

Creating and Managing Tables

EX_NO:1

DATE:17.2.24

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

QUERY:

```
Create table dept(id number(7),name varchar2(25));
```

OUTPUT:

The screenshot shows the Oracle SQL developer interface. In the top navigation bar, 'Language' is set to 'SQL'. Below it, there are buttons for 'Save' and 'Run'. The main area contains a SQL command: '1 create table dept(id number(7),name varchar2(25));'. At the bottom, the results tab is selected, showing the message 'Table created.' and a execution time of '0.02 seconds'.

RESULT:

The query is executed successfully.

2. 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

QUERY:

Create table emp(id number(7),Last_Name varchar2(25),First_Name varchar2(25),Dept_id number(7));

OUTPUT:



The screenshot shows a SQL query editor interface. The query entered is:

```
1 Create table emp(id number(7),Last_name varchar2(25),First_name varchar2(25),Dept_id number(7));
```

The results pane shows the message "Table created." and a execution time of "0.02 seconds".

RESULT:

The query is executed successfully.

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

QUERY:

Alter table emp modify(Last_Name varchar2(50));

OUTPUT:



The screenshot shows a SQL query editor interface. The query entered is:

```
1 alter table emp modify(last_name varchar2(50));
```

The results pane shows the message "Table altered." and a execution time of "0.06 seconds".

RESULT: The query is executed successfully

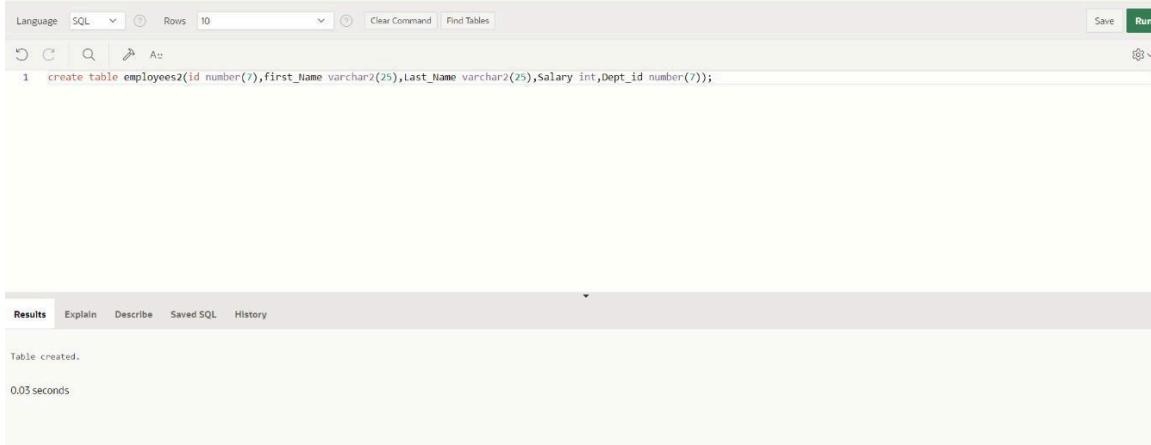
4. Create the EMPLOYEES2 table based on the structure of EMPLOYEES table. Include Only the Employee_id, First_name, Last_name, Salary and Dept_id coloumns. Name the columns Id, First_name, Last_name, salary and Dept_id respectively.

QUERY:

```
Create table employees2(id number(7),first_name varchar2(25),Last_name varchar2(25),Salary int,Dept_id number(7));
```

OUTPUT:

RESULT:



The screenshot shows a SQL development environment with the following details:

- Toolbar: Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Query Editor: A single line of SQL code: `create table employees2(id number(7),first_name varchar2(25),Last_name varchar2(25),Salary int,Dept_id number(7));`.
- Results Tab: The tab is selected, showing the output of the query.
- Output: The message "Table created." is displayed, along with a timestamp "0.03 seconds".

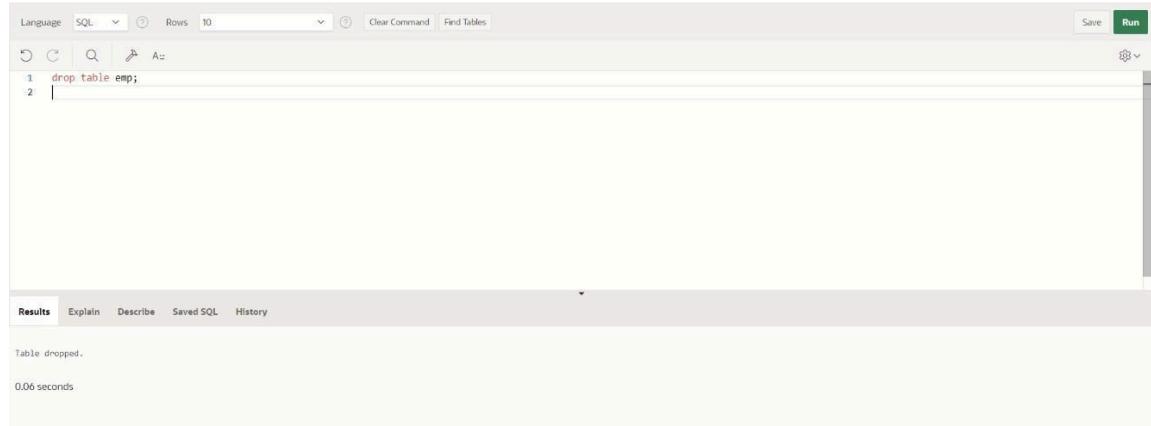
The query is executed successfully.

4. Drop the EMP table.

QUERY:

```
Drop table emp;
```

OUTPUT:



The screenshot shows a SQL development environment with the following details:

- Toolbar: Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Query Editor: Two lines of SQL code: `drop table emp;` and an empty line below it.
- Results Tab: The tab is selected, showing the output of the query.
- Output: The message "Table dropped." is displayed, along with a timestamp "0.06 seconds".

RESULT:

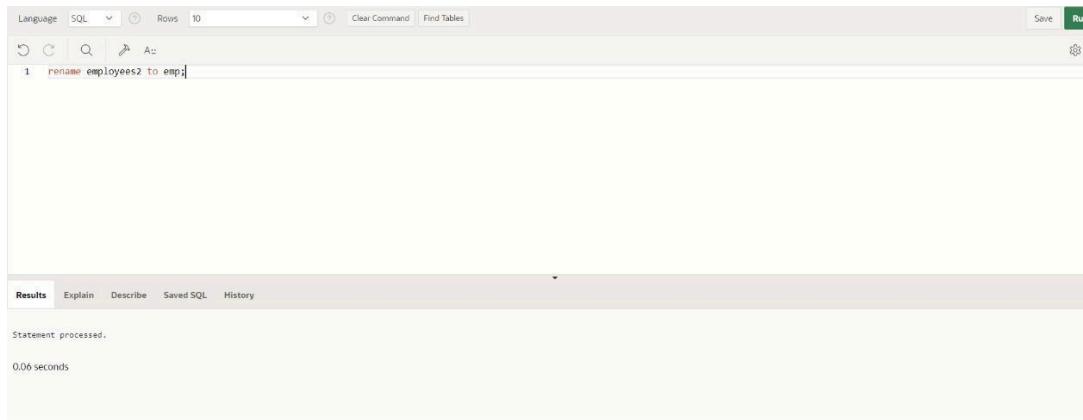
The query is executed successfully.

5. Rename the EMPLOYEES2 table as EMP.

QUERY:

Rename employees2 to emp;

OUTPUT:



A screenshot of a SQL query editor interface. The top bar includes 'Language' (set to 'SQL'), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. The main area shows a single line of SQL code: '1. rename employees2 to emp;'. Below the code, the status bar displays 'Statement processed.' and '0.06 seconds'.

RESULT:

The query is executed successfully.

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

QUERY:

comment on table dept is 'Department info';
comment on table emp is Employee info';

OUTPUT:



A screenshot of a SQL query editor interface, identical to the one above. The top bar includes 'Language' (set to 'SQL'), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. The main area shows a single line of SQL code: '1. comment on table dept is 'Department info'';. Below the code, the status bar displays 'Statement processed.' and '0.03 seconds'.

RESULT:

The query is executed successfully.

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

QUERY:

```
Alter table emp drop column first_name;
```

OUTPUT:

The screenshot shows a MySQL command-line interface. The command entered is: `alter table emp drop column first_name;`. The output shows the message: `Table altered.` and a time stamp: `0.05 seconds`.

RESULT:

The query is executed successfully.

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

MANIPULATING DATA

EX_NO:2

DATE:24.2.24

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)

QUERY:

Create table my_employee(id number(4),Last_Name varchar2(25),First_Name varchar2(25),Userid varchar2(25),Salary number(9,2));

OUTPUT:

The screenshot shows a SQL query editor interface. The query entered is:

```
create table my_employee(id number(4),Last_Name varchar2(25),First_Name varchar2(25),Userid varchar2(25),Salary number(9,2))|
```

The results pane shows the output:

Table created.
0.03 seconds

RESULT:

The query is executed successfully.

2. 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

QUERY:

```
Insert into my_employee values(1,'Patel','Ralph','rpatel',895);
```

```
Insert into my_employee values(1,'Dancs','Betty','bdancs',860);
```

OUTPUT:

The screenshot shows a SQL query editor interface. The SQL tab contains the following code:

```
1 insert into my_employee (id,last_name,First_name,Userid,salary)
2 values
3 |
4 (2,'Dancs','Betty','bdancs',860);
```

The Results tab shows the output of the query:

```
1 row(s) inserted.
```

0.01 seconds

RESULT:

The query is executed successfully.

3. Display the table with values.

QUERY:

```
Select*from my_employee;
```

OUTPUT:

The screenshot shows a SQL query editor interface. The SQL tab contains the following code:

```
1 select* from my_employee;
```

The Results tab shows the output of the query, displaying the contents of the my_employee table:

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860

2 rows returned in 0.02 seconds

RESULT:

The query is executed successfully.

4. 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.

QUERY:

```
Insert into my_employee values(1,'Patel','Ralph','rpatel',1100);
Insert into my_employee values(1,'Dancs','Betty','bdancs',750);
```

OUTPUT:

RESULT:

The screenshot shows a MySQL Workbench interface. The SQL tab contains the following command:

```
1 Insert into my_employee values(4,'Newman','Chad','Cnewman',750);
```

The Results tab shows the output:

```
1 row(s) inserted.
```

Execution time: 0.01 seconds.

The query is executed successfully.

5. Make the data additions permanent.

QUERY:

```
Select*from my_employee;
```

OUTPUT:

The screenshot shows a MySQL Workbench interface. The SQL tab contains the following command:

```
1 select*from my_employee;
```

The Results tab displays the data from the my_employee table:

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	Cnewman	1000

Execution time: 0.04 seconds.

RESULT:

The query is executed successfully.

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

QUERY:

```
Update my_employee set last_name='Drexler' where id=3;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Run button
- SQL command: `update my_employee set last_name='Drexler' where id=3;`
- Results tab selected
- Output:
 - 1 row(s) updated.
 - 0.01 seconds

RESULT:

The query is executed successfully.

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

QUERY:

```
update my_employee set salary=1000 where salary<900;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Run button
- SQL command: `update my_employee set salary=1000 where salary<900;`
- Results tab selected
- Output:
 - 3 row(s) updated.
 - 0.01 seconds

RESULT:

The query is executed successfully.

7. Delete Betty dancs from MY_EMPLOYEE table.

QUERY:

```
delete from my_employee where last_name='Dancs';
```

OUTPUT:

The screenshot shows a SQL query editor interface. The top bar includes 'Language' set to 'SQL', 'Rows' set to '10', 'Clear Command', 'Find Tables', 'Save', and a 'Run' button. Below the toolbar is a toolbar with icons for refresh, copy, search, and other functions. The main area contains the SQL command: '1 delete from my_employee where last_name='Dancs';'. At the bottom, the results section shows '1 row(s) deleted.' and a execution time of '0.01seconds'.

RESULT:

The query is executed successfully.

8. Empty the fourth row of the emp table.

QUERY:

```
Delete from emp where id=4;
```

OUTPUT:

The screenshot shows a SQL query editor interface. The top bar includes 'Language' set to 'SQL', 'Rows' set to '10', 'Clear Command', 'Find Tables', 'Save', and a 'Run' button. Below the toolbar is a toolbar with icons for refresh, copy, search, and other functions. The main area contains the SQL command: '1 delete from emp where id=4;'. At the bottom, the results section shows '0 row(s) deleted.' and a execution time of '0.05 seconds'.

RESULT:

The query is executed successfully.

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

INCLUDING CONSTRAINTS

EX_NO:3

DATE:28.2.24

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.

QUERY:

```
alter table emp add constraint my_emp_id_pk primary key(id);
```

OUTPUT:

The screenshot shows a MySQL Workbench interface. The SQL tab contains the command: `alter table emp add constraint my_emp_id_pk primary key(id);`. The Results tab shows the output: `Table altered.` and `0.07 seconds`.

RESULT:

The query is executed successfully.

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

QUERY:

```
Alter table emp add constraint my_dept_id_pk primary key(id);
```

OUTPUT:

The screenshot shows a MySQL Workbench interface. The SQL tab contains the command: `alter table dept add constraint my_dept_id_pk primary key(id);`. The Results tab shows the output: `Table altered.` and `0.07 seconds`.

RESULT:

The query is executed successfully.

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

QUERY:

```
alter table emp add constraint my_emp_dept_id_fk foreign key(dept_id) references emp(id);
```

OUTPUT:

The screenshot shows a SQL query editor interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. Below the toolbar is a toolbar with icons for refresh, search, and other functions. The main area contains the SQL command: '1 alter table emp add constraint my_emp_dept_id_fk foreign key(dept_id) references emp(id);'. The results section at the bottom displays the output: 'Table altered.' and '0.06 seconds'.

RESULT:

The query is executed successfully.

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

QUERY:

```
Alter table emp add constraint number(2,2) check(commission);
```

OUTPUT:

The screenshot shows a SQL query editor interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. Below the toolbar is a toolbar with icons for refresh, search, and other functions. The main area contains the SQL command: '1 alter table emp add commission number(2,2) check(commission>0);'. The results section at the bottom displays the output: 'Table altered.' and '0.06 seconds'.

RESULT:

The query is executed successfully.

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

Writing Basic SQL SELECT Statements

EX_NO:4

DATE:2.3.24

1. The following statement executes successfully.

Identify the Errors

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

QUERY:

```
Select employee_id, last_name, sal*12 as "ANNUAL SALARY" from employees;
```

OUTPUT

The screenshot shows a SQL query execution interface. The query is:

```
1. SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, "SALARY"=12"ANNUAL_SALARY" FROM EMPLOYEE
```

The results table has the following data:

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	ANNUAL_SALARY
157	BHARATH	BEST	180000
541	PATEL	SUNIL	144000
143	KUMAR	BHARATH	171600

3 rows returned in 0.01 seconds Download

RESULT:

The query is executed successfully.

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

QUERY:

```
Desc dept;
```

OUTPUT:

Language SQL Rows 10 Clear Command Find Tables Save Run

1 desc dept;

Results Explain **Describe** Saved SQL History

Object Type TABLE Object DEPT

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
DEPT	ID	NUMBER	-	7	0	1	-	-	-
	NAME	VARCHAR2	25	-	-	-	✓	-	-

RESULT:

The query is executed successfully.

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

QUERY:

```
Select employee_id, last_name, job_id, hire_date from employees;
```

OUTPUT:

The screenshot shows a SQL query execution interface. The query entered is: `select employee_id, last_name, job_id, hire_date from employees;`. The results pane displays a table with four columns: EMPLOYEE_ID, LAST_NAME, JOB_ID, and HIRE_DATE. Two rows are returned, corresponding to employees with IDs 113 and 114. The data is as follows:

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE
113	popp	ac_account	03/05/2024
114	raphealy	ac_account	02/03/1999

2 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

4. Provide an alias STARTDATE for the hire date.

QUERY:

```
select hire_date as "STARTDATE" from employees;
```

OUTPUT:

The screenshot shows a SQL query execution interface. The query entered is: `select hire_date as "STARTDATE" from employees;`. The results pane displays a table with one column named STARTDATE. Two rows are returned, showing the values 03/05/2024 and 02/03/1999.

STARTDATE
03/05/2024
02/03/1999

2 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select distinct job_id from employees;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command | Find Tables | Run
- Query: `select distinct job_id from employees;`
- Results tab selected.
- Output table header: **JOB_ID**
- Data row: ac_account
- Message: 1 rows returned in 0.00 seconds | Download

RESULT:

The query is executed successfully.

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

QUERY:

```
Select last_name||', '||job_id as "EMPLOYEE_AND_TITLE" from employees;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command | Find Tables | Run
- Query: `select last_name||', '||job_id as "EMPLOYEE_AND_TITLE" from employees;`
- Results tab selected.
- Output table header: **EMPLOYEE_AND_TITLE**
- Data rows:
 - popp, ac_account
 - raphealy, ac_account
- Message: 2 rows returned in 0.00 seconds | Download

RESULT:

The query is executed successfully.

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

QUERY:

Select

```
employee_id||'|'||first_name||'|'||last_name||'|'||email||'|'||phone_no||'|'||hire_date||'|'||job_id||'|'||salary||'|'||commission_pct||'|'||manager_id||'|'||department_id as THE_OUTPUT from employees;
```

OUTPUT:

The screenshot shows a SQL query execution interface. The query is:

```
select employee_id||'|'||first_name||'|'||last_name||'|'||email||'|'||phone_no||'|'||hire_date||'|'||job_id||'|'||salary||'|'||commission_pct||'|'||manager_id||'|'||department_id as THE_OUTPUT
```

The results table has one column named "THE_OUTPUT". The data returned is:

THE_OUTPUT
113,louis,popp,popp,5151244567,03/05/2024,ac_account,6900,,205,100
114,den,rapheal,drapheal,5151274561,02/03/1999,ac_account,11000,,100,30

2 rows returned in 0.03 seconds [Download](#)

RESULT:

The query is executed successfully.

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

RESTRICTING AND SORTING DATA

EX_NO:5

DATE: 06.03.2024

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

QUERY:

Select last_name from employees where salary>12000;

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for 'Language' (set to 'SQL'), 'Rows' (set to 10), and buttons for 'Clear Command' and 'Find Tables'. On the right side, there are 'Save' and 'Run' buttons. Below the toolbar, there are icons for search, refresh, and other database operations. The main area contains a code editor with the following SQL query:

```
1 select last_name,salary from employees where salary>12000;
```

Below the code editor, there is a navigation bar with tabs: 'Results' (which is selected), 'Explain', 'Describe', 'Saved SQL', and 'History'. Underneath the navigation bar, the text 'no data found' is displayed, indicating that the query did not return any results.

RESULT:

The query is executed successfully.

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

QUERY:

Select last_name,department_id from employees where employee_id=176;

OUTPUT:

The screenshot shows a SQL query editor interface, identical to the one above. It has the same toolbar, code editor, and navigation bar. The code editor contains the following SQL query:

