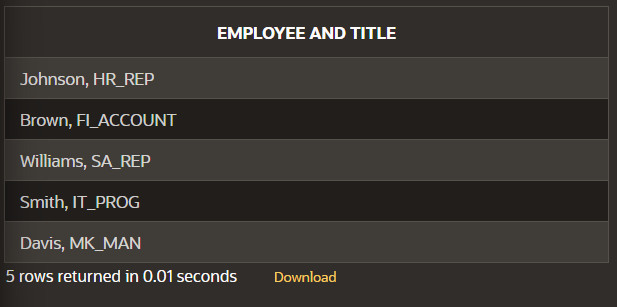


1. Display the last name concatenated with the job ID , separated by a comma and space, and name the column EMPLOYEE and TITLE.

SELECT last\_name || ', ' || job\_id

AS "EMPLOYEE AND TITLE"

FROM EMPLOYEE;



1. 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 || ', '||

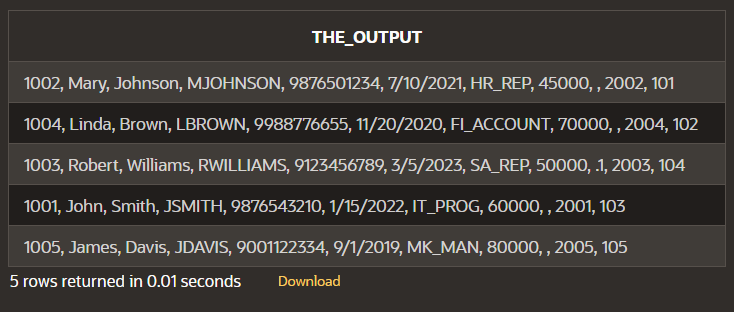
SALARY || ', '||

COMMISSION\_PCT || ', '||

MANAGER\_ID || ', '||

DEPARTMENT\_ID AS "THE\_OUTPUT"

from EMPLOYEE;



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

**EXERCISE-5**

**Restricting and Sorting data**

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 used in WHERE clause

**Syntax**

SELECT----------

FROM----------

WHERE condition;

**Example:**

SELECT employee\_id,last\_name, job\_id, deparment\_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, deparment\_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;

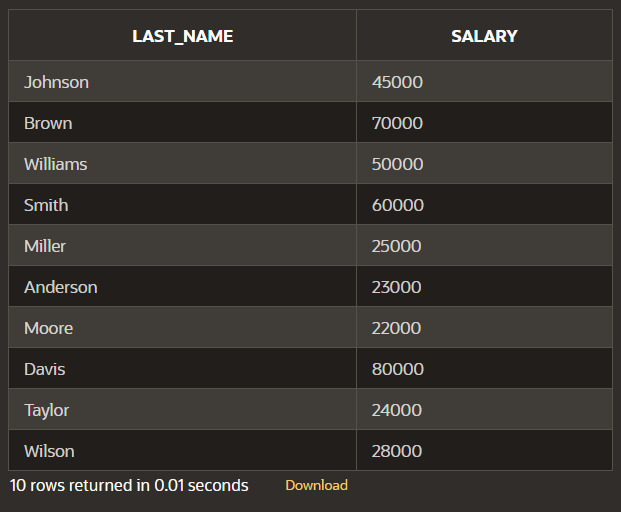
**Find the Solution for the following:**

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

SELECT last\_name,salary

FROM EMPLOYEE

WHERE salary > 12000;

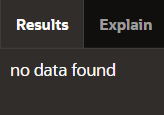


1. Create a query to display the employee last name and department number for employee number 176.

SELECT last\_name,department\_id

FROM EMPLOYEE

WHERE employee\_id = 176;

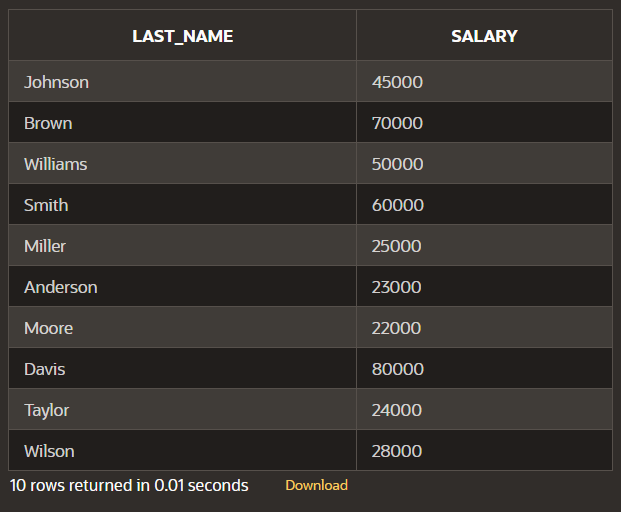


1. Create a query to display the last name and salary of employees whose salary is not in the range of 5000 and 12000. (hints: not between )

SELECT last\_name,salary

FROM EMPLOYEE

WHERE salary NOT BETWEEN 5000 AND 12000;



1. 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)

SELECT last\_name,job\_id,start\_date

FROM EMPLOYEE

WHERE hire\_date

BETWEEN TO\_DATE('1998-02-20', 'YYYY-MM-DD')

AND TO\_DATE('1998-05-01', 'YYYY-MM-DD');



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

SELECT last\_name, department\_id

FROM EMPLOYEE

WHERE department\_id in (20,50)

ORDER BY last\_name ASC;



1. 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)

SELECT last\_name AS "MONTHLY SALARY", salary AS "EMPLOYEE"

FROM EMPLOYEE

WHERE department\_id in (20,50) AND salary between 5000 AND 12000

ORDER BY last\_name ASC;



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

SELECT last\_name, hire\_date

FROM EMPLOYEE

WHERE TO\_CHAR(hire\_date, 'YYYY') LIKE '1994';

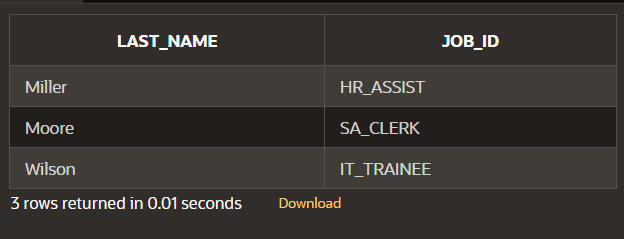


1. Display the last name and job title of all employees who do not have a manager.(hints: is null)

SELECT last\_name, job\_id

FROM EMPLOYEE

WHERE manager\_id IS NULL;



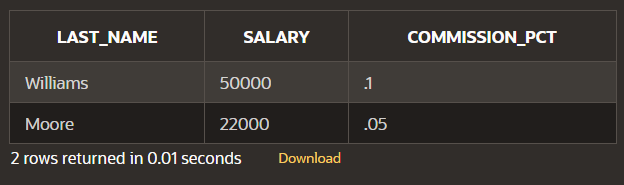
1. Display the last name, salary, and commission for all employees who earn commissions. Sort data in descending order of salary and commissions.(hints: is not nul,orderby)

SELECT last\_name, salary, commission\_pct

FROM EMPLOYEE

WHERE commission\_pct IS NOT NULL

ORDER BY salary DESC, commission\_pct DESC;



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

SELECT last\_name

FROM EMPLOYEE

WHERE last\_name LIKE '\_\_a%';



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

SELECT last\_name

FROM EMPLOYEE

WHERE last\_name LIKE '%a%'

AND last\_name LIKE '%e%';



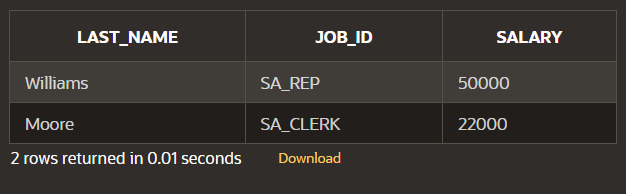
1. Display the last name and job and salary for all employees whose job is sales representative or stock clerk and whose salary is not equal to 2500 ,3500 or 7000.(hints:in,not in)

SELECT last\_name, job\_id, salary

FROM EMPLOYEE

WHERE job\_id IN ('SA\_REP', 'SA\_CLERK')

AND salary NOT IN (2500, 3500, 7000);



1. Display the last name, salary, and commission for all employees whose commission amount is 20%.(hints:use predicate logic)

SELECT last\_name, salary, commission\_pct

FROM EMPLOYEE

WHERE commission\_pct = 0.20;



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

**EXERCISE-6**

**Single Row Functions**

**Objective**

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

* Describe various types of functions available in SQL.
* Use character, number and date functions in SELECT statement.
* Describe the use of conversion functions.

**Single row functions:**

Manipulate data items.

Accept arguments and return one value.

Act on each row returned.

Return one result per row.

May modify the data type.

Can be nested.

Accept arguments which can be a column or an expression

**Syntax**

Function\_name(arg1,…argn)

An argument can be one of the following

* User-supplied constant
* Variable value
* Column name
* Expression

**SINGLE-ROW FUNCTIONS**

**CONVERSION**

**DATE**

**NUMBER**

**CHARACTER**

**GENERAL**

* Character Functions: Accept character input and can return both character and number values.
* Number functions: Accept numeric input and return numeric values.
* Date Functions: Operate on values of the DATE data type.
* Conversion Functions: Convert a value from one type to another.

**Character Functions**

Character Functions

Case-manipulation functions Character-manipulation functions

1. Lower 1. Concat
2. Upper 2. Substr
3. Initcap 3. Length

4. Instr

5. Lpad/Rpad

6. Trim

7. Repalce

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| lower(column/expr) | Converts alpha character values to lowercase |
| upper(column/expr) | Converts alpha character values to uppercase |
| initcap(column/expr) | Converts alpha character values the to uppercase for the first letter of each word, all other letters in lowercase |
| concat(column1/expr1, column2/expr2) | Concatenates the first character to the second character |
| substr(column/expr,m,n) | Returns specified characters from character value starting at character position m, n characters long |
| length(column/expr) | Returns the number of characters in the expression |
| instr(column/expr,’string’,m,n) | Returns the numeric position of a named string |
| lpad(column/expr, n,’string’) | Pads the character value right-justified to a total width of n character positions |
| rpad(column/expr,’string’,m,n) | Pads the character value left-justified to a total width of n character positions |
| trim(leading/trailing/both, trim\_character FROM trim\_source) | Enables you to trim heading or string. trailing or both from a character |
| replace(text, search\_string, replacement\_string) |  |

**Example:**

lower(‘SQL Course’)🡪sql course

upper(‘SQL Course’)🡪SQL COURSE

initcap(‘SQL Course’)🡪Sql Course

SELECT ‘The job id for’|| upper(last\_name||’is’||lower(job\_id) AS “EMPLOYEE DETAILS” FROM employees;

SELECT employee\_id, last\_name, department\_id

FROM employees

WHERE LOWER(last\_name)=’higgins’;

|  |  |
| --- | --- |
| **Function** | **Result** |
| CONCAT(‘hello’, ‘world’) | helloworld |
| Substr(‘helloworld’,1,5) | Hello |
| Length(‘helloworld’) | 10 |
| Instr(‘helloworld’,’w’) | 6 |
| Lpad(salary,10,’\*’) | \*\*\*\*\*24000 |
| Rpad(salary,10,’\*’) | 24000\*\*\*\*\* |
| Trim(‘h’ FROM ‘helloworld’) | elloworld |

|  |  |  |
| --- | --- | --- |
| **Command** | **Query** | **Output** |
| initcap(char); | *select initcap(“hello”) from dual;* | Hello |
| lower (char);  upper (char); | *select lower (‘HELLO’) from dual;*  *select upper (‘hello’) from dual;* | Hello  HELLO |
| ltrim (char,[set]); | *select ltrim (‘cseit’, ‘cse’) from dual;* | IT |
| rtrim (char,[set]); | *select rtrim (‘cseit’, ‘it’) from dual;* | CSE |
| replace (char,search string, replace string); | *select replace (‘jack and jue’, ‘j’, ‘bl’) from dual;* | black and blue |
| substr (char,m,n); | *select substr (‘information’, 3, 4) from dual;* | form |

**Example:**

SELECT employee\_id, CONCAT (first\_name,last\_name) NAME , job\_id,LENGTH(last\_name), INSTR(last\_name,’a’) “contains’a’?”