```
1 select last_name,department_id from employees where employee_id=176;
```

Under the 'Results' tab, the text 'no data found' is displayed, indicating that the query did not return any results.

RESULT:

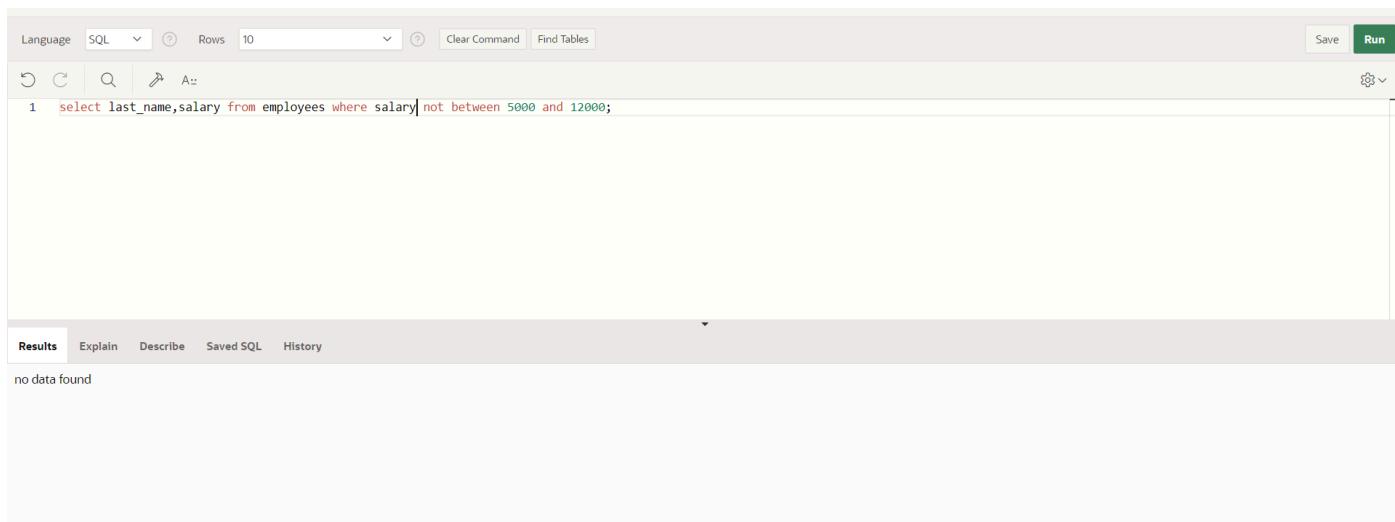
The query is executed successfully.

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

QUERY:

```
select last_name,salary from employees where salary not between 5000 and 12000;
```

OUTPUT:



A screenshot of a SQL query editor interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. Below the toolbar, there are icons for refresh, search, and other database operations. The main area shows a single line of SQL code: '1 select last_name,salary from employees where salary not between 5000 and 12000;'. Below the code, the 'Results' tab is selected, showing the message 'no data found'.

RESULT:

The query is executed successfully.

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

QUERY:

```
Select last_name,job_id,hire_date from employees where hire_date between 'February,20,1998' and 'May,1,1998';
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query is:

```
1 select last_name,job_id,hire_date from employees where hire_date between 'February,20,1998' and 'may,1,1998';
```

The results table has three columns: LAST_NAME, JOB_ID, and HIRE_DATE. One row is returned:

LAST_NAME	JOB_ID	HIRE_DATE
Mohan	ac_account	02/22/1998

1 rows returned in 0.03 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,department_id from employees where department_id in(20,50) order by last_name;
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query is:

```
1 select last_name,department_id from employees where department_id in(20,50) order by last_name;
```

The results table has two columns: LAST_NAME and DEPARTMENT_ID. One row is returned:

LAST_NAME	DEPARTMENT_ID
raphealy	20

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name as "EMPLOYEE",salary as "MONTHLY SALARY" from employees where (salary between 5000 and 12000) and (department_id in(20,50)) order by last_name asc;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Save
- Run

The query entered is:

```
1 select last_name as "EMPLOYEE",salary as "MONTHLY SALARY" from employees where (salary between 5000 and 12000) and (department_id in(20,50)) order by last_name asc;
```

The results section shows:

EMPLOYEE	MONTHLY SALARY
raphealy	11000

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,hire_date from employees where hire_date like '1994';
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Save
- Run

The query entered is:

```
1 select last_name,hire_date from employees where hire_date like '1994';
```

The results section shows:

no data found

RESULT: The query is executed successfully.

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

QUERY:

```
select last_name,job_id from employees where manager_id is null;
```

OUTPUT:

A screenshot of a SQL query editor interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and a 'Run' button. Below the toolbar is a toolbar with icons for refresh, search, and other functions. The main area contains the following SQL command:

```
1 select last_name,job_id from employees where manager_id is null;
```

The results section shows the message "no data found".

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,salary,commission_pct from employees where commission_pct is not null order by salary desc;
```

OUTPUT:

A screenshot of a SQL query editor interface, identical to the one above, showing the same query and results. The results section again shows "no data found".

RESULT:

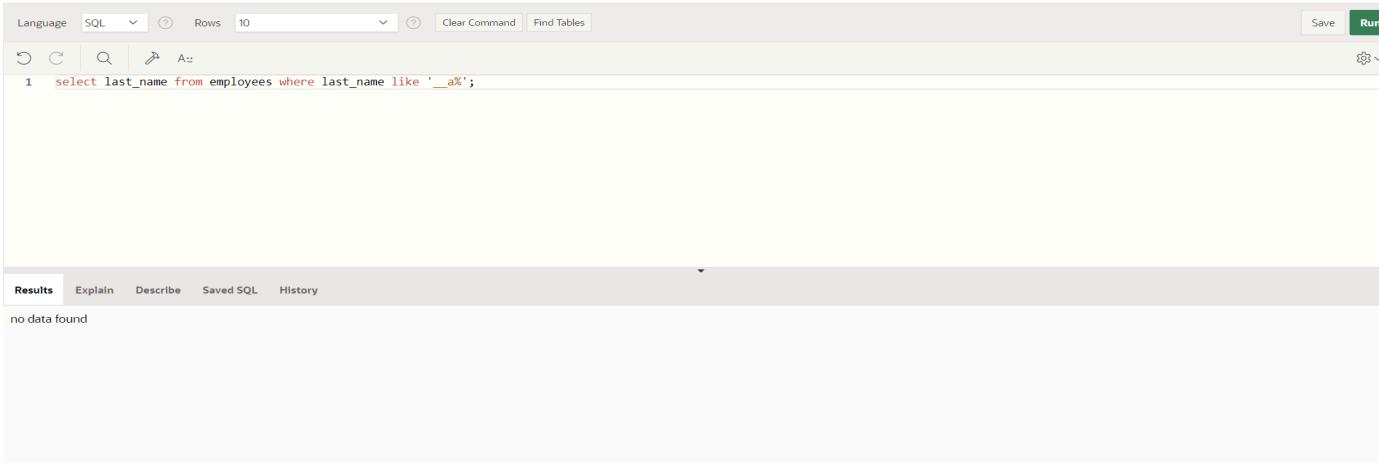
The query is executed successfully.

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

QUERY:

```
select last_name from employees where last_name like '__a%';
```

OUTPUT:



The screenshot shows a SQL query editor interface. At the top, there are tabs for 'Language' (set to 'SQL'), 'Rows' (set to 10), and buttons for 'Clear Command' and 'Find Tables'. On the right side, there are 'Save' and 'Run' buttons. The main area contains a single line of SQL code: '1 select last_name from employees where last_name like '__a%';'. Below this, a results pane titled 'Results' shows the message 'no data found'.

RESULT:

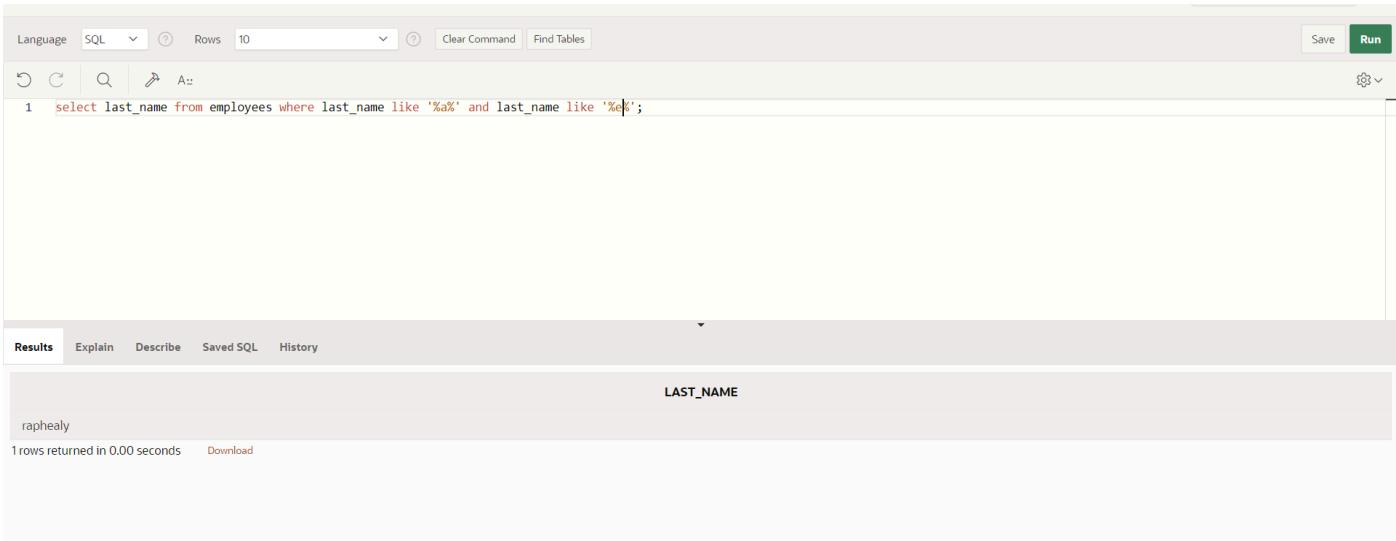
The query is executed successfully.

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

QUERY:

```
select last_name from employees where last_name like '%a%' and last_name like '%e%';
```

OUTPUT:



The screenshot shows a SQL query editor interface. At the top, there are tabs for 'Language' (set to 'SQL'), 'Rows' (set to 10), and buttons for 'Clear Command' and 'Find Tables'. On the right side, there are 'Save' and 'Run' buttons. The main area contains a single line of SQL code: '1 select last_name from employees where last_name like '%a%' and last_name like '%e%';'. Below this, a results pane titled 'Results' shows a table with one row. The table has a single column labeled 'LAST_NAME' and contains the value 'raphealy'. At the bottom of the results pane, it says '1 rows returned in 0.00 seconds' and has a 'Download' link.

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,job_id,salary from employees where job_id in ('sales representative','stock clerk') and salary not in(2500,3500,7000);
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the input field, there are icons for Refresh, Undo, Redo, Search, and Paste. The input field contains the SQL query: `select last_name,job_id,salary from employees where job_id in ('sales representative','stock clerk') and salary not in(2500,3500,7000);`. The results section below shows the message "no data found".

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,salary,commission_pct from employees where commission_pct=0.2;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the input field, there are icons for Refresh, Undo, Redo, Search, and Paste. The input field contains the SQL query: `select last_name,salary,commission_pct from employees where commission_pct=0.2;`. The results section below shows the message "no data found".

RESULT:

The query is executed successfully.

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

SINGLE ROW FUNCTIONS

EX_NO:6

DATE:13.03.2024

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

QUERY:

```
select sysdate from dual;
```

OUTPUT:

The screenshot shows a SQL query editor interface. The query entered is "select sysdate from dual;". The results pane displays a single row with the column name "SYSDATE" and the value "03/10/2024". Below the results, it says "1 rows returned in 0.01 seconds" and there is a "Download" link.

SYSDATE
03/10/2024

RESULT:

The query is executed successfully.

2. 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.

QUERY:

```
select employee_id, last_name, salary, salary+(15.5/100*salary) "new_salary" from employees;
```

OUTPUT:

Language SQL Rows 10 Clear Command Find Tables Save Run

```
1 select employee_id, last_name, salary, salary+(15.5/100*salary) "new_salary" from employees;
2
```

Results Explain Describe Saved SQL History

EMPLOYEE_ID	LAST_NAME	SALARY	new_salary
113	popp	6900	7969.5
114	raphealy	11000	12705
2	Mohan	4000	4620

3 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

3. Modify your query lab_03_02.sql to add a column that subtracts the old salary from the new salary. Label the column Increase.

QUERY:

```
select employee_id, last_name, salary, salary+(15.5/100*salary) "new_salary", new_salary-salary as "Increase"
from employees;
```

OUTPUT:

Language SQL Rows 10 Clear Command Find Tables Save Run

```
1 select employee_id, last_name, salary, salary+(15.5/100*salary) "new_salary", (salary+(15.5/100*salary))-salary as "Increase" from employees;
2
```

Results Explain Describe Saved SQL History

EMPLOYEE_ID	LAST_NAME	SALARY	new_salary	Increase
113	popp	6900	7969.5	1069.5
114	raphealy	11000	12705	1705
2	Mohan	4000	4620	620

3 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

4. 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.

QUERY:

```
select initcap(last_name),length(last_name) as "Length_of_last_name" from employees where last_name like 'J%' or last_name like 'A%' or last_name like 'M%' order by last_name asc;
```

OUTPUT:

The screenshot shows a SQL query execution interface. The query is:

```
1 select initcap(last_name),length(last_name) as"Length_of_last_name" from employees where last_name like 'J%' or last_name like 'A%' or last_name like 'M%' order by last_name asc;
```

The results table has two columns: INITCAP(LAST_NAME) and Length_of_last_name. There is one row for 'Mohan', which has a length of 5.

INITCAP(LAST_NAME)	Length_of_last_name
Mohan	5

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

5. 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.

QUERY:

```
select initcap(last_name) "Name",length(last_name) "Length of last name" from employees where last_name like 'H%' order by last_name;
```

OUTPUT:

The screenshot shows a SQL query interface with the following details:

- Language: SQL
- Rows: 10
- Clear Command | Find Tables
- Save | Run
- Results tab selected
- Query: `select initcap(last_name) "Name", length(last_name) "Length of last name" from employees where last_name like 'H%' order by last_name;`
- Output table:

Name	Length of last name
Hari	4

- 1 rows returned in 0.03 seconds
- Download link

RESULT:

The query is executed successfully.

6. 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.

QUERY:

```
select last_name, round((sysdate-hire_date)/30,0) as "MONTHS_WORKED" from employees order by round((sysdate-hire_date)/30,0) asc;
```

OUTPUT:

The screenshot shows a SQL query interface with the following details:

- Language: SQL
- Rows: 10
- Clear Command | Find Tables
- Save | Run
- Results tab selected
- Query: `select last_name, round((sysdate-hire_date)/30,0) as "MONTHS_WORKED" from employees order by round((sysdate-hire_date)/30,0) asc;`
- Output table:

LAST_NAME	MONTHS_WORKED
popp	0
raphealy	306
Mohan	317

- 3 rows returned in 0.01 seconds
- Download link

RESULT:

The query is executed successfully.

7. 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.

QUERY:

```
select last_name||' earns'||salary||' monthly but wants'||salary*3 as "DREAM_SALARIES" from employees;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (set to SQL), Clear Command, Find Tables, Save, and Run. Below the tabs, there are icons for refresh, copy, search, and paste. The main area contains the SQL query:

```
1 | select last_name||' earns'||salary||' monthly but wants'||salary*3 as "DREAM_SALARIES" from employees;
```

Below the query, there are tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The output section has a header "DREAM_SALARIES" and contains three rows of data:

DREAM_SALARIES
popp earns 6900 monthly but wants 20700
raphealy earns 11000 monthly but wants 33000
Mohan earns 4000 monthly but wants 12000

At the bottom left, it says "3 rows returned in 0.00 seconds". There is also a "Download" link.

RESULT:

The query is executed successfully.

8. 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.

QUERY:

```
select last_name, lpad(salary,15,'$') as "SALARY" from employees;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the editor area, there are tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The query entered is:

```
1 select last_name, lpad(salary, 15, '$') as "SALARY" from employees;
```

The results table has two columns: LAST_NAME and SALARY. The data is:

LAST_NAME	SALARY
popp	\$\$\$\$\$\$\$\$\$\$6900
raphealy	\$\$\$\$\$\$\$\$\$\$11000
Mohan	\$\$\$\$\$\$\$\$\$\$4000

3 rows returned in 0.01 seconds. There is a Download button at the bottom.

RESULT:

The query is executed successfully.

9. 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."

QUERY:

```
SELECT last_name, hire_date, TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6), 'MONDAY'), 'FMDay, "the
"FMDD "of "FMMonth, YYYY') AS REVIEW FROM employees;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the editor area, there are tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The query entered is:

```
1 SELECT last_name, hire_date, TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6), 'MONDAY'), 'FMDay, "the "FMDD "of "FMMonth, YYYY') AS REVIEW FROM employees;
2
```

The results table has three columns: LAST_NAME, HIRE_DATE, and REVIEW. The data is:

LAST_NAME	HIRE_DATE	REVIEW
popp	03/05/2024	Monday, the 09 of September, 2024
raphealy	02/03/1999	Monday, the 09 of August, 1999
Mohan	02/22/1998	Monday, the 24 of August, 1998

3 rows returned in 0.01 seconds. There is a Download button at the bottom.

RESULT:

The query is executed successfully.

10. 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.

QUERY:

```
SELECT last_name,hire_date,TO_CHAR(hire_date,'Day') as Day from employees order by TO_CHAR(hire_date,'Day');
```

OUTPUT:



A screenshot of a SQL query execution interface. The top bar shows 'Language' set to 'SQL', 'Rows' set to '10', and 'Run' and 'Save' buttons. Below the bar, the SQL query is displayed: 'SELECT last_name,hire_date,TO_CHAR(hire_date,'Day') as Day from employees order by TO_CHAR(hire_date,'Day');'. The results tab is selected, showing a table with three columns: LAST_NAME, HIRE_DATE, and DAY. The data rows are:

LAST_NAME	HIRE_DATE	DAY
Mohan	02/22/1998	Sunday
popp	03/05/2024	Tuesday
raphealy	02/03/1999	Wednesday

At the bottom left, it says '3 rows returned in 0.04 seconds'. There is also a 'Download' link.

RESULT:

The query is executed successfully.

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

DISPLAYING DATA FROM MULTIPLE TABLES

EX_NO:7

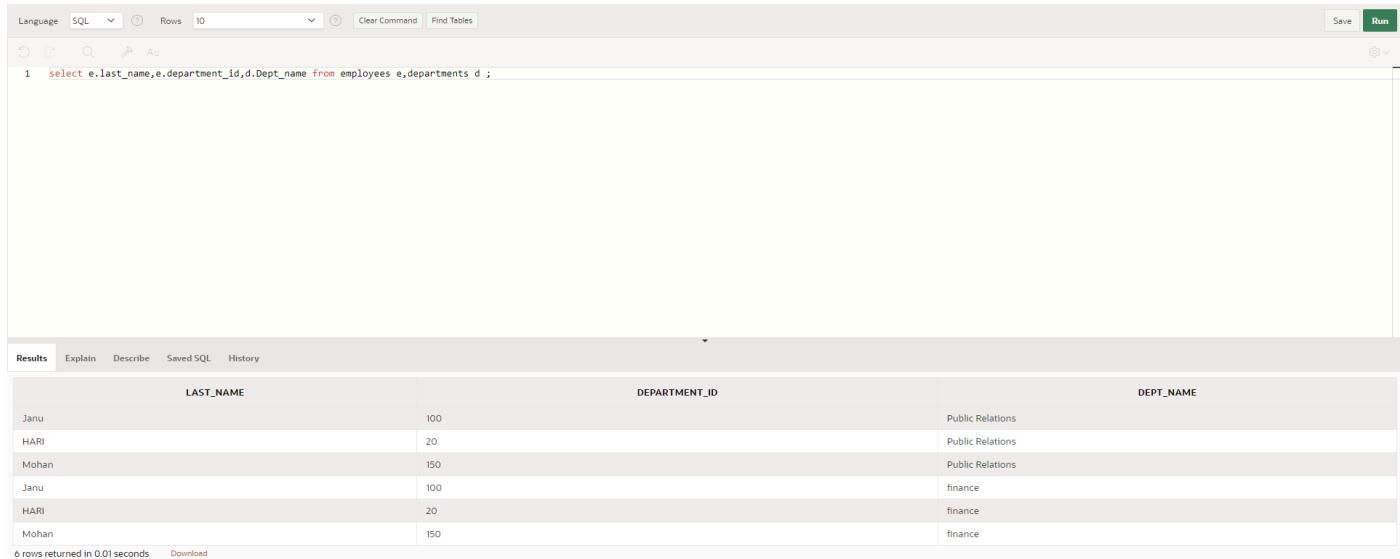
DATE:16.3.24

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

QUERY:

```
Select e.last_name,e.department_number,d.dept_id from employees e,departments d where e.department_number=d.dept_id;
```

OUTPUT:



The screenshot shows a SQL query execution interface. At the top, there are tabs for 'Language' (set to 'SQL'), 'Rows' (set to 10), and buttons for 'Clear Command', 'Find Tables', 'Save', and 'Run'. Below this is a code editor containing the following SQL query:

```
1 select e.last_name,e.department_id,d.Dept_name from employees e,departments d ;
```

Below the code editor is a results table with three columns: 'LAST_NAME', 'DEPARTMENT_ID', and 'DEPT_NAME'. The data is as follows:

LAST_NAME	DEPARTMENT_ID	DEPT_NAME
Janu	100	Public Relations
HARI	20	Public Relations
Mohan	150	Public Relations
Janu	100	finance
HARI	20	finance
Mohan	150	finance

At the bottom left of the results area, it says '6 rows returned in 0.01 seconds' and has a 'Download' link.

RESULT:

The query is executed successfully.

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

QUERY:

```
select distinct job_id,loc_id from employees e,departments d where e.department_number=d.dept_id and e.department_number=80;
```

OUTPUT:

The screenshot shows a SQL query execution interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the input area, there are icons for Refresh, Stop, Search, and Help. The query entered is:

```
1 select distinct job_id, LOCATION_ID FROM EMPLOYEES,DEPARTMENTS where employees.department_id=departments.dept_id and employees.department_id=80;
```

Below the query, the results tab is selected. The output table has two columns: JOB_ID and LOCATION_ID. A single row is shown:

JOB_ID	LOCATION_ID
ac_account	4598

Below the table, it says "1 rows returned in 0.02 seconds" and there is a "Download" link.

RESULT:

The query is executed successfully.

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

QUERY:

```
Select e.last_name,e.department_number,d.dept_name,d.loc_id,l.city from employees e,departments d,location l  
where e.department_number=d.dept_id and d.loc_id=l.location_id and e.commission_pct is not null;
```

OUTPUT:

The screenshot shows a SQL query execution interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the input area, there are icons for Refresh, Stop, Search, and Help. The query entered is:

```
1 select e.last_name,d.dept_name,d.location_id,l.city from employees e,departments d,location l where e.department_id=d.dept_id and d.location_id=l.location_id and e.commission_pct is not null
```

Below the query, the results tab is selected. The output area displays the message "no data found".

RESULT:

The query is executed successfully.

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

QUERY:

```
Select employees.last_name,departments.dept_name from employees,departments where employees.department_number=departments.dept_id and last_name like '%a%';
```

OUTPUT:



A screenshot of a SQL query execution interface. The top bar shows 'Language' set to 'SQL', 'Rows' set to 10, and a 'Run' button. The main area contains the SQL query: 'select last_name,dept_name from employees,departments where employees.department_id=departments.dept_id and last_name like "%a%"'. Below the query, the results are displayed in a table with two columns: 'LAST_NAME' and 'DEPT_NAME'. The result row shows 'Janu' in the LAST_NAME column and 'Public Relations' in the DEPT_NAME column. At the bottom left, it says '1 rows returned in 0.01 seconds'.

LAST_NAME	DEPT_NAME
Janu	Public Relations

RESULT:

The query is executed successfully.

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

QUERY:

```
Select e.last_name,e.department_number,e.job_id,d.dept_name from employees e join dept d on(e.department_number=d.dept_id) join location on (d.location_id=location.location_id) where lower(location.city)= 'toronto';
```

OUTPUT:

A screenshot of a SQL query editor interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and a 'Run' button. Below the toolbar is a search bar with icons for refresh, copy, and find, followed by a dropdown menu labeled 'A::'. The main area contains a single line of SQL code:

```
1 select last_name,job_id,department_id,dept_name from employees join departments d on (department_id=dept_id) join location l on(d.location_id=l.location_id) where lower(l.city)='toronto';
```

The results section at the bottom shows the status 'no data found'.

RESULT:

The query is executed successfully.

6. Display the employee last name and employee number along with their manager's last name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, Respectively

QUERY:

```
Select w.last_name "Employee",w.emp_id "emp#",m.last_name 'manager',m.emp_id "Mgr#" from employees  
m on (w.manager_id=m.emp_id);
```

OUTPUT:

A screenshot of a SQL query editor interface, identical to the one above, showing the same query execution and results.

RESULT:

The query is executed successfully.

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

QUERY:

Select w.last_name "Employee",w.emp_id "emp#",m.last_name 'manager',m.emp_id "Mgr#" from employees w left outer join employees m on (w.manager_id=m.emp_id);

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Run button
- SQL command:
1 select w.last_name "Employee",w.emp_id "emp#",m.last_name "Manager",m.emp_id "Mgr#" from employees w left outer join employees m on(w.manager_id=m.emp_id);
- Results tab selected
- Table output:

Employee	Emp#	Manager	Mgr#
Janu	113	-	-
Mohan	2	-	-
HARI	114	-	-
- Message: 3 rows returned in 0.01 seconds
- Download link

RESULT:

The query is executed successfully.

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

QUERY:

select e.department_number departments,e.last_name colleague from employees e join employees c on (e.department_number=c.department_number) where e.emp_id <> c.emp_id order by e.department_number,e.last_name,c.last_name;

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Run button
- SQL command:
1 select e.department_id departments,e.last_name employees,e.last_name colleague from employees e join employees c on(e.department_id=c.department_id) where e.employee_id<>c.employee_id
2 order by e.department_id,e.last_name;
- Results tab selected
- Message: no data found

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT e.last_name, e.job_id, d.dept_name, e.salary, j.grade_level
FROM emp18 e JOIN dept18 d
ON (e.department_id = d.dept_id)
JOIN job_grade j
ON (e.salary BETWEEN j.lowest_sal AND j.highest_sal);
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the tabs, the query is displayed:

```
1 SELECT e.last_name, e.job_id, d.dept_name, e.salary, j.grade_level
2 FROM employees e JOIN departments d
3 ON (e.department_id = d.dept_id)
4 JOIN job_grade j
5 ON (e.salary BETWEEN j.lowest_sal AND j.highest_sal);
```

Below the query, the results are shown in a table:

LAST_NAME	JOB_ID	DEPT_NAME	SALARY	GRADE_LEVEL
Janu	ac_account	Public Relations	6900	B
Janu	ac_account	Public Relations	6900	C

At the bottom left, it says "2 rows returned in 0.02 seconds". There is also a "Download" link.

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT e.last_name, e.hire_date
FROM emp18 e, emp18 davies
WHERE davies.last_name = 'Davies'
AND davies.hire_date < e.hire_date;
```

OUTPUT:

The screenshot shows a SQL query being run in a database environment. The query is:

```
1 SELECT e.last_name, e.hire_date
2 FROM employees e, employees davies
3 WHERE davies.last_name = 'Davies'
4 AND davies.hire_date < e.hire_date;
5
```

The results table has two columns: LAST_NAME and HIRE_DATE. The single row returned is:

LAST_NAME	HIRE_DATE
Janu	03/05/2024

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT e.last_name AS Employee, e.hire_date AS Emp_Hired,
e.manager_name AS Manager, m.hire_date AS Mgr_Hired
FROM emp18 e
JOIN emp18|m ON e.manager_name = m.last_name
WHERE e.hire_date < m.hire_date;
```

OUTPUT:

The screenshot shows a SQL query being run in a database environment. The query is:

```
1 select e.last_name as EMPLOYEE,e.hire_date as EMP_Hired,
2      m.manager_name as MANAGER,m.hire_date as MGR_HIRED
3   from employees e
4  join employees m on m.manager_name=e.last_name
5 where e.hire_date>m.hire_date;
```

The results table has four columns: EMPLOYEE, EMP_Hired, MANAGER, and MGR_HIRED. No data is found.

EMPLOYEE	EMP_Hired	MANAGER	MGR_HIRED

no data found

RESULT:

The query is executed successfully.

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

AGGREGATING DATA USING GROUP FUNCTIONS

EX_NO : 8

DATE: 23.3.24

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

FALSE

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

QUERY:

```
select Round(Max (salary),0)"Maximum", Round (Min (salary),0) "Minimum",
round(sum(salary),0)"sum", round (avg(salary),0) "Average" from EMPLOYEES;
```

OUTPUT:

The screenshot shows a SQL query execution interface. The top bar includes 'Language' (set to SQL), 'Rows' (set to 10), 'Clear Command', 'Find Tables', 'Save', and a 'Run' button. Below the bar, there are icons for refresh, copy, search, and paste. The main area contains the SQL query: 'select MAX(salary) "Maximum",MIN(salary) "Minimum",SUM(salary) "Sum",AVG(salary) "Average" from employees;'. The results section shows a table with four columns: Maximum, Minimum, Sum, and Average. The values are 11000, 4000, 21900, and 7300 respectively. A note at the bottom says '1 rows returned in 0.03 seconds' and has a 'Download' link.

Maximum	Minimum	Sum	Average
11000	4000	21900	7300

RESULT:

The query is executed successfully.

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

QUERY:

```
select job_id ,Round(MAX(salary),0) "MAXIMUM",Round (Min(salary),0)"Minimum",Round
(SUM(Salary),0)"sum" ,Round (AVg (salary),0)"average" from EMPLOYEES group by job_id;
```

OUTPUT:

The screenshot shows a SQL query being run against a database. The query is:

```
1 select MAX(salary) "Maximum",MIN(salary) "Minimum",SUM(salary) "Sum",AVG(salary) "Average" from employees group by job_id;
```

The results are displayed in a table with four columns: Maximum, Minimum, Sum, and Average. The data is as follows:

Maximum	Minimum	Sum	Average
11000	4000	21900	7300

1 rows returned in 0.00 seconds [Download](#)

RESULT:

The query is executed successfully.

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.

QUERY:

```
select job_id, count(*) from EMPLOYEES group by job_id ;
select job_id, count(*) from EMPLOYEES where job_id='47' group by job_id ;
```

OUTPUT:

The screenshot shows a SQL query being run against a database. The query is:

```
1 select count(*) "No_of_people" from employees group by job_id;
```

The results are displayed in a table with one column: No_of_people. The data is as follows:

No_of_people
3

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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.

QUERY:

```
select count(distinct manager_id )"Number of managers" from employees;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Toolbar: Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Query pane: A single line of SQL: `1 select count(manager_id) "Number of Managers" from employees where manager_id is not null;`.
- Results pane: A table titled "Number of Managers" with one row containing the value 3. Below the table, it says "1 rows returned in 0.01 seconds".
- Bottom navigation: Results, Explain, Describe, Saved SQL, History.

RESULT:

The query is executed successfully.

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

QUERY:

```
select max(salary)-min(salary) difference from employees;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Toolbar: Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Query pane: A single line of SQL: `1 select MAX(salary)-MIN(salary) "DIFFERENCE" from employees;`.
- Results pane: A table titled "DIFFERENCE" with one row containing the value 7000. Below the table, it says "1 rows returned in 0.01 seconds".
- Bottom navigation: Results, Explain, Describe, Saved SQL, History.

RESULT:

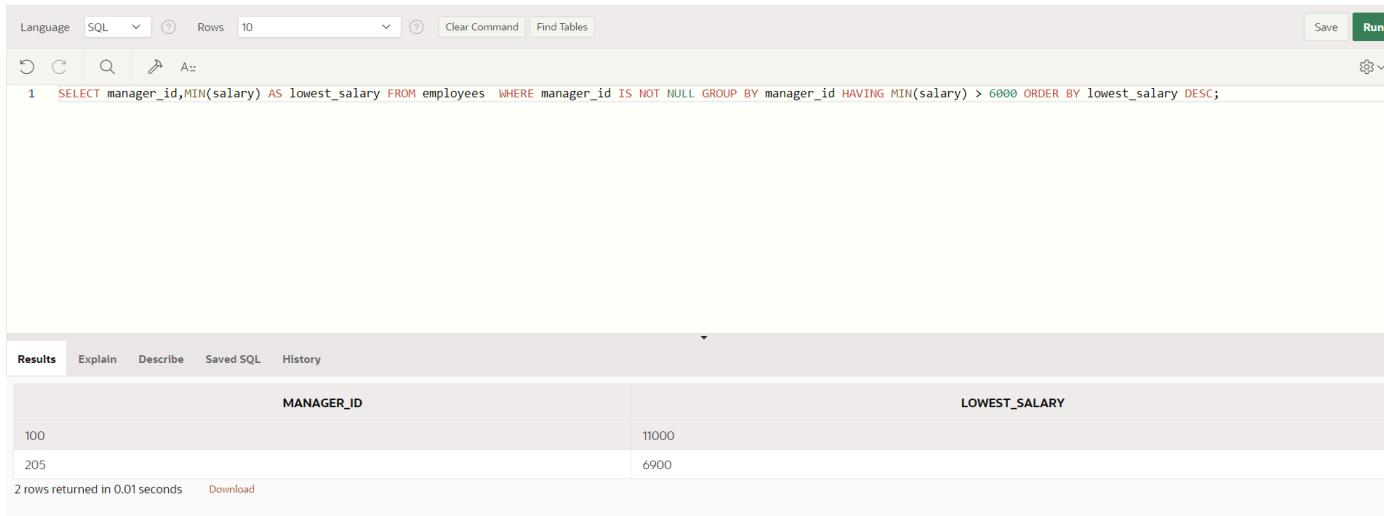
The query is executed successfully.

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.

QUERY:

```
select manager_id ,MIN(salary) from employees where manager_id is not null group by manager_id having min(salary) >6000 order by min(salary) desc;
```

OUTPUT:



A screenshot of a SQL query execution interface. The top bar shows 'Language' set to 'SQL', 'Rows' set to '10', and a 'Run' button. The query entered is:

```
1  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;
```

The results table has two columns: 'MANAGER_ID' and 'LOWEST_SALARY'. The data returned is:

MANAGER_ID	LOWEST_SALARY
100	11000
205	6900

Below the table, it says '2 rows returned in 0.01 seconds' and has a 'Download' link.

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

QUERY:

```
Select count(*) total,sum(decode(to_char(hire_date,'YYYY'),1995,1,0))"1995",sum(decode(to_char(hire_date,'YYYY'),1996,1,0))"1996",sum(decode(to_char(hire_date,'YYYY'),1997,1,0))"1997",sum(decode(to_char(hire_date,'YYYY'),1998,1,0)) "1998" from employees;
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query counts employees hired in specific years from 1995 to 1998. The results are displayed in a table with columns for TOTAL, 1995, 1996, 1997, and 1998. The data shows 3 total rows, 0 for 1995, 0 for 1996, 0 for 1997, and 1 for 1998.

TOTAL	1995	1996	1997	1998
3	0	0	0	1

3 rows returned in 0.04 seconds Download

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT DISTINCT job,
    SUM(CASE deptno WHEN 20 THEN sal END) "Dept 20",
    SUM(CASE deptno WHEN 50 THEN sal END) "Dept 50",
    SUM(CASE deptno WHEN 80 THEN sal END) "Dept 80",
    SUM(sal) "Total"
FROM emp
GROUP BY job;
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query displays the department ID, job ID, and sum of salary for departments 20, 50, 80, and 90. The results are displayed in a table with columns for DEPARTMENT_ID, JOB_ID, and SUM(SALARY). The data shows three rows corresponding to departments 80, 80, and -.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
80	ac_account	11000
80	-	11000
-	-	11000

3 rows returned in 0.03 seconds Download

RESULT:

The query is executed successfully.

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.

QUERY:

```
select d.dept_name as "dept_name",d.loc as "department location", count(*) "Number of people",round(avg(salary),2) "salary" from departments d inner join employees e on(d.dpt_id =e.department_id ) group by d.dept_name ,d.loc;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (set to SQL), Rows (set to 10), and various buttons like Save and Run. Below the tabs, the SQL code is displayed:

```
1 select d.dept_name as "Department name",l.location_id as "Location",count(e.department_id) as "Number of people",round(avg(e.salary),2) as "Salary"
2 from departments d,employees e,location l where d.dept_id=e.department_id group by d.dept_name,l.location_id,e.department_id;
```

The interface then displays the results in a table format. The table has four columns: Department name, Location, Number of people, and Salary. The data is as follows:

Department name	Location	Number of people	Salary
Public Relations	4598	1	6900
Public Relations	1231	1	6900
finance	4598	1	11000
finance	1231	1	11000

At the bottom left, it says "4 rows returned in 0.02 seconds".

RESULT:

The query is executed successfully.

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

SUB QUERIES

EX_NO:9

DATE:2.4.24

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

QUERY:

```
select last_name,hire_date from employees where department_id=(select department_id from
employees where last_name='Janu') and last_name not in('Janu');
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query is:

```
1 select last_name,hire_date from employees where department_id=(select department_id from employees where last_name='Janu') |  
2 and last_name not in('Janu');
```

The results table has two columns: LAST_NAME and HIRE_DATE. The single row returned is Doe, hired on 03/05/1997.

LAST_NAME	HIRE_DATE
Doe	03/05/1997

1 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select employee_id,last_name,salary from employees where salary>(select avg(salary) from employees) order by salary;
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query is:

```
1 select employee_id,last_name,salary from employees where salary>(select avg(salary) from employees) order by salary;
```

The results table has three columns: EMPLOYEE_ID, LAST_NAME, and SALARY. The two rows returned are Doe (Employee ID 142, Salary 30000) and Smith (Employee ID 1001, Salary 70000).

EMPLOYEE_ID	LAST_NAME	SALARY
142	Doe	30000
1001	Smith	70000

2 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select employee_id,last_name from employees where department_id=(select department_id from employees where last_name like'%u%');
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. Below the editor area, the query is displayed:

```
1 select employee_id,last_name from employees where department_id=(select department_id from employees where last_name like'%u%');
```

Below the query, the results are shown in a table format. The table has two columns: EMPLOYEE_ID and LAST_NAME. The data is as follows:

EMPLOYEE_ID	LAST_NAME
113	Janu
142	Doe

At the bottom left, it says "2 rows returned in 0.03 seconds". There is also a "Download" link.

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,department_id,job_id from employees where department_id=(select dept_id from departments where location_id=1700);
```

OUTPUT:

The screenshot shows a SQL query being run in a database environment. The query is:

```
1 select last_name,department_id,job_id from employees where department_id=(select dept_id from departments where location_id=1700);
```

The results pane displays the following data:

LAST_NAME	DEPARTMENT_ID	JOB_ID
Janu	100	ac_account
Doe	100	ac_account

2 rows returned in 0.04 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select last_name,salary from employees where manager_id=(select manager_id from employees where manager_name='King');
```

OUTPUT:

The screenshot shows a SQL query being run in a database environment. The query is:

```
1 select last_name,salary from employees where manager_id in (select manager_id from employees where manager_name='King');
```

The results pane displays the following data:

LAST_NAME	SALARY
Davies	11000
Mohan	4000

2 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select department_id, last_name, job_id from employees where department_id in (select dept_id from departments where dept_name='Executive');
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Save
- Run

SQL code:

```
1 select department_id, last_name, job_id from employees where department_id in (select dept_id from departments
2 where dept_name='Executive');
```

Results tab selected.

DEPARTMENT_ID	LAST_NAME	JOB_ID
80	Smith	Sales_rep
80	Davies	AC_ACCOUNT
20	Doe	AC_ACCOUNT

3 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

QUERY:

```
select employee_id, last_name, salary from employees where salary > (select avg(salary) from employees where last_name like '%u%');
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language: SQL
- Rows: 10
- Clear Command
- Find Tables
- Save
- Run

SQL code:

```
1 select employee_id, last_name, salary from employees where salary > (select avg(salary) from employees where last_name like '%u%');
```

Results tab selected.

EMPLOYEE_ID	LAST_NAME	SALARY
1001	Smith	70000
142	Doeu	30000

2 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

USING THE SET OPERATORS

EX_NO:10

DATE:9.4.24

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

QUERY:

```
select department_id from employees minus select department_id from employees where  
job_id='st_clerk';
```

OUTPUT:



The screenshot shows a SQL query execution interface. The query entered is:

```
1  select department_id from employees minus select department_id from employees where job_id='st_clerk';
```

The results pane shows a single column named "DEPARTMENT_ID" with two rows of data:

DEPARTMENT_ID
20
100

Below the results, it says "2 rows returned in 0.01 seconds" and has a "Download" link.