FROM employees WHERE SUBSTR(job\_id,4)=’ERP’;

**NUMBER FUNCTIONS**

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| round(column/expr, n) | Rounds the value to specified decimal |
| trunc(column/expr,n) | Truncates value to specified decimal |
| mod(m,n) | Returns remainder of division |

**Example**

|  |  |
| --- | --- |
| Function | Result |
| round(45.926,2) | 45.93 |
| trunc(45.926,2) | 45.92 |
| mod(1600,300) | 100 |

SELECT ROUND(45.923,2), ROUND(45.923,0), ROUND(45.923,-1) FROM dual;

**NOTE:**  Dual is a dummy table you can use to view results from functions and calculations.

SELECT TRUNC(45.923,2), TRUNC(45.923), TRUNC(45.923,-2) FROM dual;

SELECT last\_name,salary,MOD(salary,5000) FROM employees WHERE job\_id=’sa\_rep’;

**Working with Dates**

The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.

• The default date display format is DD-MON-RR.

– Enables you to store 21st-century dates in the 20th century by specifying only the last

two digits of the year

– Enables you to store 20th-century dates in the 21st century in the same way

**Example**

SELECT last\_name, hire\_date FROM employees WHERE hire\_date < '01-FEB-88;

**Working with Dates**

SYSDATE is a function that returns:

• Date

• Time

**Example**

**Display the current date using the DUAL table.**

SELECT SYSDATE FROM DUAL;

**Arithmetic with Dates**

• Add or subtract a number to or from a date for a resultant date value.

• Subtract two dates to find the number of days between those dates.

• Add hours to a date by dividing the number of hours by 24.

**Arithmetic with Dates**

Because the database stores dates as numbers, you can perform calculations using arithmetic

Operators such as addition and subtraction. You can add and subtract number constants as well as dates.

You can perform the following operations:

**Operation Result Description**

date + number Date Adds a number of days to a date

date – number Date Subtracts a number of days from a date

date – date Number of days Subtracts one date from another

date + number/24 Date Adds a number of hours to a date

**Example**

SELECT last\_name, (SYSDATE-hire\_date)/7 AS WEEKS

FROM employees

WHERE department\_id = 90;

**Date Functions**



**Date Functions**

Date functions operate on Oracle dates. All date functions return a value of DATE data type

except MONTHS\_BETWEEN, which returns a numeric value.

• MONTHS\_BETWEEN(date1, date2)::: Finds the number of months between date1 and date2. The result can be positive or negative. If date1 is later than date2, the result is positive; if date1 is earlier than date2, the result is negative. The noninteger part of the result represents a portion of the month.

• ADD\_MONTHS(date, n)::: Adds n number of calendar months to date. The value of n must be an integer and can be negative.

• NEXT\_DAY(date, 'char')::: Finds the date of the next specified day of the week ('char') following date. The value of char may be a number representing a day or a character string.

• LAST\_DAY(date)::: Finds the date of the last day of the month that contains date

• ROUND(date[,'fmt'])::: Returns date rounded to the unit that is specified by the format model fmt. If the format model fmt is omitted, date is rounded to the nearest day.

* TRUNC(date[, 'fmt'])::: Returns date with the time portion of the day truncated to the unit that is specified by the format model fmt. If the format model fmt is omitted, date is truncated to the nearest day.

**Using Date Functions**



**Example**

Display the employee number, hire date, number of months employed, sixmonth review date, first Friday after hire date, and last day of the hire month for all employees who have been employed for fewer than 70 months.

SELECT employee\_id, hire\_date,MONTHS\_BETWEEN (SYSDATE, hire\_date) TENURE,ADD\_MONTHS (hire\_date, 6) REVIEW,NEXT\_DAY (hire\_date, 'FRIDAY'), LAST\_DAY(hire\_date)

FROM employees

WHERE MONTHS\_BETWEEN (SYSDATE, hire\_date) < 70;

**Conversion Functions**

This covers the following topics:

• Writing a query that displays the current date

• Creating queries that require the use of numeric, character, and date functions

• Performing calculations of years and months of service for an employee



**Implicit Data Type Conversion**

For assignments, the Oracle server can automatically convert the following:



For example, the expression hire\_date > '01-JAN-90' results in the implicit conversion from the string '01-JAN-90' to a date.

For expression evaluation, the Oracle Server can automatically convert the following:



**Explicit Data Type Conversion**



**SQL provides three functions to convert a value from one data type to another:**

**Example:**

**Using the TO\_CHAR Function with Dates**

TO\_CHAR(date, 'format\_model')

**The format model:**

• Must be enclosed by single quotation marks

• Is case-sensitive

• Can include any valid date format element

• Has an fm element to remove padded blanks or suppress leading zeros

• Is separated from the date value by a comma

SELECT employee\_id, TO\_CHAR(hire\_date, 'MM/YY') Month\_Hired

FROM employees WHERE last\_name = 'Higgins';

**Elements of the Date Format Model**



**Sample Format Elements of Valid Date** 

**Date Format Elements:** Time Formats

Use the formats that are listed in the following tables to display time information and literals

and to change numerals to spelled numbers.





**Example**

SELECT last\_name,

TO\_CHAR(hire\_date, 'fmDD Month YYYY') AS HIREDATE

FROM employees;

Modify example to display the dates in a format that appears as “Seventeenth of June 1987 12:00:00 AM.”

SELECT last\_name,

TO\_CHAR (hire\_date, 'fmDdspth "of" Month YYYY fmHH:MI:SS AM') HIREDATE

FROM employees;

**Using the TO\_CHAR Function with Numbers**

TO\_CHAR(number, 'format\_model')  
These are some of the format elements that you can use with the TO\_CHAR function to display a number value as a character:



**Number Format Elements**

If you are converting a number to the character data type, you can use the following format elements:



SELECT TO\_CHAR(salary, '$99,999.00') SALARY

FROM employees

WHERE last\_name = 'Ernst';

**Using the TO\_NUMBER and TO\_DATE Functions**

• Convert a character string to a number format using the TO\_NUMBER function:

TO\_NUMBER(char[, 'format\_model']

• Convert a character string to a date format using the TO\_DATE function:

TO\_DATE(char[, 'format\_model']

• These functions have an fx modifier. This modifier specifies the exact matching for the character

argument and date format model of a TO\_DATE function.

The fx modifier specifies exact matching for the character argument and date format model of a TO\_DATE function:

• Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.

• The character argument cannot have extra blanks. Without fx, Oracle ignores extra blanks.

• Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without fx, numbers in the character argument can omit leading zeros.

SELECT last\_name, hire\_date

FROM employees

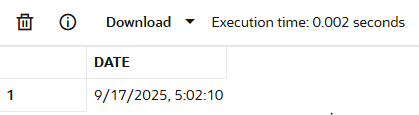
WHERE hire\_date = TO\_DATE('May 24, 1999', 'fxMonth DD, YYYY');

**Find the Solution for the following:**

1. Write a query to display the current date. Label the column Date.

SELECT SYSDATE "DATE"

FROM DUAL;

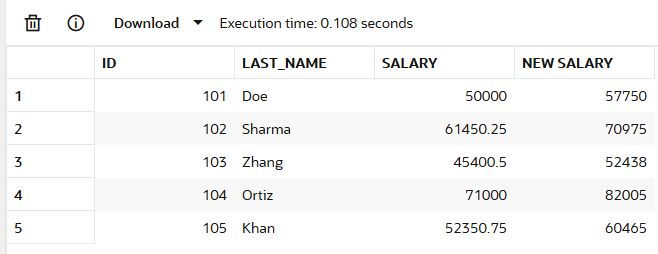


1. The HR department needs a report to display the employee number, last name, salary, and increased by 15.5% (expressed as a whole number) for each employee. Label the column New Salary.

select id,last\_name,salary,

round(salary\*1.155)"New Salary"

FROM MY\_EMPLOYEE;



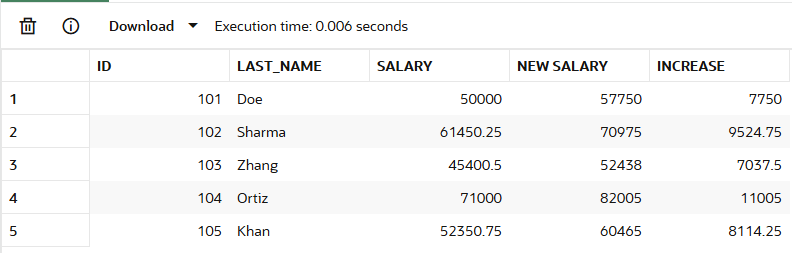
1. Modify your query lab\_03\_02.sql to add a column that subtracts the old salary from the new salary. Label the column Increase.

select id,last\_name,salary,

round(salary\*1.155)"New Salary",

round(salary\*1.155)-salary"Increase"

FROM MY\_EMPLOYEE;



1. Write a query that displays the last name (with the first letter uppercase and all other letters lowercase) and the length of the last name for all employees whose name starts with the letters J, A, or M. Give each column an appropriate label. Sort the results by the employees’ last names.