RESULT:

The query is executed successfully.

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

QUERY:

```
select country_id,state_province from location minus select country_id,state_province from  
location,departments where location.location_id=departments.location_id;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. The main area contains a SQL command:

```
1 select country_id,state_province from location minus select country_id,state_province from location,departments where location.location_id=departments.location_id;
```

Below the command, the Results tab is selected. The output table has two columns: COUNTRY_ID and STATE_PROVINCE. There is one row with COUNTRY_ID 58 and STATE_PROVINCE toronto. A note indicates 1 rows returned in 0.02 seconds. There is also a Download button.

RESULT:

The query is executed successfully.

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

QUERY:

```
select job_id,department_id from employees where department_id=10 union
select job_id,department_id from employees where department_id=50 union
select job_id,department_id from employees where department_id=20;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. The main area contains a SQL command:

```
1 select job_id,department_id from employees where department_id=10 union
2 select job_id,department_id from employees where department_id=50 union
3 select job_id,department_id from employees where department_id=20;
```

Below the command, the Results tab is selected. The output table has two columns: JOB_ID and DEPARTMENT_ID. There are three rows: ac_account with DEPARTMENT_ID 20, hr_rep with DEPARTMENT_ID 20, and another row with DEPARTMENT_ID 20. A note indicates 2 rows returned in 0.01 seconds. There is also a Download button.

RESULT:

The query is executed successfully.

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

QUERY:

```
select job_id,employee_id from employees intersect select e.job_id,e.employee_id from employees e,job_history j where e.job_id=j.old_job_id;
```

OUTPUT:

The screenshot shows a SQL query execution interface. At the top, there are tabs for Language (set to SQL), Rows (set to 10), and various buttons like Save and Run. The main area contains the SQL code: "select job_id,employee_id from employees intersect select e.job_id,e.employee_id from employees e,job_history j where e.job_id=j.old_job_id;". Below this, the Results tab is selected, showing a table with two columns: JOB_ID and EMPLOYEE_ID. The data returned is:

JOB_ID	EMPLOYEE_ID
ac_account	113
ac_account	142
sales_rep	1001

At the bottom left, it says "3 rows returned in 0.03 seconds".

RESULT:

The query is executed successfully.

5.) The HR department needs a report with the following specifications: - Last name and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to a department. - Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them. Write a compound query to accomplish this.

QUERY:

```
select first_name||' '||last_name as "Name",department_id from employees union all select dept_name,dept_id from departments;
```

OUTPUT:

The screenshot shows a SQL query being run in a database interface. The query is:

```
1 select first_name||' '||last_name as "Name",department_id from employees union all select dept_name,dept_id from departments;
```

The results are displayed in a table with two columns: **Name** and **DEPARTMENT_ID**. The data is as follows:

Name	DEPARTMENT_ID
John Smith	80
Emily Johnson	20
Jaunty Janu	100
den Davies	80
Jane Doe	20
Vijaya Mohan	150
Public Relations	100
finance	80
Executive	80
Executive	20

10 rows returned in 0.01 seconds [Download](#)

RESULT:

The query is executed successfully.

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

CREATING VIEWS

EX_NO:11

DATE:20.4.24

- 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.

QUERY:

```
CREATE OR REPLACE VIEW employees_vu AS SELECT employee_id, last_name employee,  
department_id FROM employees;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (set to SQL), Rows (set to 10), Clear Command, and Find Tables. On the right side, there are Save and Run buttons. Below the toolbar, there are icons for Undo, Redo, Search, and others. The main area contains a code editor with the following SQL command:

```
1 CREATE OR REPLACE VIEW employees_vu AS  
2   SELECT employee_id, last_name employee, department_id  
3   FROM employees;
```

Below the code editor, there is a results panel with tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The output shows:

```
View created.  
0.07 seconds
```

RESULT:

The query is executed successfully.

- 2.) Display the contents of the EMPLOYEES_VU view.

QUERY:

```
select * from employees_vu;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language:** SQL
- Rows:** 10
- Query:** `SELECT * FROM employees_vu;`
- Results:** A table with three columns: EMPLOYEE_ID, EMPLOYEE, and DEPARTMENT_ID.
- Data:**

EMPLOYEE_ID	EMPLOYEE	DEPARTMENT_ID
1001	Smith	80
125	Johnson	20
113	Janu	100
114	Davies	80
142	Doeu	20
115	Mohan	150
- Timing:** 6 rows returned in 0.03 seconds
- Actions:** Save, Run, Explain, Describe, Saved SQL, History

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT view_name, text FROM user_views;
```

OUTPUT:

The screenshot shows a SQL query editor with the following details:

- Language:** SQL
- Rows:** 10
- Query:** `SELECT view_name, text FROM user_views;`
- Results:** A table with two columns: VIEW_NAME and TEXT.
- Data:**

VIEW_NAME	TEXT
EMPLOYEES_VU	SELECT employee_id, last_name employee, department_id FROM employees
- Timing:** 1 rows returned in 0.06 seconds
- Actions:** Save, Run, Explain, Describe, Saved SQL, History

RESULT:

The query is executed successfully.

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

QUERY:

```
SELECT employee, department_id FROM employees_vu;
```

OUTPUT:

The screenshot shows a SQL query execution interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. Below the toolbar, the query is displayed: `1 SELECT employee, department_id FROM employees_vu;`. The results tab is selected, showing a table with two columns: EMPLOYEE and DEPARTMENT_ID. The data is as follows:

EMPLOYEE	DEPARTMENT_ID
Smith	80
Johnson	20
Janu	100
Davies	80
Doeu	20
Mohan	150

At the bottom left, it says "6 rows returned in 0.01 seconds". There is also a "Download" link.

RESULT:

The query is executed successfully.

5.)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.

QUERY:

```
CREATE VIEW dept50 AS SELECT employee_id empno, last_name employee, department_id deptno  
FROM employees WHERE department_id = 50 WITH CHECK OPTION CONSTRAINT emp_dept_50;
```

OUTPUT:

The screenshot shows a SQL query execution interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. Below the toolbar, the query is displayed: `1 CREATE VIEW dept50 AS
2 SELECT employee_id empno, last_name employee,
3 department_id deptno
4 FROM employees
5 WHERE department_id = 50
6 WITH CHECK OPTION CONSTRAINT emp_dept_50;`. The results tab is selected, showing the message "View created." and "0.06 seconds".

RESULT:

The query is executed successfully.

6.) Display the structure and contents of the DEPT50 view.

QUERY:

Describe dept50;

OUTPUT:

The screenshot shows the Oracle SQL Developer interface. The top bar has 'Language' set to 'SQL', 'Rows' set to '10', and buttons for 'Clear Command' and 'Find Tables'. On the right are 'Save' and 'Run' buttons. Below the toolbar is a toolbar with icons for refresh, search, and other database operations. The main area shows the command 'DESCRIBE dept50 ;' in the SQL editor. The results tab is selected, showing the structure of the view DEPT50. The 'Object Type' is 'VIEW' and the 'Object' is 'DEPT50'. The results table has columns: Table, Column, Data Type, Length, Precision, Scale, Primary Key, Nullable, Default, and Comment. The data shows three columns: EMPNO (NUMBER, 6, 0), EMPLOYEE (VARCHAR2, 20, 0), and DEPTNO (NUMBER, 4, 0). The DEPTNO column is marked as the primary key with a checkmark in the 'Primary Key' column.

RESULT:

The query is executed successfully.

7.) Attempt to reassign Matos to department 80

QUERY:

UPDATE dept50 SET deptno=80 WHERE employee='Matos';

OUTPUT:

The screenshot shows the Oracle SQL Developer interface. The top bar has 'Language' set to 'SQL', 'Rows' set to '10', and buttons for 'Clear Command' and 'Find Tables'. On the right are 'Save' and 'Run' buttons. Below the toolbar is a toolbar with icons for refresh, search, and other database operations. The main area shows the command 'UPDATE dept50 SET deptno=80 WHERE employee='Matos';'. The results tab is selected, showing the output of the update. It displays '0 row(s) updated.' and '0.05 seconds'.

RESULT:

The query is executed successfully.

8.) 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.

QUERY:

```
create or replace view salary_vu as select e.last_name "Employee",d.dept_name Department, e.salary "Salary",j.grade_level "Grades" from employees e,departments d,job_grade j where e.department_id=d.dept_id and e.salary between j.lowest_sal and j.highest_sal;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. Below the toolbar, there are icons for Undo, Redo, Search, and Filter. The main area contains the following SQL code:

```
1 create or replace view salary_vu as
2 select e.last_name "Employee",d.dept_name "Department",e.salary "Salary",j.grade_level "Grades"
3 from employees e,departments d,job_grade j
4 where e.department_id=d.dept_id and e.salary between j.lowest_sal and j.highest_sal;
```

Below the code, there are tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab is selected. The output shows:

View created.
0.06 seconds

RESULT:

The query is executed successfully.

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

EXERCISE 12

PRACTICE QUESTIONS

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?

Database can be as reliable as the data in it, and database rules are implemented as constraints to maintain data integrity.

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

- Constraints referring to more than one column are defined at Table Level
- NOT NULL constraint must be defined at column level as per ANSI/ISO SQL standard.

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

- If a constraint is violated in a SQL statement execution, it is easy to identify the cause with user-named constraints.
- It is easy to alter names/drop constraints.

4. Based on the information provided by the owners, choose a data type for each column. Indicate the length, precision, and scale for each NUMBER datatype.

Global Fast Foods global_locations Table						
NAME	TYPE	DataType	LENGTH	PRECISION	SCALE	NULLABLE
id	pk	NUMBER	6	0		No
name		VARCHAR2	50			
date_opened		DATE				No
address		VARCHAR2	50			No
city		VARCHAR2	30			No
zip_postal_code		VARCHAR2	12			
phone		VARCHAR2	20			
email	uk	VARCHAR2	75			
manager_id		NUMBER	6	0		
emergency_contact		VARCHAR2	20			

5. Use “(nullable)” to indicate those columns that can have null values.

Global Fast Foods global_locations Table						
NAME	TYPE	DataType	LENGTH	PRECISION	SCALE	NULLABLE
id	pk	NUMBER	6	0		No
name		VARCHAR2	50			Yes

date_opened		DATE			No
address		VARCHAR2	50		No
city		VARCHAR2	30		No
zip_postal_code		VARCHAR2	12		Yes
phone		VARCHAR2	20		Yes
email	uk	VARCHAR2	75		Yes
manager_id		NUMBER	6	0	Yes
emergency_contact		VARCHAR2	20		Yes

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

```
CREATE TABLE f_global_locations
( id NUMBER(6,0) CONSTRAINT f_gln_id_pk PRIMARY KEY ,
name VARCHAR2(50),
date_opened DATE CONSTRAINT f_gln_dt_opened_nn NOT NULL ENABLE,
address VARCHAR2(50) CONSTRAINT f_gln_add_nn NOT NULL ENABLE,
city VARCHAR2(30) CONSTRAINT f_gln_city_nn NOT NULL ENABLE,
zip_postal_code VARCHAR2(12),
phone VARCHAR2(20),
email VARCHAR2(75) CONSTRAINT f_gln_email_uk UNIQUE,
manager_id NUMBER(6,0),
emergency_contact VARCHAR2(20)
);
```

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

Table Created.

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

```
DESCRIBE f_global_locations;
```

9. 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.

NAME	TYPE	LENGTH	PRECISION	SCALE	NULLABLE	DEFAULT
id	number	4				
loc_name	varchar2	20			X	
	date					
address	varchar2	30				
city	varchar2	20				
zip_postal	varchar2	20			X	
phone	varchar2	15			X	
email	varchar2	80			X	
manager_id	number	4			X	
contact	varchar2	40			X	

```
CREATE TABLE f_global_locations
```

```
( id NUMBER(6,0) CONSTRAINT f_gln_id_pk PRIMARY KEY ,  
name VARCHAR2(50),  
date_opened DATE CONSTRAINT f_gln_dt_opened_nn NOT NULL ENABLE,  
address VARCHAR2(50) CONSTRAINT f_gln_add_nn NOT NULL ENABLE,  
city VARCHAR2(30) CONSTRAINT f_gln_city_nn NOT NULL ENABLE,  
zip_postal_code VARCHAR2(12),  
phone VARCHAR2(20),  
email VARCHAR2(75) ,  
manager_id NUMBER(6,0),  
emergency_contact VARCHAR2(20),  
CONSTRAINT f_gln_email_uk UNIQUE(email)  
);
```

RESULT:

The query is executed successfully.

PRIMARY KEY, FOREIGN KEY, and CHECK Constraints

1. What is the purpose of a
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK CONSTRAINT

a. PRIMARY KEY

Uniquely identify each row in the table.

b. FOREIGN KEY

Referential integrity constraint links back the parent table's primary/unique key to the child table's column.

c. CHECK CONSTRAINT

Explicitly define conditions to be met by each row's fields. This condition must be returned as true or unknown.

2. Using the column information for the animals table below, name constraints where applicable at the table level, otherwise name them at the column level. Define the primary key (animal_id). The license_tag_number must be unique. The admit_date and vaccination_date columns cannot contain null values.

animal_id NUMBER(6)	- PRIMARY KEY
name VARCHAR2(25)	
license_tag_number NUMBER(10)	- UNIQUE
admit_date DATE	-NOT NULL
adoption_id NUMBER(5),	
vaccination_date DATE	-NOT NULL

3. Create the animals table. Write the syntax you will use to create the table.

```
CREATE TABLE animals
( animal_id NUMBER(6,0) CONSTRAINT anl_anl_id_pk PRIMARY KEY ,
  name VARCHAR2(25),
  license_tag_number NUMBER(10,0) CONSTRAINT anl_l_tag_num_uk UNIQUE,
  admit_date DATE CONSTRAINT anl_adt_dat_nn NOT NULL ENABLE,
  adoption_id  NUMBER(5,0),
  vaccination_date DATE CONSTRAINT anl_vcc_dat_nn NOT NULL ENABLE
);
```

4. Enter one row into the table. Execute a SELECT * statement to verify your input. Refer to the graphic below for input.

ANIMAL_ID	NAME	LICENSE_TAG_NUMBE R	ADMIT_DAT E	ADOPTION_I D	VACCINATION_DAT E
101	Spot	35540	10-Oct-2004	205	12-Oct-2004

```
INSERT INTO animals (animal_id, name, license_tag_number, admit_date, adoption_id, vaccination_date)
VALUES( 101, 'Spot', 35540, TO_DATE('10-Oct-2004', 'DD-Mon-YYYY'), 205, TO_DATE('12-Oct-2004',
'DD-Mon-YYYY'));
```

```
SELECT * FROM animals;
```

5. Write the syntax to create a foreign key (adoption_id) in the animals table that has a corresponding primary-key reference in the adoptions table. Show both the column-level and table-level syntax. Note that because you have not actually created an adoptions table, no adoption_id primary key exists, so the foreign key cannot be added to the animals table.

COLUMN LEVEL STATEMENT:

```
ALTER TABLE animals
MODIFY ( adoption_id NUMBER(5,0) CONSTRAINT anl_adopt_id_fk REFERENCES adoptions(id)
ENABLE );
```

TABLE LEVEL STATEMENT:

```
ALTER TABLE animals ADD CONSTRAINT anl_adopt_id_fk FOREIGN KEY (adoption_id)
REFERENCES adoptions(id) ENABLE;
```

6. What is the effect of setting the foreign key in the ANIMAL table as:

- a. ON DELETE CASCADE

```
ALTER TABLE animals
ADD CONSTRAINT anl_adopt_id_fk FOREIGN KEY (adoption_id)
REFERENCES adoptions(id) ON DELETE CASCADE ENABLE ;
```

- b. ON DELETE SET NULL

```
ALTER TABLE animals
ADD CONSTRAINT anl_adopt_id_fk FOREIGN KEY (adoption_id)
REFERENCES adoptions(id) ON DELETE SET NULL ENABLE ;
```

7. What are the restrictions on defining a CHECK constraint?

- I cannot specify check constraint for a view however in this case I could use WITH CHECK OPTION clause
- I am restricted to columns from the self table and fields in self row.
- I cannot use subqueries and scalar subquery expressions.

RESULT:

The query is executed successfully.

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

PRACTICE PROBLEM

Managing Constraints

Using Oracle Application Express, click the SQL Workshop tab in the menu bar. Click the Object Browser and verify that you have a table named copy_d_clients and a table named copy_d_events. If you don't have these tables in your schema, create them before completing the exercises below. Here is how the original tables are related. The d_clients table has a primary key client_number. This has a primary-key constraint and it is referenced in the foreign-key constraint on the d_events table.

NOTE: The practice exercises use the d_clients and d_events tables in the DJs on Demand database. Students will work with copies of these two tables named copy_d_clients and copy_d_events. Make sure they have new copies of the tables (without changes made from previous exercises). Remember, tables copied using a subquery do not have the integrity constraints as established in the original tables. When using the SELECT statement to view the constraint name, the table names must be all capital letters.

1. What are four functions that an ALTER statement can perform on constraints?

- ADD
- DROP
- ENABLE
- DISABLE

2. Since the tables are copies of the original tables, the integrity rules are not passed onto the new tables; only the column datatype definitions remain. You will need to add a PRIMARY KEY constraint to the copy_d_clients table. Name the primary key copy_d_clients_pk . What is the syntax you used to create the PRIMARY KEY constraint to the copy_d_clients.table?

```
ALTER TABLE copy_d_clients
ADD CONSTRAINT copy_d_clt_client_number_pk PRIMARY KEY (client_number);
```

3. Create a FOREIGN KEY constraint in the copy_d_events table. Name the foreign key copy_d_events_fk. This key references the copy_d_clients table client_number column. What is the syntax you used to create the FOREIGN KEY constraint in the copy_d_events table?

```
ALTER TABLE copy_d_events
ADD CONSTRAINT copy_d_eve_client_number_fk FOREIGN KEY (client_number) REFERENCES
copy_d_clients (client_number) ENABLE;
```

4. Use a SELECT statement to verify the constraint names for each of the tables. Note that the tablename must be capitalized.

```
SELECT constraint_name, constraint_type, table_name
FROM user_constraints
WHERE table_name = UPPER('copy_d_events');
```

a. The constraint name for the primary key in the copy_d_clients table is_____.

COPY_D_CLT_CLIENT_NUMBER_PK

5. Drop the PRIMARY KEY constraint on the copy_d_clients table. Explain your results.

```
ALTER TABLE copy_d_clients
DROP CONSTRAINT COPY_D_CLT_CLIENT_NUMBER_PK CASCADE ;
```

6. Add the following event to the copy_d_events table. Explain your results.

ID	NAME	EVENT_DATE	DESCRIPTION	COST	VENUE_ID	PACKAGE_CODE	THEME_CODE	CLIENT_NUMBER
140	Cline Bas Mitzvah	15-Jul-2004	Church and Private Home formal	4500	105	87	77	7125

```
INSERT INTO copy_d_events(client_number,id,name,event_date,description,cost,venue_id,package_code,theme_code)
VALUES(7125,140,'Cline Bas Mitzvah',TO_DATE('15-Jul-2004','dd-Mon-yyyy'),'Church and Private Home formal',4500,105,87,77);
```

RESULT: ORA-02291: integrity constraint (HKUMAR.COPY_D_EVE_CLIENT_NUMBER_FK) violated - parent key not found

7. Create an ALTER TABLE query to disable the primary key in the copy_d_clients table. Then add the values from #6 to the copy_d_events table. Explain your results.

```
ALTER TABLE copy_d_clients
DISABLE CONSTRAINT COPY_D_CLT_CLIENT_NUMBER_PK CASCADE;
```

8. Repeat question 6: Insert the new values in the copy_d_events table. Explain your results.

```
INSERT INTO
copy_d_events(client_number,id,name,event_date,description,cost,venue_id,package_code,theme_code)
VALUES(7125,140,'Cline Bas Mitzvah',TO_DATE('15-Jul-2004','dd-Mon-yyyy'),'Church and Private Home formal',4500,105,87,77);
```

1 row(s) inserted.