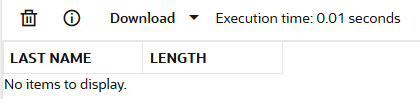
select initcap(last\_name)"LAST NAME",

length(last\_name)"LENGTH"

FROM MY\_EMPLOYEE where upper(substr(last\_name,1,1))

in ('J','A','M')

Order by last\_name;



1. Rewrite the query so that the user is prompted to enter a letter that starts the last name. For example, if the user enters H when prompted for a letter, then the output should show all employees whose last name starts with the letter H.

select first\_name,last\_name

from MY\_EMPLOYEE

where last\_name like :letter || '%';



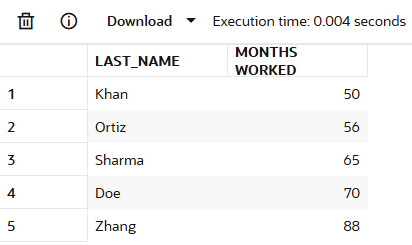
1. The HR department wants to find the length of employment for each employee. For each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column MONTHS\_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.

select last\_name,round(months\_between(SYSDATE,hire\_date))

As "Months Worked"

from MY\_EMPLOYEE

Order by round(months\_between(SYSDATE,hire\_date));



**Note:** Your results will differ.

1. Create a report that produces the following for each employee:

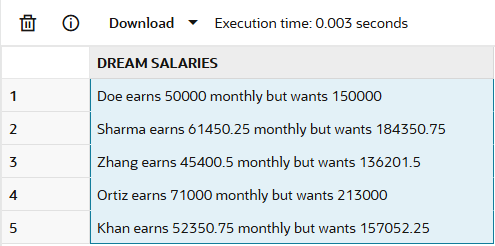
<employee last name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

select last\_name || ' earns ' || salary ||

' monthly but wants ' || (salary\*3)

AS "Dream Salaries"

FROM MY\_EMPLOYEE;



1. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left-padded with the $ symbol. Label the column SALARY.

select last\_name,

Lpad(salary,15,'$')"Salary"

FROM MY\_EMPLOYEE;



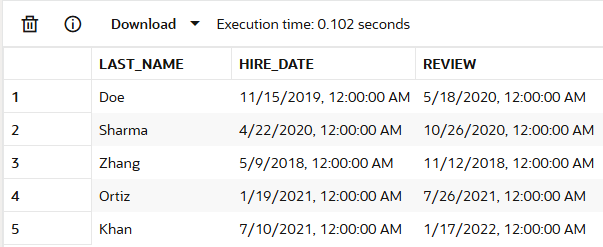
1. Display each employee’s last name, hire date, and salary review date, which is the first Monday after six months of service. Label the column REVIEW. Format the dates to appear in the format similar to “Monday, the Thirty-First of July, 2000.”

select last\_name, hire\_date,

NEXT\_DAY(ADD\_MONTHS(hire\_date,6),'Monday')

AS "REVIEW"

FROM MY\_EMPLOYEE;



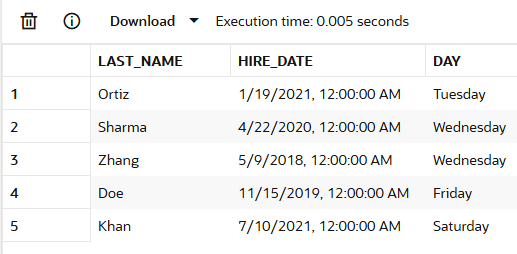
1. Display the last name, hire date, and day of the week on which the employee started. Label the column DAY. Order the results by the day of the week, starting with Monday.

select last\_name, hire\_date,

to\_char(hire\_date,'Day')AS "DAY"

FROM MY\_EMPLOYEE

Order by to\_char(hire\_date,'D');



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

**EXERCISE-7**

**Displaying data from multiple tables**

**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

SELECTemployees.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-Equijoins**

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 alslso 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.

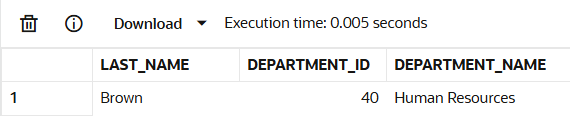
SELECT e.last\_name,e.department\_id, d.dept\_name

AS department\_name

FROM employees e

JOIN department d

ON e.department\_id = d.dept\_id;



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

SELECT DISTINCT e.job\_id, d.location\_id

FROM employees e

JOIN department d

ON e.department\_id = d.dept\_id

WHERE e.department\_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

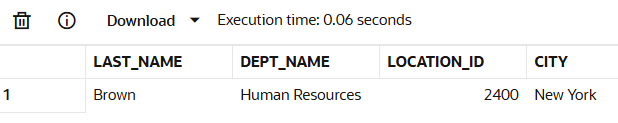
SELECT e.last\_name,d.dept\_name,d.location\_id,l.city

FROM employees e

JOIN department d ON e.department\_id = d.dept\_id

JOIN locations l ON d.location\_id = l.location\_id

WHERE e.commission\_pct IS NOT NULL;



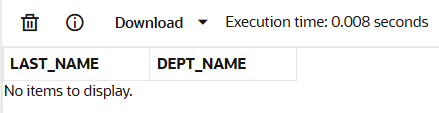
1. Display the employee last name and department name for all employees who have an a(lowercase) in their last names. P

SELECT e.last\_name, d.dept\_name

FROM employees e

JOIN department d ON e.department\_id = d.dept\_id

WHERE LOWER(e.last\_name) LIKE '%a%';



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

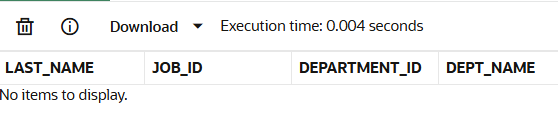
SELECT e.last\_name,e.job\_id,e.department\_id,d.dept\_name

FROM employees e

JOIN department d ON e.department\_id = d.dept\_id

JOIN locations l ON d.location\_id = l.location\_id

WHERE LOWER(l.city) = '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

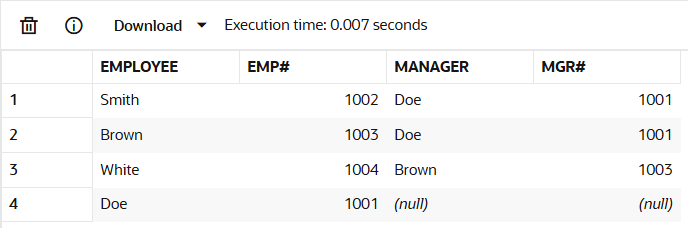
SELECT e.last\_name AS Employee,e.employee\_id AS "Emp#",

m.last\_name AS Manager,m.employee\_id AS "Mgr#"

FROM employees e

LEFT JOIN employees m ON

e.manager\_id = m.employee\_id;



7. Modify lab4\_6.sql to display all employees including King, who has no manager. Order the results by the employee number.

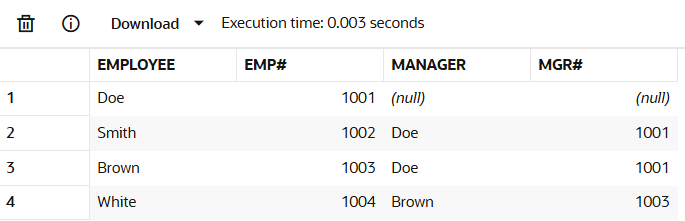
SELECT e.last\_name AS Employee,e.employee\_id AS "Emp#",

m.last\_name AS Manager,m.employee\_id AS "Mgr#"

FROM employees e

LEFT JOIN employees m ON e.manager\_id = m.employee\_id

ORDER BY e.employee\_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

SELECT e1.last\_name AS Employee, e1.department\_id AS "Dept No",

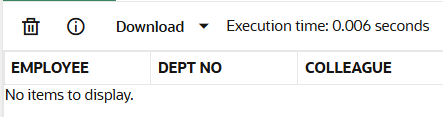
e2.last\_name AS "Colleague"

FROM employees e1

JOIN employees e2 ON

e1.department\_id = e2.department\_id

WHERE e1.employee\_id != e2.employee\_id;



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

DESC job\_grades;

SELECT e.last\_name,e.job\_id,

d.dept\_name,e.salary,j.grade\_level

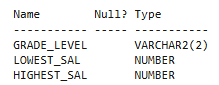
FROM employees e

JOIN department d ON

e.department\_id = d.dept\_id

JOIN job\_grade j ON e.salary

BETWEEN j.lowest\_sal AND j.highest\_sal;



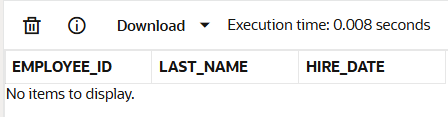
10. Create a query to display the name and hire date of any employee hired after employee Davies.

SELECT e.employee\_id,e.last\_name,e.hire\_date

FROM employees e

WHERE e.hire\_date > (SELECT hire\_date

FROM employees WHERE last\_name = '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.

SELECT e.last\_name AS Employee,e.hire\_date AS "Emp Hired",

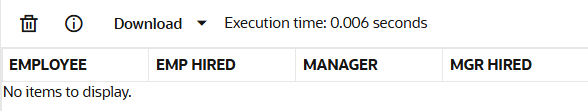
m.last\_name AS Manager,m.hire\_date AS "Mgr Hired"

FROM employees e

JOIN employees m ON

e.manager\_id = m.employee\_id

WHERE e.hire\_date < m.hire\_date;



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

**EXERCISE-8**

**Aggregating Data Using Group Functions**

***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:



**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\_idHAVING 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 **TRUE**

2. Group functions include nulls in calculations.

True/False **FALSE**

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

True/False **TRUE**

**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

SELECT

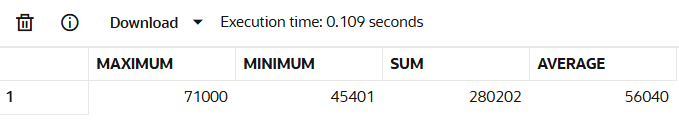
ROUND(MAX(salary)) AS Maximum,

ROUND(MIN(salary)) AS Minimum,

ROUND(SUM(salary)) AS Sum,

ROUND(AVG(salary)) AS Average

FROM MY\_EMPLOYEE;



5. Modify the above query to display the minimum, maximum, sum, and

average salary for each job type.

SELECT

job\_id,

ROUND(MIN(salary)) AS Minimum,

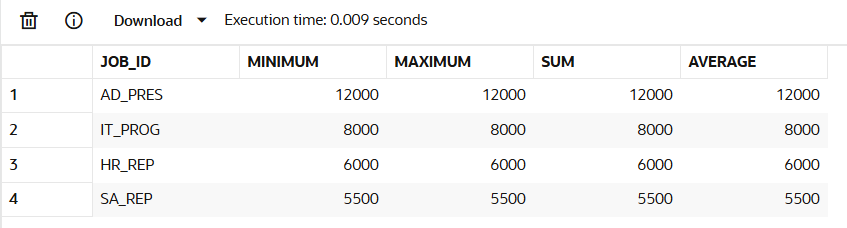
ROUND(MAX(salary)) AS Maximum,

ROUND(SUM(salary)) AS Sum,

ROUND(AVG(salary)) AS Average

FROM employees

GROUP BY job\_id;



6.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.

SELECT

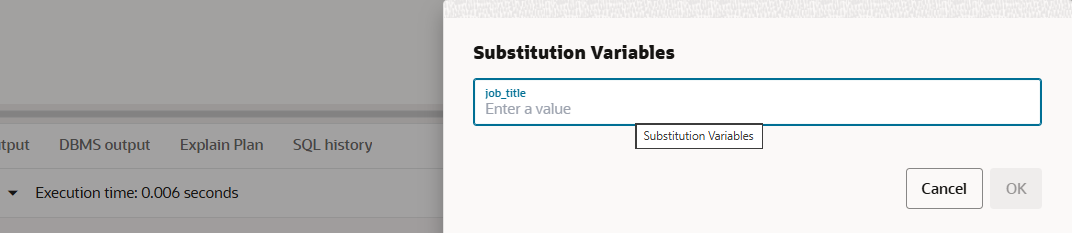
job\_id,

COUNT(\*) AS Number\_of\_People

FROM employees

WHERE job\_id = '&job\_title'

GROUP BY job\_id;



7. 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.*

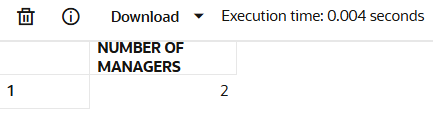
SELECT

COUNT(DISTINCT manager\_id)

AS "Number of Managers"

FROM employees

WHERE manager\_id IS NOT NULL;



8. Find the difference between the highest and lowest salaries. Label the column

DIFFERENCE.

SELECT

MAX(salary) - MIN(salary)

AS Difference

FROM employees;



9. 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.

SELECT

manager\_id,

MIN(salary) AS Lowest\_Salary

FROM employees

WHERE manager\_id IS NOT NULL

GROUP BY manager\_id

HAVING MIN(salary) > 6000

ORDER BY Lowest\_Salary DESC;



10. 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.

SELECT

COUNT(\*) AS Total\_Employees,

SUM(CASE WHEN EXTRACT(YEAR FROM hire\_date) = 1995 THEN 1 ELSE 0 END) AS Hired\_1995,

SUM(CASE WHEN EXTRACT(YEAR FROM hire\_date) = 1996 THEN 1 ELSE 0 END) AS Hired\_1996,

SUM(CASE WHEN EXTRACT(YEAR FROM hire\_date) = 1997 THEN 1 ELSE 0 END) AS Hired\_1997,

SUM(CASE WHEN EXTRACT(YEAR FROM hire\_date) = 1998 THEN 1 ELSE 0 END) AS Hired\_1998

FROM employees;



11. 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.

SELECT

job\_id AS Job,

SUM(CASE WHEN department\_id = 20 THEN salary ELSE 0 END) AS Dept\_20,

SUM(CASE WHEN department\_id = 50 THEN salary ELSE 0 END) AS Dept\_50,

SUM(CASE WHEN department\_id = 80 THEN salary ELSE 0 END) AS Dept\_80,

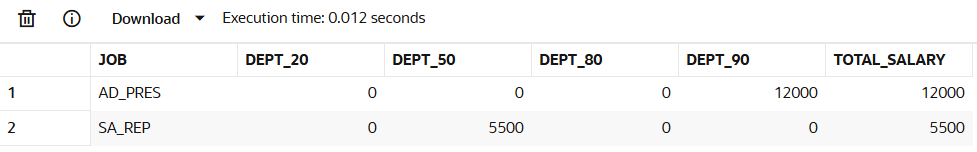
SUM(CASE WHEN department\_id = 90 THEN salary ELSE 0 END) AS Dept\_90,

SUM(salary) AS Total\_Salary

FROM employees

WHERE department\_id IN (20, 50, 80, 90)

GROUP BY job\_id;



12.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.

SELECT

d.dept\_name AS Name,

d.location\_id AS Location,

COUNT(e.employee\_id) AS "Number of People",

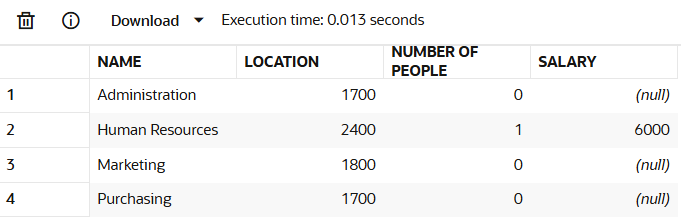
ROUND(AVG(e.salary), 2) AS Salary

FROM department d

LEFT JOIN employees e ON d.dept\_id = e.department\_id

GROUP BY d.dept\_name, d.location\_id

ORDER BY d.dept\_name;



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

**EXERCISE-9**

**Sub queries**

**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 employeesWHERE 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).

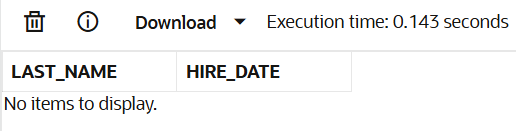
SELECT last\_name, hire\_date

FROM employees

WHERE department\_id=(SELECT department\_id FROM employees

WHERE last\_name = 'Zlotkey')

AND last\_name <> 'Zlotkey';



1. 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.

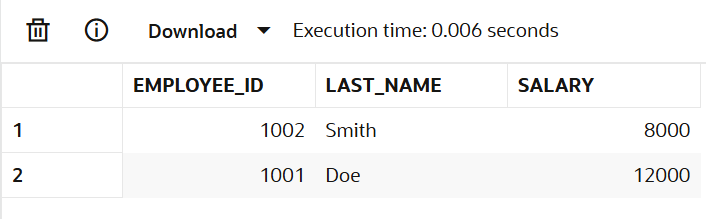
SELECT employee\_id, last\_name, salary

FROM employees

WHERE salary > (SELECT AVG(salary)

FROM employees)

ORDER BY salary ASC;



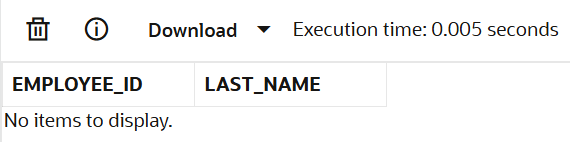
1. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains a *u*.

SELECT employee\_id, last\_name

FROM employees

WHERE department\_id IN ( SELECT department\_id

FROM employees WHERE last\_name LIKE '%u%');



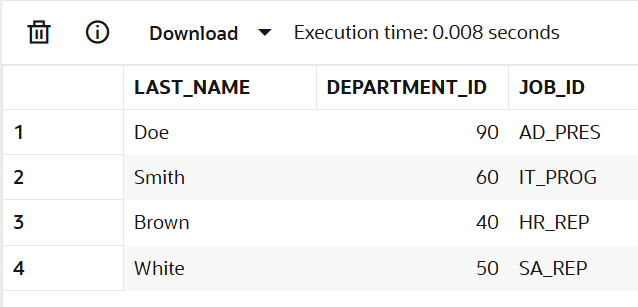
1. The HR department needs a report that displays the last name, department number, and job ID of all employees whose department location ID is 1700.

SELECT last\_name, department\_id, job\_id

FROM employees

WHERE department\_id IN ( SELECT department\_id

FROM department WHERE location\_id = 1700 );



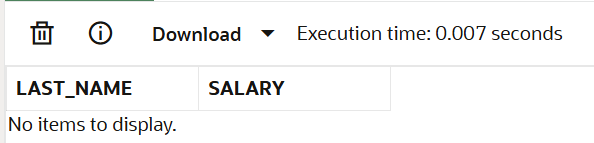
1. Create a report for HR that displays the last name and salary of every employee who reports to King.

SELECT last\_name, salary

FROM employees

WHERE manager\_id = ( ELECT employee\_id

FROM employees WHERE last\_name = 'King' );



1. Create a report for HR that displays the department number, last name, and job ID for

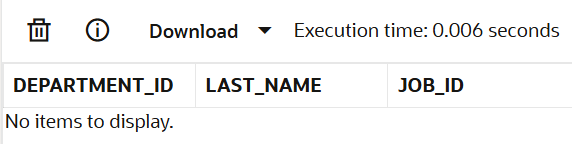
every employee in the Executive department.

SELECT department\_id, last\_name, job\_id

FROM employees

WHERE department\_id IN ( SELECT department\_id

FROM department WHERE dept\_name = 'Executive');



1. 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*.

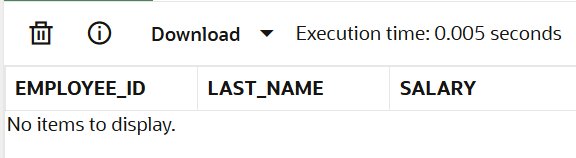
SELECT employee\_id, last\_name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees)

AND department\_id IN ( SELECT department\_id

FROM employees WHERE last\_name LIKE '%u%');



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

**EXERCISE-10**

**USING THE SET OPERATORS**

**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*.



**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

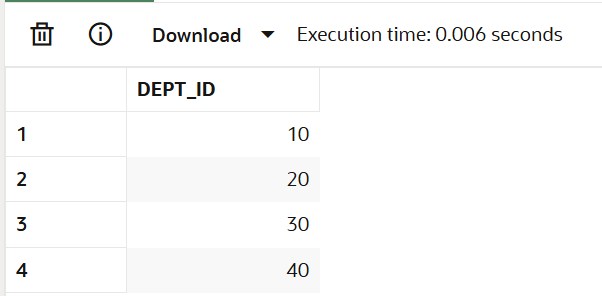
the job ID ST\_CLERK. Use set operators to create this report.

SELECT dept\_id FROM department

MINUS

SELECT department\_id FROM employees

WHERE job\_id = 'ST\_CLERK';



2. The HR department needs a list of countries that have no departments located in them.

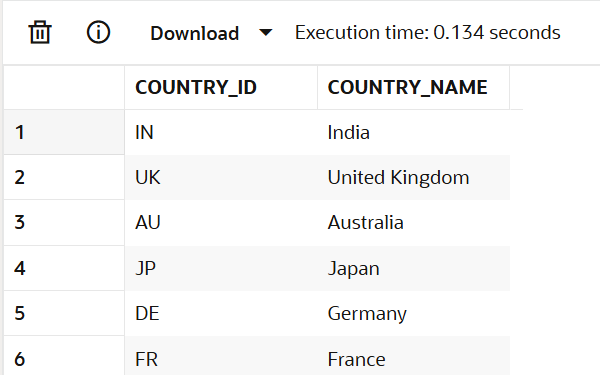
Display the country ID and the name of the countries. Use set operators to create this report.

SELECT country\_id, country\_name FROM countries

MINUS

SELECT l.country\_id, c.country\_name FROM locations l,

countries c WHERE l.country\_id = c.country\_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.