9. Enable the primary-key constraint in the copy_d_clients table. Explain your results.

```
ALTER TABLE copy_d_clients
ENABLE CONSTRAINT COPY_D_CLT_CLIENT_NUMBER_PK ;
```

10. If you wanted to enable the foreign-key column and reestablish the referential integrity between these two tables, what must be done?

```
DELETE FROM copy_d_events WHERE
client_number NOT IN ( SELECT client_number FROM copy_d_clients);
```

1 row(s) deleted.

```
ALTER TABLE copy_d_events
ENABLE CONSTRAINT COPY_D_EVE_CLIENT_NUMBER_FK;
```

Table altered.

11. Why might you want to disable and then re-enable a constraint?

Generally to make bulk operations fast, where my input data is diligently sanitized and I am sure, it is safe to save some time in this clumsy process.

12. Query the data dictionary for some of the constraints that you have created. How does the data dictionary identify each constraint type?

Queries are the same as in point 2,3, 4 above.

- C - Check constraint
 - Sub-case - if I see SEARCH_CONDITION something like "FIRST_NAME" IS NOT NULL , it's a NOT NULL constraint.
- P - Primary key
- R - Referential integrity (fk)
- U - Unique key

RESULT:

The query is executed successfully.

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

EXERCISE 13

Creating Views

1. What are three uses for a view from a DBA's perspective?
 - **Restrict access and display selected columns**
 - **Reduce complexity of queries from other internal systems. So, providing a way to view same data in a different manner.**
 - **Let the app code rely on views and allow the internal implementation of tables to be modified later.**

2. 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 d_songs.id, d_songs.title "Song Title", d_songs.artist
from d_songs INNER JOIN d_types ON d_songs.type_code = d_types.code
where d_types.description = 'New Age';
```

3. SELECT * FROM view_d_songs. What was returned?

Results		
	Explain	Describe
	Saved SQL	History
ID	Song Title	ARTIST
47	Hurrah for Today	The Jubilant Trio
49	Lets Celebrate	The Celebrants

2 rows returned in 0.00 seconds [Download](#)

4. REPLACE view_d_songs. Add type_code to the column list. Use aliases for all columns. Or use alias after the CREATE statement as shown.

```
CREATE OR REPLACE VIEW view_d_songs AS
SELECT d_songs.id, d_songs.title "Song Title", d_songs.artist, d_songs.type_code
from d_songs INNER JOIN d_types ON d_songs.type_code = d_types.code
where d_types.description = 'New Age';
```

5. 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 OR REPLACE VIEW view_d_events_pkgs AS
SELECT evt.name "Name of Event", TO_CHAR(evt.event_date, 'dd-Month-yyyy') "Event date", thm.description
"Theme description"
FROM d_events evt INNER JOIN d_themes thm ON evt.theme_code = thm.code
```

```
WHERE evt.event_date <= ADD_MONTHS(SYSDATE,1);
```

6. 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 OR REPLACE VIEW view_min_max_avg_dpt_salary ("Department Id", "Department Name", "Max Salary", "Min Salary", "Average Salary") AS  
SELECT dpt.department_id, dpt.department_name, MAX(NVL(emp.salary,0)), MIN(NVL(emp.salary,0)),  
ROUND(AVG(NVL(emp.salary,0)),2)  
FROM departments dpt LEFT OUTER JOIN employees emp ON dpt.department_id = emp.department_id  
GROUP BY (dpt.department_id, dpt.department_name);
```

RESULT:

The query is executed successfully.

DML Operations and Views

Use the DESCRIBE statement to verify that you have tables named copy_d_songs, copy_d_events, copy_d_cds, and copy_d_clients in your schema. If you don't, write a query to create a copy of each.

1. Query the data dictionary USER_UPDATABLE_COLUMNS to make sure the columns in the base tables will allow UPDATE, INSERT, or DELETE. All table names in the data dictionary are stored in uppercase.

```
SELECT owner, table_name, column_name, updatable,insertable, deletable  
FROM user_updatable_columns WHERE LOWER(table_name) = 'copy_d_songs';
```

```
SELECT owner, table_name, column_name, updatable,insertable, deletable  
FROM user_updatable_columns WHERE LOWER(table_name) = 'copy_d_events';
```

```
SELECT owner, table_name, column_name, updatable,insertable, deletable  
FROM user_updatable_columns WHERE LOWER(table_name) = 'copy_d_cds';
```

2. Use the CREATE or REPLACE option to create a view of *all* the columns in the copy_d_songs table called view_copy_d_songs.

```
CREATE OR REPLACE VIEW view_copy_d_songs AS  
SELECT *  
FROM copy_d_songs;  
SELECT * FROM view_copy_d_songs;
```

3. Use view_copy_d_songs to INSERT the following data into the underlying copy_d_songs table. Execute a SELECT * from copy_d_songs to verify your DML command. See the graphic.

ID	TITLE	DURATION	ARTIST	TYPE_CODE
88	Mello Jello	2	The What	4

```
INSERT INTO view_copy_d_songs(id,title,duration,artist,type_code)  
VALUES(88,'Mello Jello','2 min','The What',4);
```

4. Create a view based on the DJs on Demand COPY_D_CDS table. Name the view read_copy_d_cds. Select all columns to be included in the view. Add a WHERE clause to restrict the year to 2000. Add the WITH READ ONLY option.

```
CREATE OR REPLACE VIEW read_copy_d_cds AS  
SELECT *  
FROM copy_d_cds  
WHERE year = '2000'  
WITH READ ONLY;  
SELECT * FROM read_copy_d_cds;
```

5. Using the read_copy_d_cds view, execute a DELETE FROM read_copy_d_cds WHERE cd_number = 90;

ORA-42399: cannot perform a DML operation on a read-only view

6. Use REPLACE to modify read_copy_d_cds. Replace the READ ONLY option with WITH CHECK OPTION CONSTRAINT ck_read_copy_d_cds. Execute a SELECT * statement to verify that the view exists.

CREATE OR REPLACE VIEW read_copy_d_cds AS

```
SELECT *
FROM copy_d_cds
WHERE year = '2000'
WITH CHECK OPTION CONSTRAINT ck_read_copy_d_cds;
```

7. Use the read_copy_d_cds view to delete any CD of year 2000 from the underlying copy_d_cds.

```
DELETE FROM read_copy_d_cds
WHERE year = '2000';
```

8. Use the read_copy_d_cds view to delete cd_number 90 from the underlying copy_d_cds table.

```
DELETE FROM read_copy_d_cds
WHERE cd_number = 90;
```

9. Use the read_copy_d_cds view to delete year 2001 records.

```
DELETE FROM read_copy_d_cds
WHERE year = '2001';
```

10. Execute a SELECT * statement for the base table copy_d_cds. What rows were deleted?

Only the one in problem 7 above, not the one in 8 and 9

11.What are the restrictions on modifying data through a view?

DELETE,INSERT,MODIFY restricted if it contains:

Group functions
GROUP BY CLAUSE
DISTINCT
pseudocolumn ROWNUM Keyword

12. What is Moore's Law? Do you consider that it will continue to apply indefinitely? Support your opinion with research from the internet.

It roughly predicted that computing power nearly doubles every year. But Moore also said in 2005 that as per the nature of exponential functions, this trend may not continue forever.

13. What is the "singularity" in terms of computing?

Singularity is the hypothesis that the invention of artificial superintelligence will abruptly trigger runaway technological growth, resulting in unfathomable changes to human civilization

RESULT:

The query is executed successfully.

Managing Views

1. Create a view from the copy_d_songs table called view_copy_d_songs that includes only the title and artist. Execute a SELECT * statement to verify that the view exists.

```
CREATE OR REPLACE VIEW view_copy_d_songs AS  
SELECT title, artist  
FROM copy_d_songs;  
  
SELECT * FROM view_copy_d_songs;
```

2. Issue a DROP view_copy_d_songs. Execute a SELECT * statement to verify that the view has been deleted.

```
DROP VIEW view_copy_d_songs;  
SELECT * FROM view_copy_d_songs;
```

ORA-00942: table or view does not exist

3. Create a query that selects the last name and salary from the Oracle database. Rank the salaries from highest to lowest for the top three employees.

```
SELECT * FROM  
(SELECT last_name, salary FROM employees ORDER BY salary DESC)  
WHERE ROWNUM <= 3;
```

4. Construct an inline view from the Oracle database that lists the last name, salary, department ID, and maximum salary for each department. Hint: One query will need to calculate maximum salary by department ID.

```
SELECT empm.last_name, empm.salary, dptmx.department_id  
FROM  
(SELECT dpt.department_id, MAX(NVL(emp.salary,0)) max_dpt_sal  
FROM departments dpt LEFT OUTER JOIN employees emp ON dpt.department_id = emp.department_id  
GROUP BY dpt.department_id) dptmx LEFT OUTER JOIN employees empm ON dptmx.department_id =  
empm.department_id  
WHERE NVL(empm.salary,0) = dptmx.max_dpt_sal;
```

5. Create a query that will return the staff members of Global Fast Foods ranked by salary from lowest to highest.

```
SELECT ROWNUM, last_name, salary  
FROM  
(SELECT * FROM f_staffs ORDER BY SALARY);
```

RESULT:

The query is executed successfully.

Indexes and Synonyms

1. What is an index and what is it used for?

Definition: These are schema objects which make retrieval of rows from table faster.

Purpose: An index provides direct and fast access to rows in the table. They provide an indexed path to locate data quickly, thereby reducing the necessity of heavy disk input/output operations.

2. What is a ROWID, and how is it used?

Indexes use ROWID (base 64 string representation of the row address containing block identifier, row location in the block and the database file identifier) which is the fastest way to access any particular row.

3. When will an index be created automatically?

Primary key/unique key uses an already existing unique index but if index is not present already, it is created while applying unique/primary key constraint.

4. Create a non unique index (foreign key) for the DJs on Demand column (cd_number) in the D_TRACK_LISTINGS table. Use the Oracle Application Express SQL Workshop Data Browser to confirm that the index was created.

**CREATE INDEX d_tlg_cd_number_fk_i
on d_track_listings (cd_number);**

5. Use the join statement to display the indexes and uniqueness that exist in the data dictionary for the DJs on Demand D_SONGS table.

**SELECT ucm.index_name, ucm.column_name, ucm.column_position, uix.uniqueness
FROM user_indexes uix INNER JOIN user_ind_columns ucm ON uix.index_name = ucm.index_name
WHERE ucm.table_name = 'D_SONGS';**

6. Use a SELECT statement to display the index_name, table_name, and uniqueness from the data dictionary USER_INDEXES for the DJs on Demand D_EVENTS table.

SELECT index_name, table_name,uniqueness FROM user_indexes where table_name = 'D_EVENTS';

7. Write a query to create a synonym called dj_tracks for the DJs on Demand d_track_listings table.

CREATE SYNONYM dj_tracks FOR d_track_listings;

8. Create a function-based index for the last_name column in DJs on Demand D_PARTNERS table that makes it possible not to have to capitalize the table name for searches. Write a SELECT statement that would use this index.

CREATE INDEX d_ptr_last_name_idx

ON d_partners(LOWER(last_name));

9. Create a synonym for the D_TRACK_LISTINGS table. Confirm that it has been created by querying the data dictionary.

CREATE SYNONYM dj_tracks2 FOR d_track_listings;

SELECT * FROM user_synonyms WHERE table_NAME = UPPER('d_track_listings');

10. Drop the synonym that you created in question

DROP SYNONYM dj_tracks2;

RESULT:

The query is executed successfully.

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

OTHER DATABASE OBJECTS

EX_NO:14

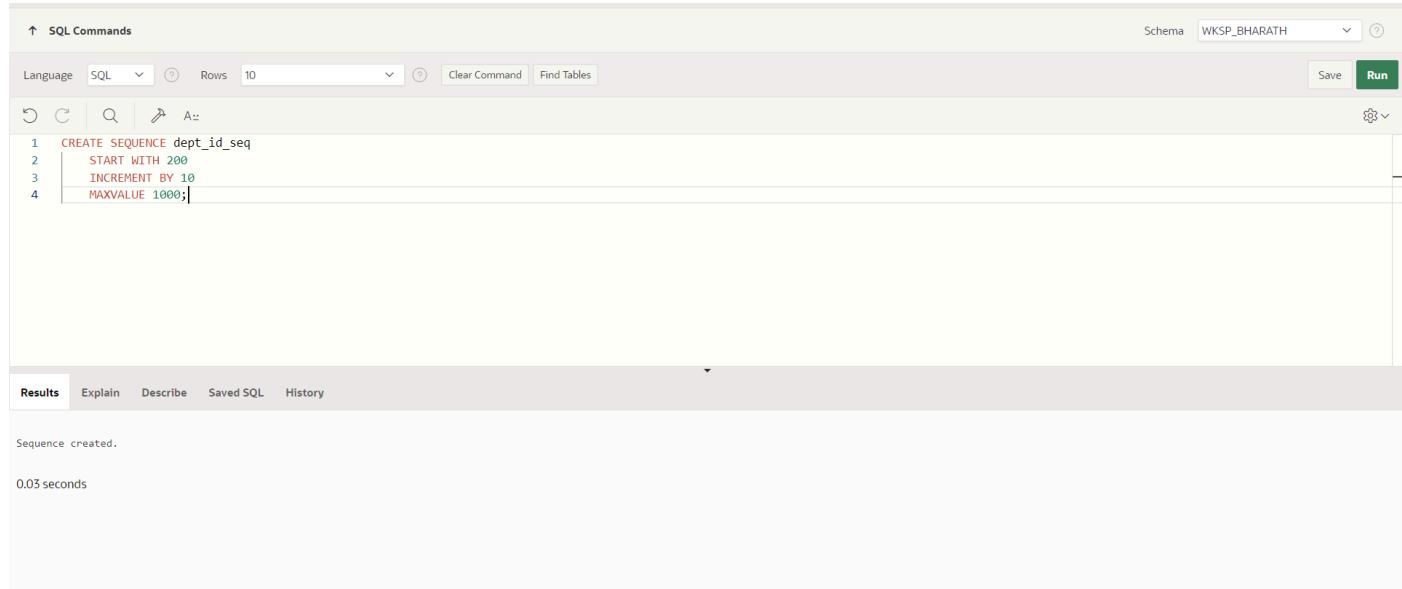
DATE:27.4.24

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

QUERY:

```
CREATE SEQUENCE dept_id_seq START WITH 200 INCREMENT BY 10 MAXVALUE 1000;
```

OUTPUT:



The screenshot shows a SQL command window in Oracle SQL Developer. The schema is set to 'WKSP_BHARATH'. The SQL tab contains the following code:

```
1 CREATE SEQUENCE dept_id_seq
2   START WITH 200
3   INCREMENT BY 10
4   MAXVALUE 1000;
```

The 'Results' tab shows the output:

```
Sequence created.
```

Execution time: 0.03 seconds

RESULT:

The query is executed successfully.

2.)Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number

QUERY:

```
SELECT sequence_name, max_value, increment_by, last_number FROM user_sequences;
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are tabs for Language (SQL), Rows (10), Clear Command, and Find Tables. On the right, there are Save and Run buttons. Below the toolbar, there are icons for refresh, search, and other database operations. The main area contains the following SQL code:

```

1  SELECT sequence_name, max_value, increment_by, last_number
2  FROM user_sequences;
3

```

Below the code, the Results tab is selected. The output table has columns: SEQUENCE_NAME, MAX_VALUE, INCREMENT_BY, and LAST_NUMBER. A single row is displayed:

SEQUENCE_NAME	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPT_ID_SEQ	1000	10	200

At the bottom left, it says "1 rows returned in 0.02 seconds". There is also a "Download" link.

RESULT:

The query is executed successfully.

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.

QUERY:

```
INSERT INTO dept VALUES (dept_id_seq.nextval, 'Education');
```

OUTPUT:

The screenshot shows a results page for a script named "exercise14". The status is "Complete". The summary table shows one row inserted with an elapsed time of 0.03 seconds. The details section shows the following table:

Number	Elapsed	Statement	Feedback	Rows
1	0.03	INSERT INTO dept VALUES (dept_id_seq.nextval, 'Education')	1 row(s) inserted.	1

At the bottom, it says "row(s) 1 - 1 of 1".

RESULT:

The query is executed successfully.

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

QUERY:

```
CREATE INDEX emp_dept_id_idx ON EMPLOYEES (department_id);
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables, along with Save and Run buttons. Below the toolbar is a toolbar with icons for copy, paste, search, and other functions. The main area contains the following SQL code:

```
1 CREATE INDEX emp_dept_id_idx ON EMPLOYEES (department_id);  
2
```

Below the code, the status bar shows "Index created." and "0.03 seconds". The bottom navigation bar includes Results, Explain, Describe, Saved SQL, and History.

RESULT:

The query is executed successfully.

5.)Display the indexes and uniqueness that exist in the data dictionary for the EMP table.

QUERY:

```
SELECT index_name,table_name,uniqueness FROM user_indexes WHERE table_name='EMPLOYEES';
```

OUTPUT:

The screenshot shows a SQL query editor interface. At the top, there are buttons for Language (SQL), Rows (10), Clear Command, and Find Tables, along with Save and Run buttons. Below the toolbar is a toolbar with icons for copy, paste, search, and other functions. The main area contains the following SQL code:

```
1 SELECT index_name, table_name, uniqueness  
2 FROM user_indexes  
3 WHERE table_name = 'EMPLOYEES';  
4
```

Below the code, the status bar shows "1 rows returned in 0.03 seconds." and a Download link. The bottom navigation bar includes Results, Explain, Describe, Saved SQL, and History.

INDEX_NAME	TABLE_NAME	UNIQUENESS
EMP_DEPT_ID_IDX	EMPLOYEES	NONUNIQUE

RESULT:

The query is executed successfully.

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

CONTROLLING USER ACCESS

EX_NO:15

DATE:4.5.24

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

The CREATE SESSION system privilege

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

The CREATE TABLE privilege

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

You can, or anyone you have given those privileges to by using the WITH GRANT OPTION.

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?

Create a role containing the system privileges and grant the role to the users

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

The ALTER USER statement

6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.

Team 2 executes the GRANT statement. GRANT select ON departments TO <user1>;

Team 1 executes the GRANT statement. GRANT select ON departments TO <user2>;

7. Query all the rows in your DEPARTMENTS table.

SELECT * FROM departments;

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department number 500. Team 2 should add Human Resources department number 510. Query the other team's table.

Team 1 executes this INSERT statement. INSERT INTO departments(department_id, department_name) VALUES (500, 'Education'); COMMIT;

Team 2 executes this INSERT statement. INSERT INTO departments(department_id, department_name) VALUES (510, 'Administration'); COMMIT;

9. Query the USER_TABLES data dictionary to see information about the tables that you own.

SELECT table_name FROM user_tables;

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

Team 1 revokes the privilege.

```
REVOKE select  
ON departments  
FROM user2;
```

Team 2 revokes the privilege.

```
REVOKE select  
ON departments  
FROM user1;
```

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

Team 1 executes this INSERT statement.

```
DELETE FROM departments  
WHERE department_id = 500;  
COMMIT;
```

Team 2 executes this INSERT statement.