SELECT job\_id, department\_id, 'a' sort\_order FROM employees

WHERE department\_id = 10

UNION

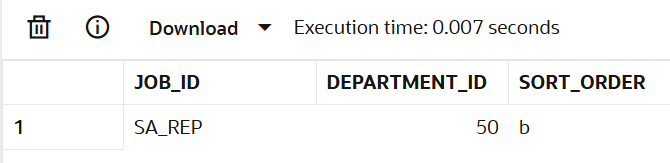
SELECT job\_id, department\_id, 'b' FROM employees

WHERE department\_id = 50

UNION

SELECT job\_id, department\_id, 'c' FROM employees

WHERE department\_id = 20 ORDER BY sort\_order;



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).

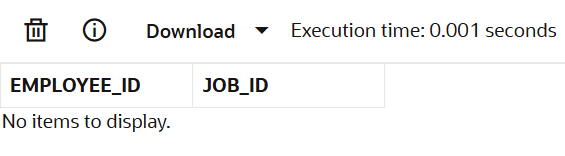
SELECT employee\_id, job\_id

FROM employees

INTERSECT

SELECT employee\_id, job\_id

FROM job\_history;



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.

SELECT last\_name, department\_id, TO\_CHAR(NULL)

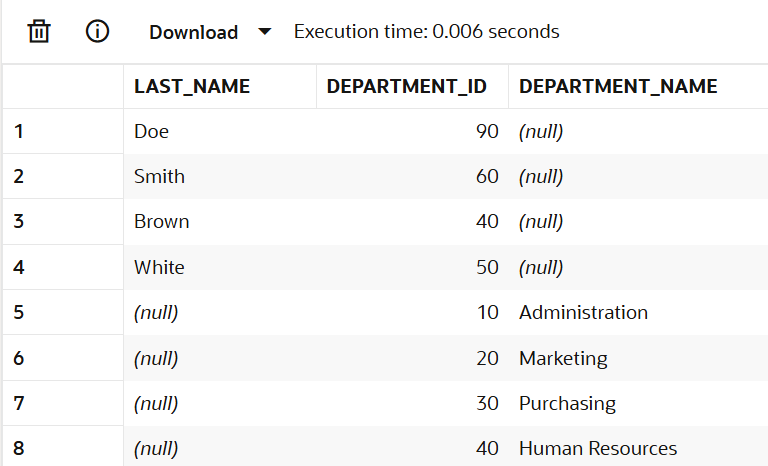
AS department\_name

FROM employees

UNION

SELECT TO\_CHAR(NULL), dept\_id, dept\_name

FROM department;



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

**EXERCISE-11**

**CREATING VIEWS**

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 ase 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** (W**ithout 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:**

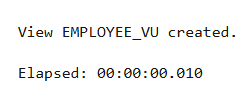
1. Create a view called EMPLOYEE\_VU based on the employee numbers, employee names and department numbers from the EMPLOYEES table. Change the heading for the employee name to EMPLOYEE.

CREATE OR REPLACE VIEW employee\_vu AS

SELECT employee\_id, last\_name

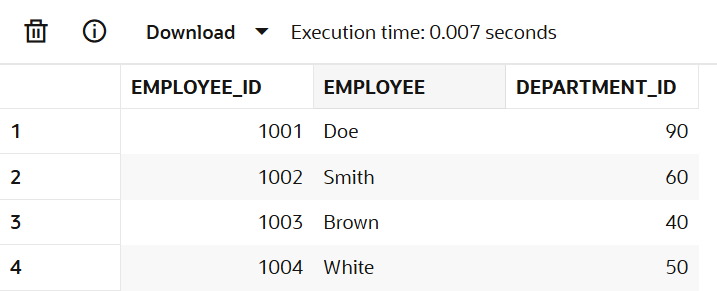
AS employee, department\_id

FROM employees;



1. Display the contents of the EMPLOYEES\_VU view.

SELECT \* FROM employee\_vu;

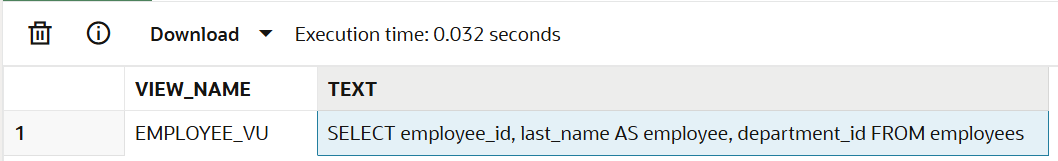


1. Select the view name and text from the USER\_VIEWS data dictionary views.

SELECT view\_name, text

FROM user\_views

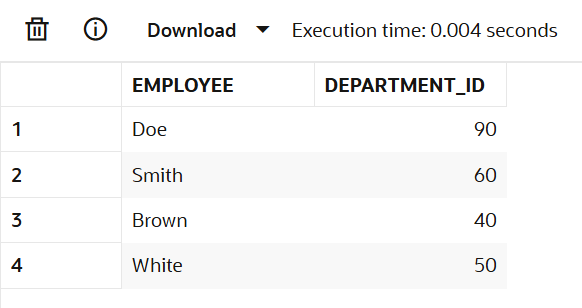
WHERE view\_name = 'EMPLOYEE\_VU';



1. Using your EMPLOYEES\_VU view, enter a query to display all employees names and department.

SELECT employee, department\_id

FROM employee\_vu;



1. 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.

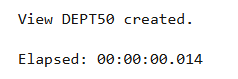
CREATE OR REPLACE VIEW dept50 (empno, employee, deptno) AS

SELECT employee\_id, last\_name, department\_id

FROM employees

WHERE department\_id = 50

WITH CHECK OPTION CONSTRAINT dept50\_ck;



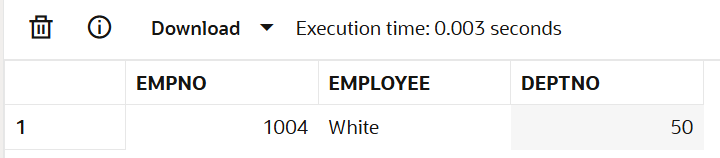
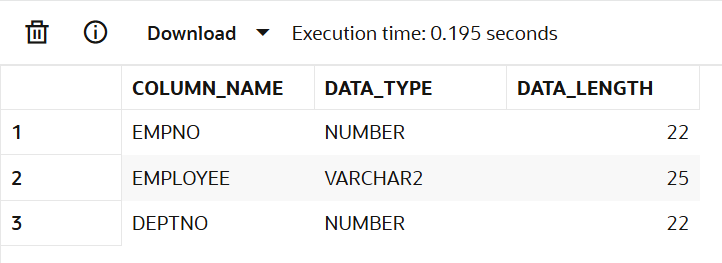
1. Display the structure and contents of the DEPT50 view.

SELECT column\_name, data\_type, data\_length

FROM user\_tab\_columns

WHERE table\_name = 'DEPT50';

SELECT \* FROM dept50;

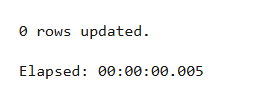


1. Attempt to reassign Matos to department 80.

UPDATE dept50

SET deptno = 80

WHERE employee = 'Matos';



1. 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 OR REPLACE VIEW salary\_vu AS

SELECT e.last\_name AS Employee,

d.dept\_name AS Department,

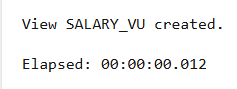
e.salary AS Salary,

j.grade\_level AS Grade

FROM employees e

JOIN department d ON e.department\_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 |  |

**EXERCISE 12**

# Intro to Constraints; NOT NULL and UNIQUE Constraints

Global Fast Foods has been very successful this past year and has opened several new stores. They need to add a table to their database to store information about each of their store’s locations. The owners want to make sure that all entries have an identification number, date opened, address, and city and that no other entry in the table can have the same email address. Based on this information, answer the following questions about the global\_locations table. Use the table for your answers.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Global Fast Foods global\_locations Table** | | | | | | |
| **NAME** | **TYPE** | **LENGTH** | **PRECISION** | **SCALE** | **NULLABLE** | **DEFAULT** |
| Id |  |  |  |  |  |  |
| name |  |  |  |  |  |  |
| date\_opened |  |  |  |  |  |  |
| address |  |  |  |  |  |  |
| city |  |  |  |  |  |  |
| zip/postal code |  |  |  |  |  |  |
| phone |  |  |  |  |  |  |
| email |  |  |  |  |  |  |
| manager\_id |  |  |  |  |  |  |
| Emergency contact |  |  |  |  |  |  |

1. What is a “constraint” as it relates to data integrity?

A constraint is a rule applied to data columns in relational databases to enforce data integrity, such as preventing duplicates or ensuring non-null values. Constraints help maintain correctness and validity of the data stored in the table

1. What are the limitations of constraints that may be applied at the column level and at the table level?

Column-level constraints are defined alongside individual columns and can only apply to that specific column (e.g., NOT NULL, UNIQUE).

Table-level constraints are defined after all columns are listed and can involve multiple columns, such as composite PRIMARY KEY or FOREIGN KEY constraints, and named constraints

1. Why is it important to give meaningful names to constraints?

Meaningful constraint names help in understanding the business rule being enforced and make debugging or managing the database easier (e.g., unique\_email for a UNIQUE constraint on the email column).

1. Based on the information provided by the owners, choose a datatype for each column. Indicate the length, precision, and scale for each NUMBER datatype. Use “(nullable)” to indicate those columns that can have null values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column name** | **Example Data Type** | **Length/Precision/Scale** | **Nullable** |
| id | NUMBER | (6) | NOT NULL |
| name | VARCHAR2 | (50) | NOT NULL |
| date\_opened | DATE |  | NOT NULL |
| address | VARCHAR2 | (100) | NOT NULL |
| city | VARCHAR2 | (30) | NOT NULL |
| zip/postal\_code | VARCHAR2 | (10) | NULLABLE |
| phone | VARCHAR2 | (15) | NULLABLE |
| email | VARCHAR2 | (50) | UNIQUE, NULLABLE |
| manager\_id | NUMBER | (6) | NULLABLE |
| emergency\_contact | VARCHAR2 | (50) | NULLABLE |

1. Write the CREATE TABLE statement for the Global Fast Foods locations table to define the constraints at the column level.

CREATE TABLE global\_locations (

id NUMBER(6) NOT NULL,

name VARCHAR2(50) NOT NULL,

date\_opened DATE NOT NULL,

address VARCHAR2(100) NOT NULL,

city VARCHAR2(30) NOT NULL,

zip\_postal\_code VARCHAR2(10),

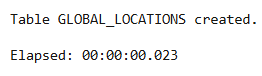
phone VARCHAR2(15),

email VARCHAR2(50) UNIQUE,

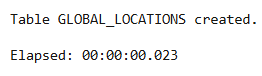
manager\_id NUMBER(6),

emergency\_contact VARCHAR2(50)

);

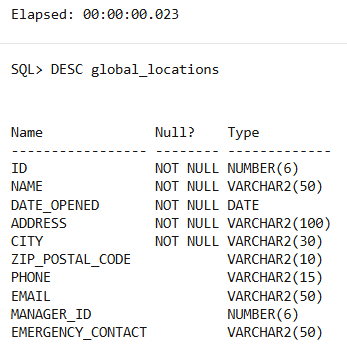


1. Execute the CREATE TABLE statement in Oracle Application Express.

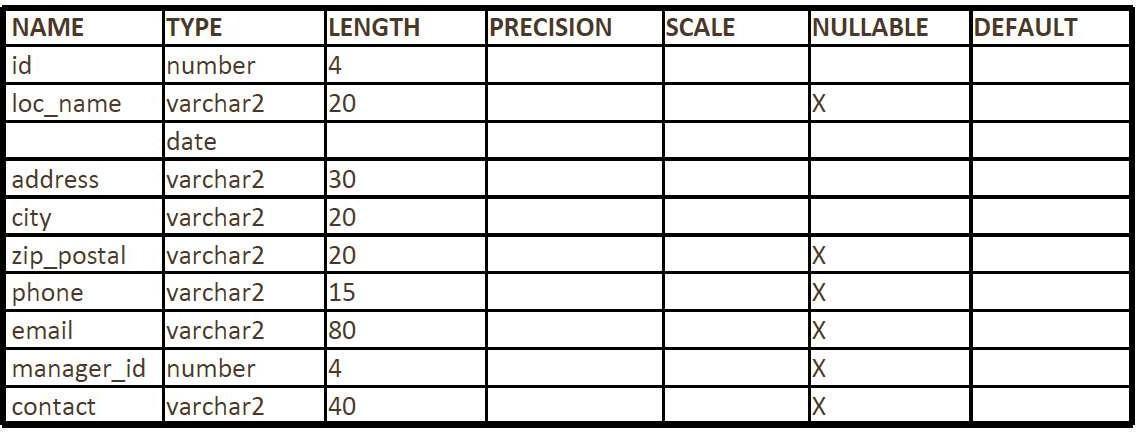


1. Execute a DESCRIBE command to view the Table Summary information.

DESC global\_locations;



1. Rewrite the CREATE TABLE statement for the Global Fast Foods locations table to define the UNIQUE constraints at the table level. Do not execute this statement.



# EXERCISE 13

# Creating Views

1. What are three uses for a view from a DBA’s perspective?

Simplified Querying: Views allow you to write complex queries once and reference them easily, reducing code repetition.

Security Control: Views restrict access to sensitive columns and rows, enabling users to see only permitted data.

Data Abstraction: Views present data in a meaningful or business-relevant format, hiding underlying schema complexity.

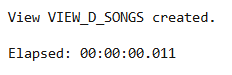
1. Create a simple view called view\_d\_songs that contains the ID, title and artist from the DJs on Demand table for each “New Age” type code. In the subquery, use the alias “Song Title” for the title column.

CREATE VIEW view\_d\_songs AS

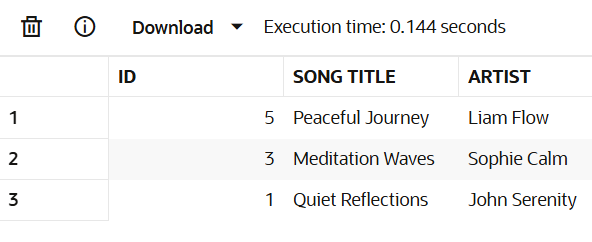
SELECT id, title AS "Song Title", artist

FROM DJs\_on\_Demand

WHERE type\_code = 'New Age';



1. SELECT \* FROM view\_d\_songs. What was returned?



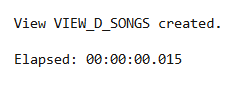
1. REPLACE view\_d\_songs. Add type\_code to the column list. Use aliases for all columns.

CREATE OR REPLACE VIEW view\_d\_songs AS

SELECT id AS "Song ID", title AS "Song Title", artist AS "Song Artist", type\_code AS "Music Category"

FROM DJs\_on\_Demand

WHERE type\_code = 'New Age'



1. Jason Tsang, the disk jockey for DJs on Demand, needs a list of the past events and those planned for the coming months so he can make arrangements for each event’s equipment setup. As the company manager, you do not want him to have access to the price that clients paid for their events. Create a view for Jason to use that displays the name of the event, the event date, and the theme description. Use aliases for each column name.

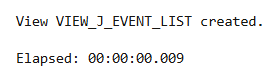
CREATE VIEW view\_j\_event\_list AS

SELECT event\_name AS "Event Name",

event\_date AS "Event Date",

theme\_description AS "Theme"

FROM Events;



1. It is company policy that only upper-level management be allowed access to individual employee salaries. The department managers, however, need to know the minimum, maximum, and average salaries, grouped by department. Use the Oracle database to prepare a view that displays the needed information for department managers.

CREATE VIEW view\_dept\_salary\_summary AS

SELECT department\_id AS "Department",

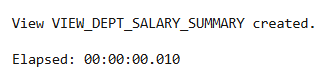
MIN(salary) AS "Min Salary",

MAX(salary) AS "Max Salary",

AVG(salary) AS "Avg Salary"

FROM employees

GROUP BY department\_id;



**EXERCISE-14**

**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.

CREATE SEQUENCE DEPT\_ID\_SEQ

START WITH 200

INCREMENT BY 10

MAXVALUE 1000

NOCYCLE



2. Write a query in a script to display the following information about your sequences: sequence

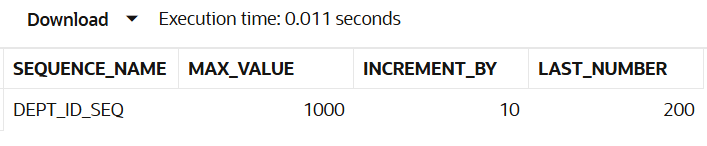
name, maximum value, increment size, and last number

SELECT sequence\_name, max\_value,

increment\_by, last\_number

FROM user\_sequences

WHERE sequence\_name = 'DEPT\_ID\_SEQ'



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

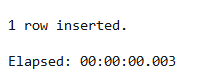
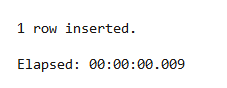
Administration. Confirm your additions. Run the commands in your script.

INSERT INTO DEPARTMENT (DEPT\_ID, DEPT\_NAME)

VALUES (DEPT\_ID\_SEQ.NEXTVAL, 'Education');

INSERT INTO DEPARTMENT (DEPT\_ID, DEPT\_NAME)

VALUES (DEPT\_ID\_SEQ.NEXTVAL, 'Administration');

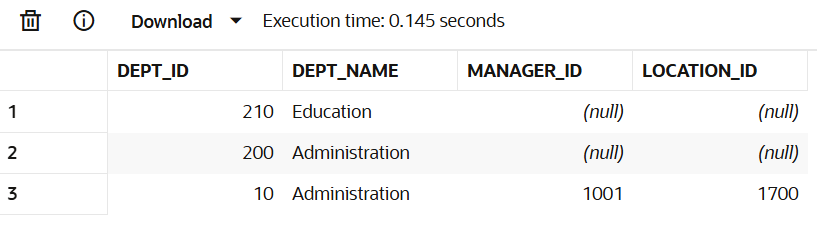


4. Create a nonunique index on the foreign key column (DEPT\_ID) in the EMP table.

SELECT \* FROM DEPARTMENT

WHERE DEPT\_NAME

IN ('Education', 'Administration');



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

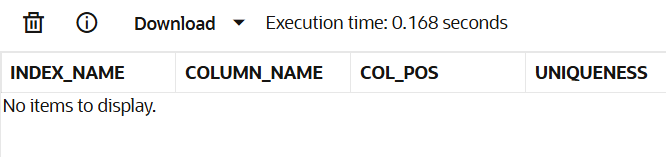
AS col\_pos, ix.uniqueness

FROM user\_indexes ix

JOIN user\_ind\_columns ic

ON ic.index\_name = ix.index\_name

WHERE ic.table\_name = 'EMP';



**EXERCISE-15**

**Controlling User Access**

**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

****

**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**

**.**

**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 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?

To log on to the Oracle Server, a user must be granted the CREATE SESSION privilege. This is a system privilege allowing a user to establish a connection to the database and start a session.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2. What privilege should a user be given to create tables?

To create tables, a user must have the CREATE TABLE privilege, which is also a system privilege enabling the user to create new tables within their schema or database.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. If you create a table, who can pass along privileges to other users on your table?

When you create a table, you as the owner have all privileges on that table, including the ability to grant (pass along) privileges to other users on your table. Only the owner or a user with special privileges can grant such object privileges.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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?

As a DBA creating many users needing the same system privileges, you should create a role. A role is a named group of privileges that can be granted to many users, simplifying privilege management.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. What command do you use to change your password?

To change your Oracle user password, use the command:

ALTER USER your\_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 your\_schema.departments

TO other\_user WITH GRANT OPTION;

7. Query all the rows in your DEPARTMENTS table.

SELECT \* FROM your\_schema.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

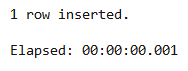
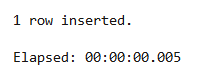
team’s table.

INSERT INTO departments (department\_id, department\_name)

VALUES (500, 'Education');

INSERT INTO departments (department\_id, department\_name)

VALUES (510, 'Human Resources');



9. Query the USER\_TABLES data dictionary to see information about the tables that you own.

SELECT \* FROM other\_team\_schema.department;

10. Revoke the SELECT privilege on your table from the other team.

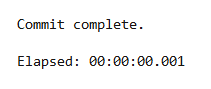
REVOKE SELECT ON your\_schema.departments

FROM other\_user;

11. Remove the row you inserted into the DEPARTMENTS table in step 8 and save the changes.

DELETE FROM departments WHERE department\_id = 500;

COMMIT;



|  |  |
| --- | --- |
| Evaluation Procedure | Marks awarded |
| Practice Evaluation (5) |  |
| Viva(5) |  |
| Total (10) |  |
| Faculty Signature |  |

PL/SQL

**EXERCISE-16**

**PROCEDURES AND FUNCTIONS**

**PROCEDURES**

**DEFINITION**

      A procedure or function is a logically grouped set of SQL and PL/SQL statements that perform a specific task. They are essentially sub-programs. Procedures and functions are made up of,

* Declarative part
* Executable part
* Optional exception handling part

These procedures and functions do not show the errors.

**KEYWORDS AND THEIR PURPOSES**

**REPLACE:** It recreates the procedure if it already exists.

**PROCEDURE:** It is the name of the procedure to be created.

**ARGUMENT:** It is the name of the argument to the procedure. Paranthesis can be omitted if no arguments are present.

**IN:** Specifies that a value for the argument must be specified when calling the procedure ie. used to pass values to a sub-program. This is the default parameter.