```
DELETE FROM departments  
WHERE department_id = 510;  
COMMIT;
```

RESULT:

The query is executed successfully.

Evaluation Procedure	Marks awarded
<u>Practice Evaluation (5)</u>	
<u>Viva(5)</u>	
<u>Total (10)</u>	
Faculty Signature	

PL/SQL

CONTROL STRUCTURES

EX_NO: 16

DATE:8.5.24

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

PROCEDURE:

1. **Declare Variables**:

- `incentive NUMBER(8,2);`

- A variable named `incentive` is declared to store the calculated incentive amount. The data type `NUMBER(8,2)` means the variable can hold a number with up to 8 digits, 2 of which can be after the decimal point.

2. **Begin Block**:

- `BEGIN`

- This marks the start of the executable part of the PL/SQL block.

3. **SQL Query Execution**:

- `SELECT salary*0.12 INTO incentive FROM employees WHERE employee_id = 110;`

- This SQL query selects the `salary` of the employee with `employee_id` 110 from the `employees` table.

- The selected `salary` is multiplied by `0.12` to calculate the incentive, which is 12% of the salary.

- The result of this calculation is stored in the `incentive` variable.

4. **Output the Result**:

- `DBMS_OUTPUT.PUT_LINE('Incentive = ' || TO_CHAR(incentive));`

- This line outputs the calculated incentive using the `DBMS_OUTPUT.PUT_LINE` procedure.

- `TO_CHAR(incentive)` converts the numeric `incentive` value to a string so it can be concatenated with the text 'Incentive = '.

- The result is displayed as a message in the output.

5. **End Block**:

- `END;`

- This marks the end of the PL/SQL block.

Written Procedure

1. **Declare a Variable**:

- Define a variable `incentive` to store the calculated incentive.

2. **Start the Executable Section**:

- Begin the PL/SQL block with `BEGIN`.

3. **Perform a SQL Query to Calculate Incentive**:

- Execute a SQL `SELECT` statement to retrieve the `salary` of the employee with `employee_id` 110 from the `employees` table.
- Multiply the retrieved `salary` by 12% (0.12) to calculate the incentive.
- Store the result of this calculation in the `incentive` variable.

4. **Output the Calculated Incentive**:

- Use the `DBMS_OUTPUT.PUT_LINE` procedure to output the incentive amount.
- Convert the numeric `incentive` value to a string format for output.

5. **End the Block**:

- Conclude the PL/SQL block with `END;`.

This step-by-step explanation should give you a clear understanding of what each part of the provided PL/SQL block does and how it calculates and outputs the incentive for the employee with ID 110.

QUERY:

```
DECLARE
incentive  NUMBER(8,2);
BEGIN
SELECT salary*0.12 INTO incentive
FROM employees
WHERE employee_id = 110;
DBMS_OUTPUT.PUT_LINE('Incentive = ' || TO_CHAR(incentive));
```

END;

OUTPUT:

The screenshot shows a SQL developer interface. The top bar includes 'Language: SQL', 'Rows: 10', 'Clear Command', 'Find Tables', 'Save', and 'Run' buttons. Below the toolbar is a toolbar with icons for search, refresh, and other functions. The main area contains a code editor with the following PL/SQL block:

```
1 DECLARE
2 | incentive  NUMBER(8,2);
3 BEGIN
4 |   SELECT salary*0.12 INTO incentive
5 |   FROM employees
6 |   WHERE employee_id = 110;
7 DBMS_OUTPUT.PUT_LINE('Incentive  = ' || TO_CHAR(incentive));
8 END;
9
```

Below the code editor is a results panel with tabs for 'Results', 'Explain', 'Describe', 'Saved SQL', and 'History'. The 'Results' tab is selected, displaying the output of the executed query:

```
Incentive  = 8400
Statement processed.
0.00 seconds
```

RESULT:

The program is executed successfully.

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

PROCEDURE:

The procedure outlined in the code can be explained in these steps:

- Variable Declaration:** It creates a variable named "WELCOME" that can store text up to 10 characters long. This variable is then assigned the value "welcome".
- Printing the Message:** The code then instructs the program to print something to the console. It uses the `DBMS_OUTPUT.Put_Line` command to achieve this. However, there's a minor error in the original code.

Correction: The `DBMS_OUTPUT.Put_Line` command should be outside the `DECLARE` block. This is because the `DECLARE` block is meant to define variables, and the printing happens after the variable is set.

Here's the corrected procedure in words:

- Define a variable named "WELCOME" that can hold text (up to 10 characters).

2. Set the value of "WELCOME" to the text "welcome".
3. Print the value of the "WELCOME" variable to the console.

QUERY:

```
DECLARE
WELCOME varchar2(10) := 'welcome';
BEGIN
DBMS_Output.Put_Line("Welcome");
END;
/
```

```
DECLARE
WELCOME varchar2(10) := 'welcome';
BEGIN
DBMS_Output.Put_Line("Welcome");
END;
/
```

OUTPUT:

Language SQL Rows 10 Clear Command Find Tables Save Run

```
1 DECLARE
2   | "WELCOME" varchar2(10) := 'welcome';
3 BEGIN
4   | DBMS_Output.Put_Line("Welcome");
5 END;
6 /
7
```

Results Explain Describe Saved SQL History

Error at line 4/25: ORA-06550: line 4, column 25:
 PLS-00201: identifier 'Welcome' must be declared
 ORA-06512: at "SYS.WMV_DBMS_SQL_APEX_230200", line 801
 ORA-06550: line 4, column 3:
 PL/SQL: Statement ignored

```
2.   "WELCOME" varchar2(10) := 'welcome';
3. BEGIN
4.   DBMS_Output.Put_Line("Welcome");
5. END;
6. /
```

Language SQL Rows 10 Clear Command Find Tables Save Run

```
1 DECLARE
2   | WELCOME varchar2(10) := 'welcome';
3 BEGIN
4   | DBMS_Output.Put_Line("Welcome");
5 END;
6 /
7
```

Results Explain Describe Saved SQL History

Error at line 4/25: ORA-06550: line 4, column 25:
 PLS-00201: identifier 'Welcome' must be declared
 ORA-06512: at "SYS.WMV_DBMS_SQL_APEX_230200", line 801
 ORA-06550: line 4, column 3:
 PL/SQL: Statement ignored

```
2.   WELCOME varchar2(10) := 'welcome';
3. BEGIN
4.   DBMS_Output.Put_Line("Welcome");
5. END;
6. /
```

RESULT:

The program is executed successfully.

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

PROCEDURE:

The procedure simulates a salary increase for an employee but doesn't modify the actual data in the database. Here's how it works in words:

1. Setting Up:

- A variable named `salary_of_emp` is created to store an employee's salary with two decimal places.

2. Salary Increase Procedure:

- A procedure named `approx_salary` is defined. This procedure takes three inputs:
 - An employee ID (unused in this example).
 - A reference to the employee's salary (`empsal`). This allows the procedure to modify the salary value.
 - An amount to be added to the salary (`addless`).
- Inside the procedure, the value stored in `empsal` (which represents the current salary) is directly increased by adding the `addless` value.

3. Simulating Salary Increase:

- The main program block retrieves the current salary of an employee with ID 122 (assuming it exists in the `employees` table).
- It prints the current salary to the console.
- The `approx_salary` procedure is called with:
 - Employee ID set to 100 (ignored in the current logic).
 - The `salary_of_emp` variable as the reference to the salary (`empsal`).
 - An amount of 1000 to be added (`addless`).
- After the procedure call, the updated salary (which reflects the increase) is printed.

Important Note:

- This procedure only modifies a copy of the salary value within the program, not the actual employee data in the database. To permanently update the salary, a separate SQL statement like `UPDATE` would be needed.

QUERY:

```
DECLARE
    salary_of_emp NUMBER(8,2);
PROCEDURE approx_salary (
    emp      NUMBER,
    empsal IN OUT NUMBER,
    addless  NUMBER
) IS
BEGIN
    empsal := empsal + addless;
END;

BEGIN
    SELECT salary INTO salary_of_emp
    FROM employees
    WHERE employee_id = 122;
    DBMS_OUTPUT.PUT_LINE
    ('Before invoking procedure, salary_of_emp: ' || salary_of_emp);
    approx_salary (100, salary_of_emp, 1000);
    DBMS_OUTPUT.PUT_LINE
    ('After invoking procedure, salary_of_emp: ' || salary_of_emp);
END;
/
```

OUTPUT:

```

1 DECLARE
2   salary_of_emp  NUMBER(8,2);
3
4   PROCEDURE approx_salary (
5     emp      NUMBER,
6     empsal  IN OUT NUMBER,
7     adddress  NUMBER
8   ) IS
9   BEGIN
10    empsal := empsal + adddress;
11  END;
12
13 BEGIN
14   SELECT salary INTO salary_of_emp
15   FROM employees
16   WHERE employee_id = 122;
17

```

Results Explain Describe Saved SQL History

Before invoking procedure, salary_of_emp: 6900
After invoking procedure, salary_of_emp: 7900
Statement processed.
0.00 seconds

RESULT:

The procedure is executed successfully.

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

PROCEDURE:

The procedure you provided can be explained in words as follows:

1. Setting Up:

- It defines a procedure named `pri_bool` that takes two inputs:
 - A string (`boo_name`) that represents the name of a boolean variable.
 - A boolean variable itself (`boo_val`).

2. Checking the Boolean Value:

- The procedure uses an `IF` statement to check the value of `boo_val`:
 - If `boo_val` is NULL, it means the boolean variable doesn't have a clear TRUE or FALSE value assigned. In this case, the procedure prints a message stating "`boo_name` is NULL".
 - If `boo_val` is TRUE, it indicates a positive or affirmative condition. The procedure then prints a message stating "`boo_name` is TRUE".
 - If neither of the above conditions is true (meaning `boo_val` must be FALSE), it represents a negative or inactive condition. The procedure prints a message stating "`boo_name` is FALSE".

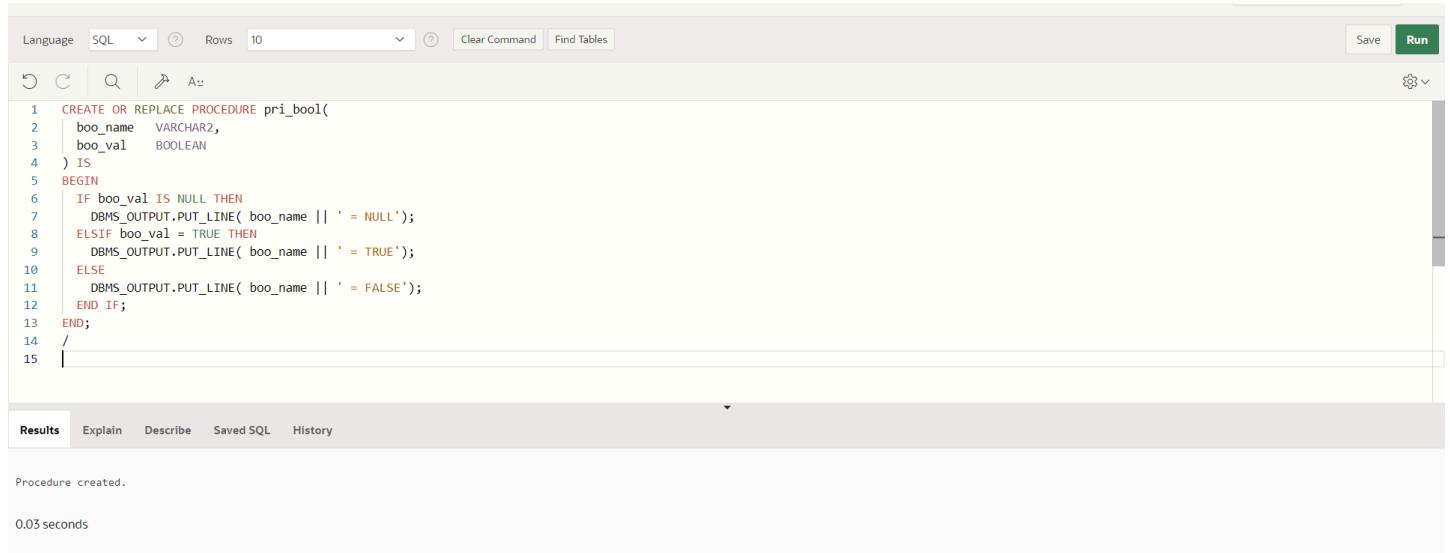
3. Purpose:

- This procedure is designed to make your code more readable and understandable. By providing the variable's name (`boo_name`) in the output message, it clarifies what boolean value is being checked.

QUERY:

```
CREATE OR REPLACE PROCEDURE pri_bool(
    boo_name  VARCHAR2,
    boo_val   BOOLEAN
) IS
BEGIN
    IF boo_val IS NULL THEN
        DBMS_OUTPUT.PUT_LINE( boo_name || ' = NULL');
    ELSIF boo_val = TRUE THEN
        DBMS_OUTPUT.PUT_LINE( boo_name || ' = TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE( boo_name || ' = FALSE');
    END IF;
END;
/
```

OUTPUT:



The screenshot shows the Oracle SQL Developer interface with the SQL tab selected. A procedure named `pri_bool` is being created. The code is as follows:

```
1 CREATE OR REPLACE PROCEDURE pri_bool(
2     boo_name  VARCHAR2,
3     boo_val   BOOLEAN
4 ) IS
5 BEGIN
6     IF boo_val IS NULL THEN
7         DBMS_OUTPUT.PUT_LINE( boo_name || ' = NULL');
8     ELSIF boo_val = TRUE THEN
9         DBMS_OUTPUT.PUT_LINE( boo_name || ' = TRUE');
10    ELSE
11        DBMS_OUTPUT.PUT_LINE( boo_name || ' = FALSE');
12    END IF;
13 END;
14 /
```

At the bottom of the interface, the results pane shows the message "Procedure created." and a execution time of "0.03 seconds".

RESULT:

The procedure is executed successfully.

5.) Write a PL/SQL block to describe the usage of LIKE operators including wildcard characters and escape characters.

PROCEDURE:

The algorithm for the provided PL/SQL code can be described as follows:

Procedure: pat_match

1. Inputs:

- `test_string`: String to be matched against the pattern.
- `pattern`: String containing wildcard characters for matching.

2. Matching Logic:

- Use the `LIKE` operator to compare `test_string` with `pattern`.
 - The `LIKE` operator performs pattern matching based on wildcards:
 - `%`: Matches any sequence of characters (including zero characters).
 - `_`: Matches a single character.
- If `test_string LIKE pattern`:
 - The strings match the pattern.
 - Return TRUE (indicate a match).
- Else:
 - The strings don't match the pattern.
 - Return FALSE (indicate no match).

Overall Algorithm:

1. Define the `pat_match` procedure with the specified inputs (`test_string` and `pattern`).
2. Inside the procedure:
 - Use the `IF` statement and the `LIKE` operator to check if `test_string` matches `pattern`.
 - If there's a match, print "TRUE" using `DBMS_OUTPUT.PUT_LINE`.
 - If there's no match, print "FALSE".
3. In the main program block:
 - Call the `pat_match` procedure with different test strings and patterns to perform the matching checks.

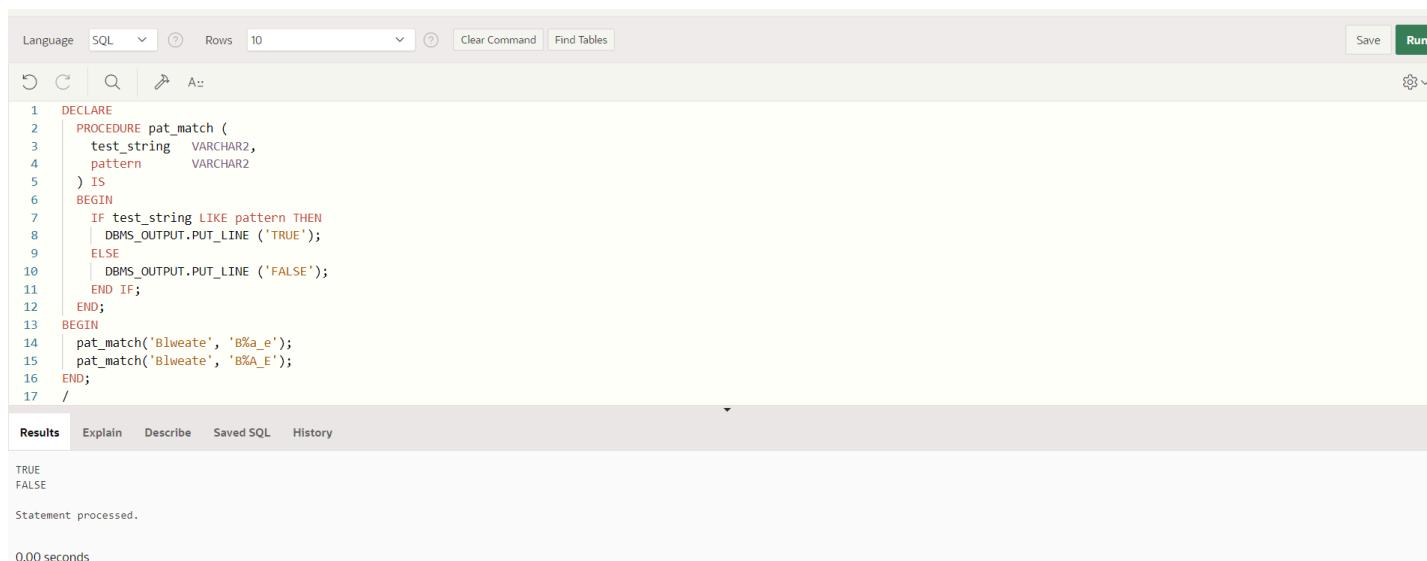
Time Complexity:

The time complexity of this algorithm depends on the complexity of the pattern and the length of the string. In most cases, the `LIKE` operator performs the matching efficiently. However, patterns with many wildcards or very long strings could potentially take longer to compare.

QUERY:

```
DECLARE
  PROCEDURE pat_match (
    test_string  VARCHAR2,
    pattern      VARCHAR2
  ) IS
  BEGIN
    IF test_string LIKE pattern THEN
      DBMS_OUTPUT.PUT_LINE ('TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('FALSE');
    END IF;
  END;
BEGIN
  pat_match('Blweate', 'B%a_e');
  pat_match('Blweate', 'B%A_E');
END;
/
```

OUTPUT:



The screenshot shows a database query editor interface. The top bar includes buttons for Language (SQL), Rows (10), Clear Command, Find Tables, Save, and Run. The main area displays the PL/SQL code for the `pat_match` procedure. The code uses the `DBMS_OUTPUT.PUT_LINE` function to output 'TRUE' for the first pattern and 'FALSE' for the second. The bottom section shows the results: 'TRUE' followed by 'FALSE'. A status message 'Statement processed.' and a timing message '0.00 seconds' are also present.

```
1  DECLARE
2    PROCEDURE pat_match (
3      test_string  VARCHAR2,
4      pattern      VARCHAR2
5    ) IS
6    BEGIN
7      IF test_string LIKE pattern THEN
8        | DBMS_OUTPUT.PUT_LINE ('TRUE');
9      ELSE
10        | DBMS_OUTPUT.PUT_LINE ('FALSE');
11      END IF;
12    END;
13  BEGIN
14    pat_match('Blweate', 'B%a_e');
15    pat_match('Blweate', 'B%A_E');
16  END;
17 /
```

Results Explain Describe Saved SQL History

TRUE
FALSE
Statement processed.
0.00 seconds

RESULT:

The procedure is executed successfully.

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

PROCEDURE

The algorithm for the provided PL/SQL code snippet can be described as follows:

1. Variable Initialization:

- Declare three numeric variables:
 - num_small: Assigned a starting value (here, 8).
 - num_large: Assigned a starting value (here, 5).
 - num_temp (temporary variable): Used to hold a value during the swap (initially not assigned a value).

2. Swapping Condition Check:

- Use an IF statement to check if num_small is greater than num_large.

3. Swapping Logic (if condition is true):

- If the condition is true (meaning num_small is larger):
 - Assign the current value of num_small to the temporary variable num_temp.
 - Assign the current value of num_large to num_small.
 - Assign the value stored in num_temp (which was originally the value of num_small) to num_large. This effectively swaps the contents of num_small and num_large.

4. Printing Results (always executed):

- Regardless of whether the swapping condition was true or false, use DBMS_OUTPUT.PUT_LINE statements to print the final values of both num_small and num_large to the console.

Time Complexity:

This algorithm has a time complexity of O(1), which is considered constant time. This is because the number of operations performed (variable assignments and comparisons) is constant and independent of the input values.

QUERY:

DECLARE

num_small NUMBER := 8;

num_large NUMBER := 5;

num_temp NUMBER;

BEGIN

IF num_small > num_large THEN

num_temp := num_small;

num_small := num_large;

num_large := num_temp;

END IF;

DBMS_OUTPUT.PUT_LINE ('num_small = '||num_small);

DBMS_OUTPUT.PUT_LINE ('num_large = '||num_large);

END;

/

OUTPUT:

The screenshot shows the Oracle SQL Developer interface. In the top navigation bar, 'Language' is set to 'SQL'. Below the toolbar, there are buttons for Undo, Redo, Find, and Save. The main area contains the PL/SQL code. The results pane at the bottom shows the output of the DBMS_OUTPUT.PUT_LINE statements.