**OUT:**  Specifies that the procedure passes a value for this argument back to it’s calling environment after execution ie. used to return values to a caller of the sub-program.

**INOUT:** Specifies that a value for the argument must be specified when calling the procedure and that procedure passes a value for this argument back to it’s calling environment after execution.

**RETURN:** It is the datatype of the function’s return value because every function must return a value, this clause is required.

**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.

**Program 1**

**FACTORIAL OF A NUMBER USING FUNCTION**

CREATE OR REPLACE FUNCTION factorial

(n IN NUMBER) RETURN NUMBER IS

result NUMBER := 1;

BEGIN

IF n < 0 THEN

RETURN NULL;

END IF;

FOR i IN 1..n LOOP

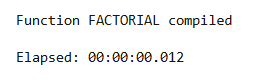
result := result \* i;

END LOOP;

RETURN result;

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 get\_book\_info(

p\_book\_id IN NUMBER,

p\_title OUT VARCHAR2,

p\_author OUT VARCHAR2,

p\_copies IN OUT NUMBER

) IS

BEGIN

SELECT title, author, copies INTO p\_title, p\_author, p\_copies

FROM library\_books

WHERE book\_id = p\_book\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

p\_title := 'NOT FOUND';

p\_author := 'NOT FOUND';

p\_copies := 0;

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_book\_id NUMBER := 1;

v\_title VARCHAR2(100);

v\_author VARCHAR2(100);

v\_copies NUMBER := 0;

BEGIN

get\_book\_info(v\_book\_id, v\_title, v\_author, v\_copies);

DBMS\_OUTPUT.PUT\_LINE('Title: ' || v\_title);

DBMS\_OUTPUT.PUT\_LINE('Author: ' || v\_author);

DBMS\_OUTPUT.PUT\_LINE('Copies available: ' || v\_copies);

END;

/



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

**EXERCISE-17**

**TRIGGER**

**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.

**TO CREATE THE TABLE ‘ITEMPLS’**

SQL> create table itempls (ename varchar2(10), eid number(5), salary number(10));

Table created.

SQL> insert into itempls values('xxx',11,10000);

1 row created.

SQL> insert into itempls values('yyy',12,10500);

1 row created.

SQL> insert into itempls values('zzz',13,15500);

1 row created.

SQL> select \* from itempls;

ENAME            EID    SALARY

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

xxx               11     10000

yyy               12     10500

zzz               13     15500

**TO CREATE A SIMPLE TRIGGER THAT DOES NOT ALLOW INSERT UPDATE AND DELETE OPERATIONS ON THE TABLE**

SQL> create trigger ittrigg before insert or update or delete on itempls for each row

  2  begin

  3  raise\_application\_error(-20010,'You cannot do manipulation');

  4  end;

  5

  6  /

Trigger created.

SQL> insert into itempls values('aaa',14,34000);

insert into itempls values('aaa',14,34000)

            \*

ERROR at line 1:

ORA-20010: You cannot do manipulation

ORA-06512: at "STUDENT.ITTRIGG", line 2

ORA-04088: error during execution of trigger 'STUDENT.ITTRIGG'

SQL> delete from itempls where ename='xxx';

delete from itempls where ename='xxx'

            \*

ERROR at line 1:

ORA-20010: You cannot do manipulation

ORA-06512: at "STUDENT.ITTRIGG", line 2

ORA-04088: error during execution of trigger 'STUDENT.ITTRIGG'

SQL> update itempls set eid=15 where ename='yyy';

update itempls set eid=15 where ename='yyy'

       \*

ERROR at line 1:

ORA-20010: You cannot do manipulation

ORA-06512: at "STUDENT.ITTRIGG", line 2

ORA-04088: error during execution of trigger 'STUDENT.ITTRIGG'

**TO DROP THE CREATED TRIGGER**

SQL> drop trigger ittrigg;

Trigger dropped.

**TO CREATE A TRIGGER THAT RAISES AN USER DEFINED ERROR MESSAGE AND DOES NOT ALLOW UPDATION AND INSERTION**

SQL> create trigger ittriggs before insert or update of salary on itempls for each row

  2  declare

  3  triggsal itempls.salary%type;

  4  begin

  5  select salary into triggsal from itempls where eid=12;

  6  if(:new.salary>triggsal or :new.salary<triggsal) then

  7  raise\_application\_error(-20100,'Salary has not been changed');

  8  end if;

  9  end;

 10  /

Trigger created.

SQL> insert into itempls values ('bbb',16,45000);

insert into itempls values ('bbb',16,45000)

            \*

ERROR at line 1:

ORA-04098: trigger 'STUDENT.ITTRIGGS' is invalid and failed re-validation

SQL> update itempls set eid=18 where ename='zzz';

update itempls set eid=18 where ename='zzz'

       \*

ERROR at line 1:

ORA-04298: trigger 'STUDENT.ITTRIGGS' is invalid and failed re-validation

         Cursor for loop

         Explicit cursor

         Implicit cursor

**TO CREATE THE TABLE ‘SSEMPP’**

SQL> create table ssempp( eid number(10), ename varchar2(20), job varchar2(20), sal number (10),dnonumber(5));

Table created.

SQL> insert into ssempp values(1,'nala','lecturer',34000,11);

1 row created.

SQL> insert into ssempp values(2,'kala',' seniorlecturer',20000,12);

1 row created.

SQL> insert into ssempp values(5,'ajay','lecturer',30000,11);

1 row created.

SQL> insert into ssempp values(6,'vijay','lecturer',18000,11);

1 row created.

SQL> insert into ssempp values(3,'nila','professor',60000,12);

1 row created.

SQL> select \* from ssempp;

    EID    ENAME                JOB                        SAL       DNO

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

        1       nala                   lecturer                  34000        11

        2       kala                  seniorlecturer         20000         12

        5       ajay                  lecturer                  30000         11

        6      vijay                 lecturer                   18000         11

        3      nila                   professor                60000         12

**EXTRA PROGRAMS**

**TO WRITE A PL/SQL BLOCK TO DISPLAY THE EMPOYEE ID AND EMPLOYEE NAME USING CURSOR FOR LOOP**

SQL> set serveroutput on;

SQL> declare

  2  begin

  3  for emy in (select eid,ename from ssempp)

  4  loop

  5  dbms\_output.put\_line('Employee id and employee name  are '|| emy.eid ‘and’|| emy.ename);

  6  end loop;

  7  end;

  8  /

Employee id and employee name are 1 and nala

Employee id and employee name are 2 and kala

Employee id and employee name are 5 and ajay

Employee id and employee name are 6 and vijay

Employee id and employee name are 3 and nila

PL/SQL procedure successfully completed.

**TO WRITE A PL/SQL BLOCK TO UPDATE THE SALARY OF ALL EMPLOYEES WHERE DEPARTMENT NO IS 11 BY 5000 USING CURSOR FOR LOOP AND TO DISPLAY THE UPDATED TABLE**

SQL> set serveroutput on;

SQL> declare

  2  cursor cem is select eid,ename,sal,dno from ssempp where dno=11;

  3  begin

  4  --open cem;

  5  for rem in cem

  6  loop

  7  update ssempp set sal=rem.sal+5000 where eid=rem.eid;

  8  end loop;

  9  --close cem;

 10  end;

 11  /

PL/SQL procedure successfully completed.

SQL> select \* from ssempp;

      EID      ENAME                JOB                 SAL       DNO

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

        1       nala                       lecturer             39000        11

        2       kala                  seniorlecturer        20000        12

        5       ajay                      lecturer             35000        11

        6     vijay                       lecturer             23000        11

        3     nila                       professor            60000        12

**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 cenl is select eid,sal from ssempp where dno=11;

  3  ecode ssempp.eid%type;

  4  esal empp.sal%type;

  5  begin

  6  open cenl;

  7  loop

  8  fetch cenl into ecode,esal;

  9  exit when cenl%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

**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 numbe

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 SCA LCULATE 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 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 prevent\_parent\_deletion

BEFORE DELETE ON department

FOR EACH ROW

DECLARE

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM employees

WHERE department\_id = :OLD.dept\_id;

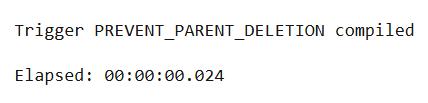
IF v\_count > 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Cannot delete department with associated employees.');

END IF;

END;

/



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 prevent\_duplicates

BEFORE INSERT ON products

FOR EACH ROW

DECLARE

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count FROM products WHERE product\_name = :NEW.product\_name;

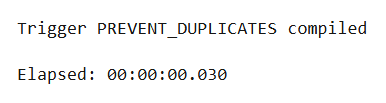
IF v\_count > 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Product name already exists.');

END IF;

END;

/



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 check\_order\_amount

BEFORE INSERT ON orders

FOR EACH ROW

DECLARE

total\_amount NUMBER;

max\_threshold NUMBER := 10000;

BEGIN

SELECT NVL(SUM(order\_amount),0) INTO total\_amount FROM orders WHERE customer\_id = :NEW.customer\_id;

IF total\_amount + :order\_amount > max\_threshold THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Total order amount exceeds the threshold.');

END IF;

END;

/



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 SEQUENCE seq\_salary\_audit START WITH 1 INCREMENT BY 1;

CREATE OR REPLACE TRIGGER salary\_change\_audit

AFTER UPDATE ON employees

FOR EACH ROW

WHEN (NEW.salary <> OLD.salary)

DECLARE

v\_audit\_id NUMBER;

BEGIN

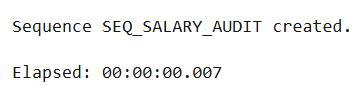
SELECT seq\_salary\_audit.NEXTVAL INTO v\_audit\_id FROM DUAL;

INSERT INTO salary\_audit (audit\_id, employee\_id, old\_salary, new\_salary, change\_date)

VALUES (v\_audit\_id, :OLD.employee\_id, :OLD.salary, :NEW.salary, SYSTIMESTAMP);

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 SEQUENCE Audit\_Log\_Seq START WITH 1 INCREMENT BY 1;

CREATE OR REPLACE TRIGGER Employee\_Audit\_Trigger

AFTER INSERT OR UPDATE OR DELETE ON Employees

FOR EACH ROW

DECLARE

v\_activity\_type VARCHAR2(20);

BEGIN

IF INSERTING THEN

v\_activity\_type := 'INSERT';

ELSIF UPDATING THEN

v\_activity\_type := 'UPDATE';

ELSIF DELETING THEN

v\_activity\_type := 'DELETE';

END IF;

INSERT INTO Audit\_Log (log\_id, table\_name, activity\_type, activity\_date, user\_id)

VALUES (Audit\_Log\_Seq.NEXTVAL, 'Employees', v\_activity\_type, SYSTIMESTAMP, USER);

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

BEFORE INSERT ON Sales

FOR EACH ROW

BEGIN

IF :NEW.running\_total IS NULL THEN

SELECT NVL(MAX(running\_total), 0) + :NEW.amount INTO :NEW.running\_total FROM Sales;

ELSE

:NEW.running\_total := :NEW.running\_total + :NEW.amount;

END IF;

END;

/



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 OR REPLACE TRIGGER Validate\_Order\_Availability

BEFORE INSERT ON Orders

FOR EACH ROW

DECLARE

v\_current\_stock NUMBER;

v\_pending\_orders NUMBER;

BEGIN

SELECT stock\_quantity INTO v\_current\_stock FROM Products WHERE product\_id = :NEW.product\_id;

SELECT NVL(SUM(order\_quantity), 0) INTO v\_pending\_orders FROM Orders WHERE product\_id = :NEW.product\_id;

IF v\_current\_stock - v\_pending\_orders - :NEW.order\_quantity < 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient stock for the order');

END IF;

END;

/



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

**MONGO DB**

**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: { $regex: /^Wil/ } }

]

},

{ restaurant\_id: 1, name: 1, borough: 1, cuisine: 1, \_id: 0 }

);



**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")

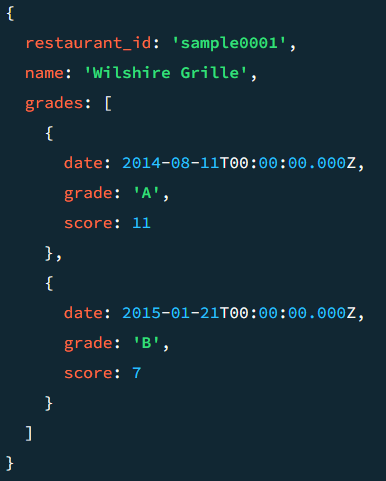
}

}

},

{ restaurant\_id: 1, name: 1, grades: 1, \_id: 0 }

);



**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",

score: 9,

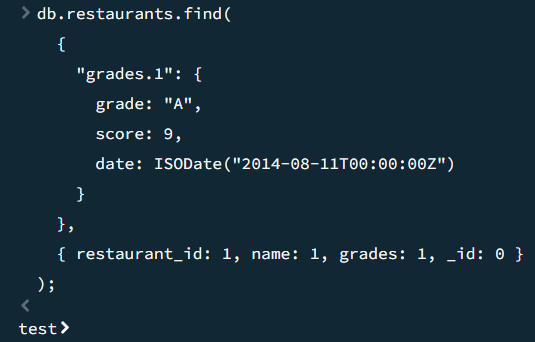
date: ISODate("2014-08-11T00:00:00Z")

}

},

{ restaurant\_id: 1, name: 1, grades: 1, \_id: 0 }

);



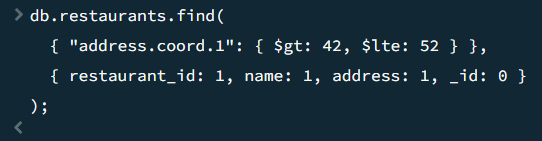
**4. Write a MongoDB query to find the restaurant Id, name, address and geographical location for those restaurants where 2nd element of coord array contains a value which is more than 42 and upto 52..**

db.restaurants.find(

{ "address.coord.1": { $gt: 42, $lte: 52 } },

{ restaurant\_id: 1, name: 1, address: 1, \_id: 0 }

);



**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.**

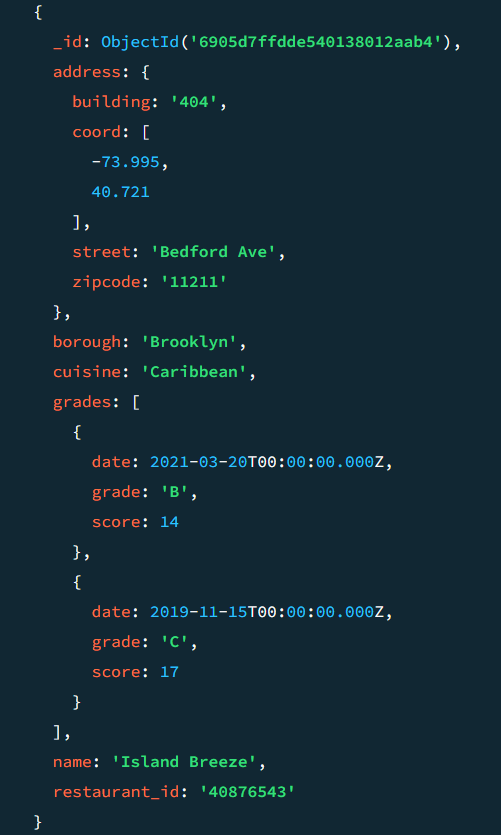
db.restaurants.find().sort({ name: -1 });



**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: false } });



**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.**

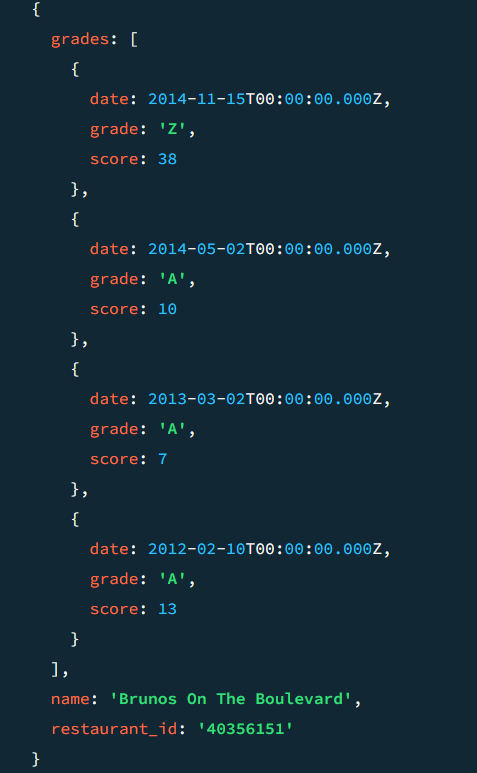
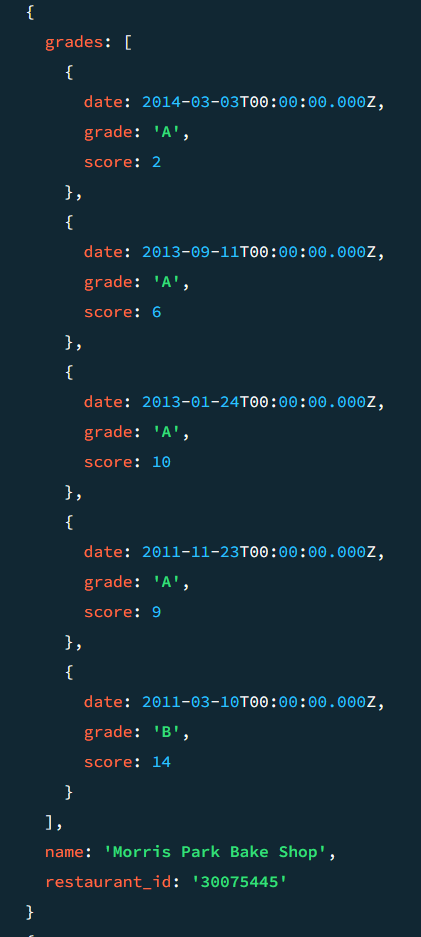
db.restaurants.find(

{ "grades.score": { $mod: [7, 0] } },

{ restaurant\_id: 1,

name: 1, grades: 1, \_id: 0 }

);



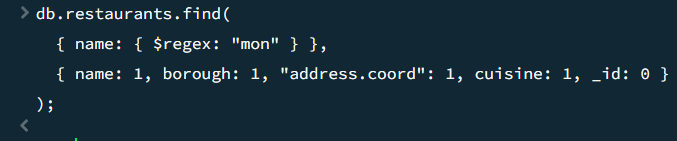
**11. Write a MongoDB query to find the restaurant name, borough, longitude and attitude and cuisine for those restaurants which contains 'mon' as three letters somewhere in its name.**

db.restaurants.find(

{ name: { $regex: "mon" } },

{ name: 1, borough: 1, "address.coord": 1, cuisine: 1, \_id: 0 }

);



**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: { $regex: "^Mad" } },

{ name: 1, borough: 1, "address.coord": 1, cuisine: 1, \_id: 0 }

);



**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 } }

);

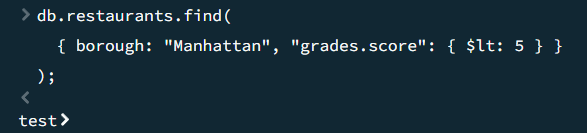


**14. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5 and that are located in the borough of Manhattan.**

db.restaurants.find(

{ borough: "Manhattan", "grades.score": { $lt: 5 } }

);



**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(

{ borough: { $in: ["Manhattan", "Brooklyn"] }, "grades.score": { $lt: 5 } }

);



**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(

{

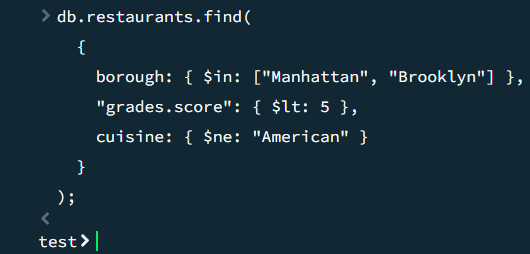
borough: { $in: ["Manhattan", "Brooklyn"] },

"grades.score": { $lt: 5 },

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: {

$all: [

{ $elemMatch: { score: 2 } },

{ $elemMatch: { score: 6 } }

]

}

}

);



**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(

{

borough: "Manhattan",

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(

{

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

grades: {

$all: [

{ $elemMatch: { score: 2 } },

{ $elemMatch: { score: 6 } }

]

}

}

);



**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.  
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(

{

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

grades: {

$all: [

{ $elemMatch: { score: 2 } },

{ $elemMatch: { score: 6 } }

]

},

cuisine: { $ne: "American" }

}

);



db.restaurants.find({

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

cuisine: { $ne: "American" },

grades: {

$all: [

{ $elemMatch: { score: 2 } },

{ $elemMatch: { score: 6 } }

]

}

});



**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(

{

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

grades: {

$all: [

{ $elemMatch: { score: 2 } },

{ $elemMatch: { score: 6 } }

]

},

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] } }

}

}

);



|  |  |
| --- | --- |
| **Evaluation Procedure** | **Marks awarded** |
| **MONGODB Procedure(5)** |  |
| **Program/Execution (5)** |  |
| **Viva(5)** |  |
| **Total (15)** |  |
| **Faculty Signature** |  |