```
2 num_small NUMBER := 8;
3 num_large NUMBER := 5;
4 num_temp NUMBER;
5 BEGIN
6
7 IF num_small > num_large THEN
8 num_temp := num_small;
9 num_small := num_large;
10 num_large := num_temp;
11 END IF;
12
13 DBMS_OUTPUT.PUT_LINE ('num_small = '||num_small);
14 DBMS_OUTPUT.PUT_LINE ('num_large = '||num_large);
15 END;
16 /
17
```

Results

```
num_small = 5
num_large = 8
Statement processed.

0.00 seconds
```

RESULT:

The program is executed successfully..

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

PROCEDURE

The procedure named `test1` simulates a scenario where an employee might receive a sales incentive.

Here's how it works in words:

1. Setting Up:

- The procedure takes three inputs:
 - `sal_achieve`: The employee's achieved sales amount (number).
 - `target_qty`: The target sales quantity for the employee (number).
 - `emp_id`: The employee's ID number.

2. Incentive Calculation (if target exceeded):

- The procedure checks if the achieved sales (`sal_achieve`) are greater than the target sales (`target_qty`) by at least 200.
- If the condition is met (employee exceeded the target), an incentive amount is calculated.
 - The incentive is calculated by subtracting the target quantity from the achieved sales and then dividing the result by 4.

3. Updating Employee Salary (if applicable):

- If an incentive was calculated (meaning the employee exceeded the target by enough), the procedure attempts to update the employee's salary in a table named `employees`.
 - It assumes a table named `employees` exists with a `salary` field for each employee identified by an `employee_id`.
 - The update would increase the employee's salary by the calculated incentive amount.

4. Note: This code snippet might not directly update the database due to potential permission limitations within the PL/SQL environment. A separate mechanism to commit the changes might be required.

5. Keeping Track of Updates:

- A variable named `updated` (initially 'No') is used to track whether the update happened.
- If the update is successful (assuming proper permissions exist), the variable is changed to 'Yes'.

6. Printing Information:

- Regardless of whether an incentive was applied or not, the procedure prints information to the console, including:
 - Whether the employee table was updated (based on the `updated` variable).
 - The amount of incentive calculated (might be 0 if the target wasn't met).

In essence, this procedure acts like a simulation of calculating and potentially applying a sales incentive to an employee based on their performance. It interacts with a hypothetical `employees` table to update the salary but might require additional steps to commit those changes permanently.

QUERY:

```

DECLARE
  PROCEDURE test1 (
    sal_achieve NUMBER,
    target_qty NUMBER,
    emp_id NUMBER
  )
  IS
    incentive NUMBER := 0;
    updated VARCHAR2(3) := 'No';
  BEGIN
    IF sal_achieve > (target_qty + 200) THEN
      incentive := (sal_achieve - target_qty)/4;
      UPDATE employees
      SET salary = salary + incentive
      WHERE employee_id = emp_id;
      updated := 'Yes';
    END IF;
    DBMS_OUTPUT.PUT_LINE (
      'Table updated? ' || updated || ',' ||
      'incentive = ' || incentive || ''
    );
  END test1;
BEGIN
  test1(2300, 2000, 144);
  test1(3600, 3000, 145);
END;

```

/

OUTPUT:

```
1  DECLARE
2      PROCEDURE test1 (
3          sal_achieve NUMBER,
4          target_qty NUMBER,
5          emp_id NUMBER
6      )
7      IS
8          incentive NUMBER := 0;
9          updated VARCHAR2(3) := 'No';
10     BEGIN
11         IF sal_achieve > (target_qty + 200) THEN
12             incentive := (sal_achieve - target_qty)/4;
13         UPDATE employees
14             SET salary = salary + incentive
15             WHERE employee_id = emp_id;
16     END;
17 
```

Results Explain Describe Saved SQL History

Table updated? Yes, incentive = 75.
Table updated? Yes, incentive = 150.

1 row(s) updated.

0.02 seconds

RESULT:

The procedure is executed successfully.

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

PROCEDURE

The procedure named `test1` calculates a sales incentive for an employee based on their achieved sales amount. Here's how it works in words:

1. Setting Up:

- The procedure takes one input: `sal_achieve`, which is a number representing the employee's total sales achieved.

2. Incentive Calculation:

- The procedure uses an `IF-ELSIF-ELSE` statement to determine the incentive amount based on different sales thresholds:
 - If the `sal_achieve` is greater than 44000 (highest threshold), the incentive is set to 1800 (highest incentive).
 - Else if the `sal_achieve` is greater than 32000 (but not more than 44000), the incentive is set to 800 (medium incentive).
 - Else (if neither of the above conditions are met, meaning sales are below 32000), the incentive is set to 500 (default incentive).

3. Printing Information:

- The procedure prints a message to the console that shows:

- The employee's achieved sales amount (`sal_achieve`).
- The calculated incentive amount (`incentive`) based on the achieved sales.

Overall:

- This procedure defines a simple incentive structure with three tiers based on achieved sales performance.
- It calculates the incentive amount based on these tiers and displays the results along with the achieved sales figure.

QUERY:

```

DECLARE
  PROCEDURE test1 (sal_achieve NUMBER)
  IS
    incentive NUMBER := 0;
  BEGIN
    IF sal_achieve > 44000 THEN
      incentive := 1800;
    ELSIF sal_achieve > 32000 THEN
      incentive := 800;
    ELSE
      incentive := 500;
    END IF;
    DBMS_OUTPUT.NEW_LINE;
    DBMS_OUTPUT.PUT_LINE (
      'Sale achieved : ' || sal_achieve || ', incentive : ' || incentive || '');
  END test1;
BEGIN
  test1(45000);
  test1(36000);
  test1(28000);
END;
/

```

```

1  DECLARE
2  PROCEDURE test1 (sal_achieve NUMBER)
3  IS
4      incentive NUMBER := 0;
5  BEGIN
6      IF sal_achieve > 44000 THEN
7          incentive := 1800;
8      ELSIF sal_achieve > 32000 THEN
9          incentive := 800;
10     ELSE
11         incentive := 500;
12     END IF;
13     DBMS_OUTPUT.NEW_LINE;
14     DBMS_OUTPUT.PUT_LINE (
15         'Sale achieved : ' || sal_achieve || ', incentive : ' || incentive || ',');

```

Results Explain Describe Saved SQL History

Sale achieved : 45000, incentive : 1800.
 Sale achieved : 36000, incentive : 800.
 Sale achieved : 28000, incentive : 500.
 Statement processed.

0.00 seconds

RESULT:

The program is executed successfully.

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

PROCEDURE

The provided code doesn't define a procedure, but rather a code block that calculates and displays department vacancies. Here's how it works in words:

1. Enabling Output:

- It starts by turning on server output using `SET SERVEROUTPUT ON`. This allows the code to display messages on the console.

2. Setting Up Variables:

- Two numeric variables are declared:
 - `tot_emp`: This variable will store the total number of employees retrieved from a database query.
 - `get_dep_id`: This variable stores the department ID (set to 80 in this case) for which we want to calculate vacancies.

3. Counting Employees in Department:

- A `SELECT` statement is used to query the database and count the number of employees in a specific department:
 - It counts all employees (`COUNT(*)`) from the `employees` table (aliased as `e`).

- It joins the `employees` table with the `departments` table (aliased as `d`) based on the `department_id` column. This ensures we only count employees who belong to the specified department.
- It filters the results using a `WHERE` clause to include only employees whose `department_id` matches the value stored in `get_dep_id` (which is 80 by default).
- The total count is then stored in the `tot_emp` variable.

4. Displaying Employee Count:

- The code uses `DBMS_OUTPUT.PUT_LINE` to print a message to the console. This message shows the total number of employees found in the department with the ID stored in `get_dep_id`. It converts the number retrieved from the database to a string for display using `TO_CHAR`.

5. Checking for Vacancies:

- An `IF` statement checks if the total number of employees (`tot_emp`) is greater than or equal to 45 (assuming a maximum capacity of 45 employees for this department).
 - If the department is full (condition is true), another message is printed indicating there are no vacancies in that department.
- If the `IF` condition is false (meaning there are less than 45 employees), it calculates the number of vacancies:
 - It subtracts the total number of employees (`tot_emp`) from the assumed maximum capacity (45).
 - It converts the vacancy count to a string using `TO_CHAR`.
- Finally, it prints a message to the console indicating the number of vacancies in the department with the ID stored in `get_dep_id`.

In essence:

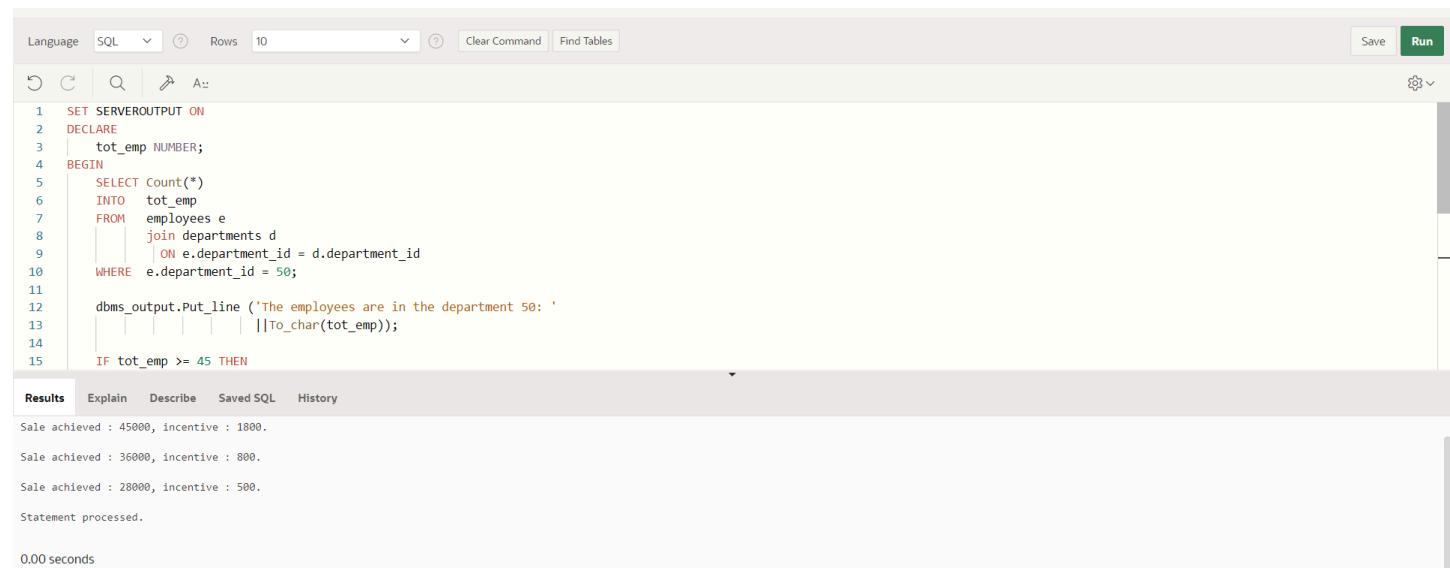
This code snippet simulates a scenario where you want to check how many openings (vacancies) are available in a specific department of a company. It retrieves employee data, calculates vacancies based on a predefined maximum capacity, and displays the results.

QUERY:

```
SET SERVEROUTPUT ON
DECLARE
    tot_emp NUMBER;
    get_dep_id NUMBER;

BEGIN
    get_dep_id := 80;
    SELECT Count(*)
        INTO tot_emp
        FROM employees e
        JOIN departments d
            ON e.department_id = d.department_id
    WHERE e.department_id = get_dep_id;
    dbms_output.Put_line ('The employees are in the department '||get_dep_id||' is: '
        ||To_char(tot_emp));
    IF tot_emp >= 45 THEN
        dbms_output.Put_line ('There are no vacancies in the department '||get_dep_id);
    ELSE
        dbms_output.Put_line ('There are '||to_char(45-tot_emp)||' vacancies in department '|| get_dep_id
    );
    END IF;
END;
/
```

OUTPUT:



The screenshot shows the Oracle SQL developer interface with the following details:

- Toolbar:** Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Code Area:** The query is displayed, including the declaration of variables, the cursor selection, and the conditional logic for outputting employee counts or vacancies.
- Results Area:** The output of the query is shown:
 - Sale achieved : 45000, incentive : 1800.
 - Sale achieved : 36000, incentive : 800.
 - Sale achieved : 28000, incentive : 500.
 - Statement processed.
- Timing:** 0.00 seconds.

RESULT:

The program is executed successfully.

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

PROCEDURE

The algorithm for the provided PL/SQL code can be described as follows:

1. Initialization:

- Enable server output for displaying messages (`SET SERVEROUTPUT ON`).
- Declare two numeric variables:
 - `get_dep_id`: Stores the department ID to check for vacancies (default value 80).
 - `tot_emp`: Stores the total number of employees retrieved from the database query.

2. Counting Employees in Department:

- Construct a `SELECT` statement:
 - Use `COUNT (*)` to count the number of rows (employees) in the result set.
 - Specify the `FROM` clause to retrieve data from the `employees` table (aliased as `e`).
 - Use a `JOIN` operation to combine data from the `employees` table with the `departments` table (aliased as `d`) based on the matching `department_id` columns.
 - Add a `WHERE` clause to filter the results and include only employees whose `department_id` in the `employees` table matches the value stored in the `get_dep_id` variable.

3. Displaying Employee Count:

- Use `DBMS_OUTPUT.PUT_LINE` to print a message to the console.
- The message includes:
 - Text indicating the department ID (`get_dep_id`).
 - The total number of employees (`tot_emp`) retrieved from the database query. Convert the number to a string using `TO_CHAR` for display.

4. Checking for Vacancies:

- Use an `IF` statement to check if the total number of employees (`tot_emp`) is greater than or equal to 45 (assuming a maximum capacity of 45 for this department).
 - If the condition is true (department is full):
 - Use `DBMS_OUTPUT.PUT_LINE` to print a message indicating there are no vacancies in that department.
- Else (department is not full):
 - Calculate the number of vacancies by subtracting the total number of employees (`tot_emp`) from the assumed maximum capacity (45).
 - Use `DBMS_OUTPUT.PUT_LINE` to print a message indicating the number of vacancies in the department with the ID stored in `get_dep_id`. Convert the vacancy count to a string using `TO_CHAR` for display.

5. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

Time Complexity:

This algorithm has a time complexity of $O(1)$ because the number of operations performed (variable assignments, comparisons, and the database query) is constant and independent of the input value (department ID). The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered constant.

QUERY:

DECLARE

```
tot_emp NUMBER;
get_dep_id NUMBER;
```

BEGIN

```
get_dep_id := 80;
SELECT Count(*)
INTO tot_emp
FROM employees e
```

```

join departments d
  ON e.department_id = d.dept_id
WHERE e.department_id = get_dep_id;

dbms_output.Put_line ('The employees are in the department'||get_dep_id||' is: '
    ||To_char(tot_emp));

IF tot_emp >= 45 THEN
  dbms_output.Put_line ('There are no vacancies in the department'||get_dep_id);
ELSE
  dbms_output.Put_line ('There are'||to_char(45-tot_emp)||' vacancies in department'||get_dep_id
);
END IF;
END;
/

```

OUTPUT:

```

Language SQL Rows 10 Clear Command Find Tables Save Run
DECLARE
  tot_emp NUMBER;
  get_dep_id NUMBER;
BEGIN
  get_dep_id := 80;
  SELECT Count(*)
  INTO tot_emp
  FROM employees e
  JOIN departments d
  ON e.department_id = d.dept_id
  WHERE e.department_id = get_dep_id;
  dbms_output.Put_line ('The employees are in the department'||get_dep_id||' is: '
    ||To_char(tot_emp));
END;

```

Results

```

The employees are in the department 80 is: 4
There are 41 vacancies in department 80
Statement processed.
0.03 seconds

```

RESULT:

The program is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL code that retrieves and displays employee information:

1. Variable Initialization:

- Declare several variables with data types matching the corresponding columns in the `employees` table:

- `v_employee_id`: Stores an employee ID (NUMBER).
- `v_full_name`: Stores an employee's full name (VARCHAR2).
- `v_job_id`: Stores an employee's job ID (VARCHAR2).
- `v_hire_date`: Stores an employee's hire date (DATE).
- `v_salary`: Stores an employee's salary (NUMBER).

2. Cursor Declaration:

- Declare a cursor named `c_employees`. This cursor acts as a pointer to navigate through the result set of a database query.

3. Cursor Definition:

- Define the SQL statement associated with the cursor (`c_employees`). This statement:

- Selects specific columns from the `employees` table:
 - `employee_id`
 - `first_name`
 - `last_name` (concatenated with `first_name` using `||` to create the full name)
 - `job_id`
 - `hire_date`
 - `salary`

4. Printing Header:

- Use `DBMS_OUTPUT.PUT_LINE` to print a header row to the console displaying labels for each employee attribute.
- Use `DBMS_OUTPUT.PUT_LINE` again to print a separator line.

5. Opening the Cursor:

- Use the `OPEN` statement to establish the cursor for reading data from the database.

6. Fetching Data (Loop):

- Use a `FETCH` statement within a `WHILE` loop to retrieve data from the cursor row by row.

- The `FETCH` statement assigns the retrieved values to the declared variables:
 - `v_employee_id`
 - `v_full_name`
 - `v_job_id`
 - `v_hire_date`
 - `v_salary`
- The `WHILE` loop continues as long as data can be fetched using `c_employees%FOUND` (cursor's "found" attribute).

7. Displaying Employee Information:

- Within the `WHILE` loop, use `DBMS_OUTPUT.PUT_LINE` to print a formatted line displaying the retrieved employee information for each row. It uses the values stored in the variables:
 - `v_employee_id`
 - `v_full_name`
 - `v_job_id`
 - `v_hire_date`
 - `v_salary`

8. Closing the Cursor:

- After the loop exits (no more data to fetch), use the `CLOSE` statement to release resources associated with the cursor.

9. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

Time Complexity:

This algorithm has a time complexity of $O(N)$, where N is the number of employees in the `employees` table. This is because the `WHILE` loop iterates through each row of data retrieved from the database. The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered part of the loop's processing.

QUERY:

```
DECLARE
v_employee_id employees.employee_id%TYPE;
v_full_name employees.first_name%TYPE;
v_job_id employees.job_id%TYPE;
v_hire_date employees.hire_date%TYPE;
v_salary employees.salary%TYPE;
CURSOR c_employees IS
SELECT employee_id, first_name || ' ' || last_name AS full_name, job_id, hire_date, salary
FROM employees;
BEGIN
DBMS_OUTPUT.PUT_LINE('Employee ID | Full Name | Job Title | Hire Date | Salary');
DBMS_OUTPUT.PUT_LINE('-----');
OPEN c_employees;
FETCH c_employees INTO v_employee_id, v_full_name, v_job_id, v_hire_date, v_salary;
WHILE c_employees%FOUND LOOP
DBMS_OUTPUT.PUT_LINE(v_employee_id || ' ' || v_full_name || ' ' || v_job_id || ' ' ||
v_hire_date || ' ' || v_salary);
FETCH c_employees INTO v_employee_id, v_full_name, v_job_id, v_hire_date, v_salary;
END LOOP;
CLOSE c_employees;
END;
/
```

OUTPUT:



```
1 DECLARE
2   v_employee_id employees.employee_id%TYPE;
3   v_full_name employees.first_name%TYPE;
4   v_job_id employees.job_id%TYPE;
5   v_hire_date employees.hire_date%TYPE;
6   v_salary employees.salary%TYPE;
7   CURSOR c_employees IS
8     SELECT employee_id, first_name || ' ' || last_name AS full_name, job_id, hire_date, salary
9     FROM employees;
10 BEGIN
11   DBMS_OUTPUT.PUT_LINE('Employee ID | Full Name | Job Title | Hire Date | Salary');
12   DBMS_OUTPUT.PUT_LINE('-----');
13   OPEN c_employees;
```

Employee ID	Full Name	Job Title	Hire Date	Salary
110	John Smith	sales_rep	02/28/1995	70000
125	Emily Johnson	hr_rep	04/26/1998	50000
122	Jaunty Janu	ac_account	03/05/2024	6900
114	den Davies	st_clerk	02/03/1999	11000
142	Jane Doe	ac_account	03/05/1997	30000
115	Vijaya Mohan	st_clerk	02/22/1998	4000

RESULT:

The program is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL code that retrieves and displays employee information including their manager's name:

1. Cursor Declaration:

- Declare a cursor named `emp_cursor`. This cursor will act as a pointer to navigate through the result set of a database query.

2. Cursor Definition:

- Define the SQL statement associated with the cursor (`emp_cursor`). This statement:
 - Uses a `SELECT` clause to retrieve data from two tables:
 - `employees` table (aliased as `e`): Selects `employee_id` and `first_name` (employee's name).
 - `employees` table again (aliased as `m`): Selects `first_name` of the manager.
 - Uses a `LEFT JOIN` to connect the `employees` table with itself. This joins rows in `e` with rows in `m` based on matching `manager_id` (in `e`) and `employee_id` (in `m`). This enables retrieving manager names for employees who have managers.

3. Record Variable Declaration:

- Declare a variable named `emp_record` with a data type that matches the structure of a row retrieved by the cursor (`emp_cursor%ROWTYPE`). This variable will store the fetched data for each employee.

4. Opening the Cursor:

- Use the `OPEN` statement to establish the cursor and prepare it to retrieve data from the database.

5. Fetching Data (Loop):

- Use a `WHILE` loop to iterate through the data fetched by the cursor:
 - Inside the loop, use a `FETCH` statement to retrieve data from the cursor and assign it to the `emp_record` variable. This variable now holds information about the current employee.
 - The `WHILE` loop continues as long as there is data to be fetched using the `emp_cursor%FOUND` attribute (cursor's "found" status).

6. Displaying Employee Information:

- Within the `WHILE` loop, use `DBMS_OUTPUT.PUT_LINE` statements to print the following details for each employee retrieved from `emp_record`:
 - Employee ID
 - Employee Name (first name)
 - Manager Name (first name of the manager, or `NULL` if no manager is assigned)
- Print a separator line after displaying information for each employee.

7. Closing the Cursor:

- After the loop exits (no more data to fetch), use the `CLOSE` statement to release resources associated with the cursor.

8. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

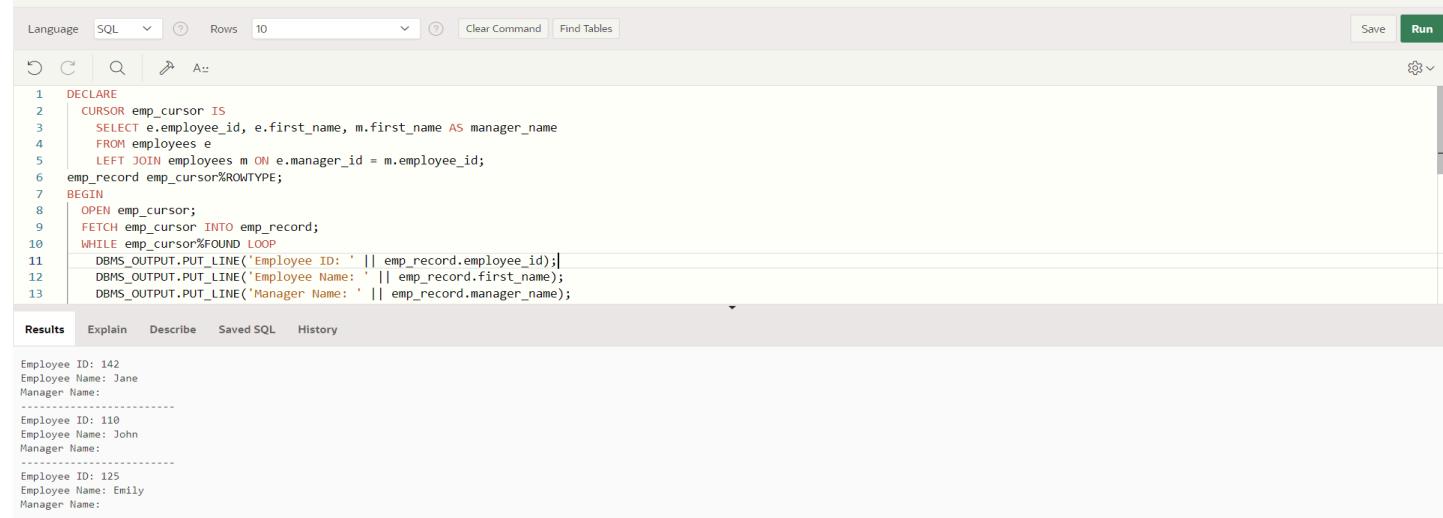
Time Complexity:

This algorithm has a time complexity of $O(N)$, where N is the number of employees in the `employees` table. This is because the `WHILE` loop iterates through each row of data retrieved from the database using the cursor. The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered part of the loop's processing.

QUERY:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT e.employee_id, e.first_name, m.first_name AS manager_name
    FROM employees e
    LEFT JOIN employees m ON e.manager_id = m.employee_id;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  FETCH emp_cursor INTO emp_record;
  WHILE emp_cursor%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.employee_id);
    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_record.first_name);
    DBMS_OUTPUT.PUT_LINE('Manager Name: ' || emp_record.manager_name);
    DBMS_OUTPUT.PUT_LINE('-----');
    FETCH emp_cursor INTO emp_record;
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

OUTPUT:



The screenshot shows the Oracle SQL developer interface with the query executed. The results pane displays the output of the DBMS_OUTPUT.PUT_LINE statements. The output is as follows:

```
Employee ID: 142
Employee Name: Jane
Manager Name:
-----
Employee ID: 110
Employee Name: John
Manager Name:
-----
Employee ID: 125
Employee Name: Emily
Manager Name:
-----
```

RESULT:

The query is executed successfully.

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

PROCEDURE

The provided PL/SQL code snippet retrieves minimum salary information associated with job IDs, but it might contain an error in the table join. Here's a breakdown of the algorithm assuming the `employees` table is not relevant to minimum salaries and was joined unintentionally:

1. Cursor Declaration:

- Declare a cursor named `job_cursor`. This cursor acts as a pointer to navigate through the result set of a database query.

2. Cursor Definition (Assuming Error in Join):

- Define the SQL statement associated with the cursor (`job_cursor`). This statement (assuming the `employees` table is not relevant):
 - Uses a `SELECT` clause to retrieve data from one table:
 - `job_grade` table (aliased as `j`): Selects the `job_id` and `lowest_sal` (assumed minimum salary for the job grade).
- Correction: Remove the reference to the `employees` table (aliased as `e`) from the `FROM` clause. This is likely a mistake as the `employees` table doesn't seem to provide relevant data for minimum salaries based on job grades.

3. Record Variable Declaration:

- Declare a variable named `job_record` with a data type that matches the structure of a row retrieved by the cursor (`job_cursor%ROWTYPE`). This variable will store the fetched data for each job.

4. Opening the Cursor:

- Use the `OPEN` statement to establish the cursor and prepare it to retrieve data from the database.

5. Fetching Data (Loop):

- Use a `WHILE` loop to iterate through the data fetched by the cursor:
 - Inside the loop, use a `FETCH` statement to retrieve data from the cursor and assign it to the `job_record` variable. This variable now holds information about the current job grade (including `job_id` and minimum salary).
 - The `WHILE` loop continues as long as there is data to be fetched using the `job_cursor%FOUND` attribute (cursor's "found" status).

6. Displaying Job Information:

- Within the `WHILE` loop, use `DBMS_OUTPUT.PUT_LINE` statements to print the following details for each job retrieved from `job_record`:
 - Job ID
 - Minimum Salary (from the `lowest_sal` column)
- Print a separator line after displaying information for each job grade.

7. Closing the Cursor:

- After the loop exits (no more data to fetch), use the `CLOSE` statement to release resources associated with the cursor.

8. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

Time Complexity:

This algorithm has a time complexity of $O(J)$, where J is the number of job grades in the `job_grade` table. This is because the `WHILE` loop iterates through each row of data retrieved from the database using the cursor. The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered part of the loop's processing.

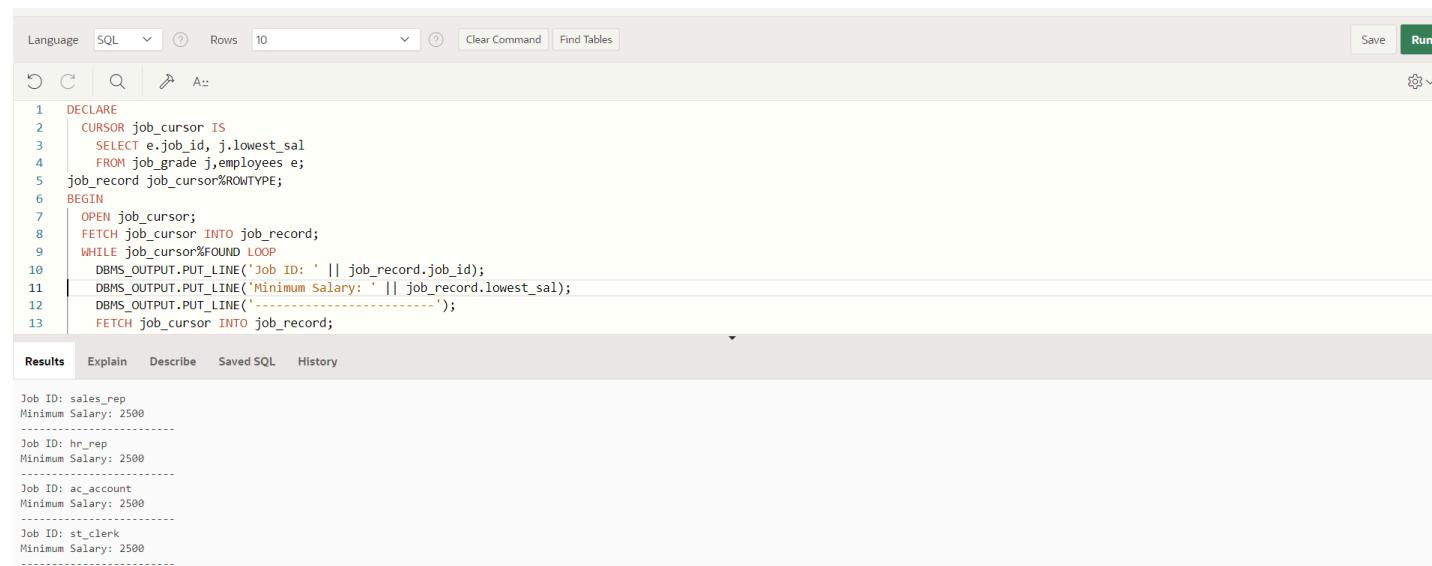
Note: If the `employees` table is actually intended to be part of the query and relevant to minimum salaries, the logic and cursor definition would need to be adjusted to incorporate the join condition between the tables.

QUERY:

DECLARE

```
CURSOR job_cursor IS
  SELECT e.job_id, j.lowest_sal
    FROM job_grade j,employees e;
job_record job_cursor%ROWTYPE;
BEGIN
  OPEN job_cursor;
  FETCH job_cursor INTO job_record;
  WHILE job_cursor%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE('Job ID: ' || job_record.job_id);
    DBMS_OUTPUT.PUT_LINE('Minimum Salary: ' || job_record.lowest_sal);
    DBMS_OUTPUT.PUT_LINE('-----');
    FETCH job_cursor INTO job_record;
  END LOOP;
  CLOSE job_cursor;
END;
/
```

OUTPUT:



The screenshot shows a database query editor interface with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Find Tables
- Toolbar:** Includes icons for Undo, Redo, Search, and others.
- Code Area:** Displays the PL/SQL code provided above.
- Results Area:** Displays the output of the code execution, showing four rows of data:

Job ID	Minimum Salary
sales_rep	2500
hr_rep	2500
ac_account	2500
st_clerk	2500

RESULT:

The program is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL procedure `Get_Employee_Start_Dates` that retrieves employee information and displays the most recent start date for each employee:

1. Procedure Declaration:

- Declare a procedure named `Get_Employee_Start_Dates`. This procedure encapsulates the logic for retrieving and displaying employee data.

2. Cursor Definition:

- Define a cursor named `employees_cur` to iterate through employee records. This cursor leverages a window function for efficiency.
- The `SELECT` statement within the cursor definition uses an `INNER JOIN` to combine data from the `employees` and `job_history` tables based on the matching `employee_id`.
- It retrieves the following data for each employee:
 - `employee_id`
 - `last_name`
 - `job_id`
 - `start_date` (from the `employees` table)
- A window function `MAX` is used with `PARTITION BY e.employee_id` and `ORDER BY jh.end_date DESC` within the subquery. This calculates the maximum `end_date` (potentially representing the end date of the previous job) from the `job_history` table for each employee group.
- The result of the window function is aliased as `prev_end_date`.

3. Printing Header Row:

- Use `DBMS_OUTPUT.PUT_LINE` statements to print a formatted header row displaying labels for each employee attribute:
 - Employee ID
 - Last Name
 - Job ID
 - Start Date

4. Looping through Employees:

- A `FOR` loop iterates through each row retrieved by the cursor and assigns the data to a record variable named `emp_record`.

5. Determining Start Date:

- Inside the loop, an `IF` statement checks if `emp_record.prev_end_date` is `NULL`:
 - If `NULL` (meaning no previous job history was found for the employee), the `start_date` retrieved from the cursor (`emp_record.start_date`) is used for the employee.
- Otherwise, the `prev_end_date` (most recent end date from job history) is incremented by 1 (assuming a one-day gap between jobs) and used as the start date.

6. Displaying Employee Information:

- Use `DBMS_OUTPUT.PUT_LINE` to print a formatted line with the following details for each employee:
 - `emp_record.employee_id` (padded for right alignment)
 - `emp_record.last_name` (padded for right alignment)
 - `emp_record.job_id` (padded for right alignment)
 - The `start_date` converted to a readable format using `TO_CHAR` (either the initial `start_date` or the incremented `prev_end_date`).

7. Loop Termination:

- The `FOR` loop continues iterating until all rows from the cursor have been processed.

8. Procedure Termination:

- The `END Get_Employee_Start_Dates;` statement marks the end of the procedure.

Time Complexity:

This algorithm has a time complexity of $O(N)$, where N is the number of employees in the `employees` table. This is because the cursor iterates through each employee record retrieved from the database using the window function within the cursor definition, which is typically optimized for efficiency. The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered part of the loop's processing.

QUERY:

```

DECLARE
  CURSOR employees_cur IS
    SELECT employee_id, last_name, job_id, start_date
    FROM employees NATURAL JOIN job_history;
    emp_start_date DATE;
BEGIN
  dbms_output.Put_line(Rpad('Employee ID', 15) || Rpad('Last Name', 25) || Rpad('Job Id', 35)
  || 'Start Date');
  dbms_output.Put_line('-----');
FOR emp_sal_rec IN employees_cur LOOP
  -- find out most recent end_date in job_history
  SELECT Max(end_date) + 1
  INTO emp_start_date
  FROM job_history
  WHERE employee_id = emp_sal_rec.employee_id;
  IF emp_start_date IS NULL THEN
    emp_start_date := emp_sal_rec.start_date;
  END IF;
  dbms_output.Put_line(Rpad(emp_sal_rec.employee_id, 15)
    || Rpad(emp_sal_rec.last_name, 25)
    || Rpad(emp_sal_rec.job_id, 35)
    || To_char(emp_start_date, 'dd-mon-yyyy'));
END LOOP;
END;

```

OUTPUT:

The screenshot shows a PL/SQL editor window in Oracle SQL Developer. The code declares a cursor for employees, selects employee_id, last_name, job_id, and start_date from employees NATURALLY JOIN job_history, and then prints these fields to the dbms_output. The results show two rows for employee 125: Johnson with job_id hr_rep and start date 22-apr-1999.

```

1  DECLARE
2      CURSOR employees_cur IS
3          SELECT employee_id,
4                  last_name,
5                      job_id,
6                      start_date
7              FROM employees
8          NATURAL JOIN job_history;
9      emp_start_date DATE;
10 BEGIN
11     dbms_output.Put_line(Rpad('Employee ID', 15)
12                           ||Rpad('Last Name', 25)
13                           ||Rpad('Job Id', 35)
14                           ||'Start Date');
15
16    dbms_output.Put_line('-----');
17

```

Employee ID	Last Name	Job Id	Start Date
125	Johnson	hr_rep	22-apr-1999
125	Johnson	hr_rep	22-apr-1999

Statement processed.

RESULT:

The program is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL code that retrieves and displays employee information with their most recent end date:

1. Variable Declaration:

- Declare three variables to hold data fetched from the database:
 - `v_employee_id`: Stores an employee ID (type: `employees.employee_id%TYPE`).
 - `v_first_name`: Stores an employee's first name (type: `employees.last_name%TYPE` - note the potential typo). This should likely be `e.first_name` to retrieve the first name based on the query.
 - `v_end_date`: Stores the end date of a job (type: `job_history.end_date%TYPE`).

2. Cursor Declaration:

- Declare a cursor named `c_employees` to iterate through employee records.

3. Cursor Definition:

- Define the SQL statement associated with the cursor (`c_employees`). This statement:
 - Uses a `JOIN` to combine data from two tables:
 - `employees` table (aliased as `e`): Selects `employee_id` and `first_name`.
 - `job_history` table (aliased as `jh`): Selects the most recent `end_date` for each employee using a join condition (`e.employee_id = jh.employee_id`). This likely retrieves the end date of the most recent job for each employee.
- Note: The cursor definition assumes the `job_history` table has an `end_date` column representing the end date of an employment period.

4. Opening the Cursor:

- Use the `OPEN` statement to prepare the cursor for fetching data from the database.

5. Fetching Data (Loop):

- Use a `WHILE` loop to iterate through the data retrieved by the cursor:
 - Inside the loop, use a `FETCH` statement to retrieve data from the cursor and assign it to the previously declared variables:
 - `v_employee_id`
 - `v_first_name`
 - `v_end_date`
 - The `WHILE` loop continues as long as there is data to be fetched using the `c_employees%FOUND` attribute (cursor's "found" status).

6. Displaying Employee Information:

- Inside the `WHILE` loop, use `DBMS_OUTPUT.PUT_LINE` statements to print the following details for each employee:
 - Employee ID (using `v_employee_id`)
 - Employee Name (using `v_first_name`)
 - End Date (using `v_end_date`)
- Print a separator line after displaying information for each employee.

7. Closing the Cursor:

- After the loop finishes iterating (no more data to fetch), use the `CLOSE` statement to release resources associated with the cursor.

8. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

Time Complexity:

This algorithm has a time complexity of $O(N)$, where N is the number of employees in the `employees` table. This is because the `WHILE` loop iterates through each row of data retrieved from the database using the cursor. The database query itself might have its own time complexity depending on the database engine and table size, but in the context of this algorithm, it's considered part of the loop's processing.

QUERY:

```
DECLARE
v_employee_id employees.employee_id%TYPE;
v_first_name employees.last_name%TYPE;
v_end_date job_history.end_date%TYPE;
CURSOR c_employees IS
  SELECT e.employee_id, e.first_name, jh.end_date
  FROM employees e
  JOIN job_history jh ON e.employee_id = jh.employee_id;
BEGIN
  OPEN c_employees;
  FETCH c_employees INTO v_employee_id, v_first_name, v_end_date;
  WHILE c_employees%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id);
    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_first_name);
    DBMS_OUTPUT.PUT_LINE('End Date: ' || v_end_date);
    DBMS_OUTPUT.PUT_LINE('-----');
    FETCH c_employees INTO v_employee_id, v_first_name, v_end_date;
  END LOOP;
```

```
CLOSE c_employees;
```

```
END;
```

OUTPUT:

The screenshot shows a SQL developer interface with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run
- Code Area:** Contains a PL/SQL block:

```
1 DECLARE
2   v_employee_id employees.employee_id%TYPE;
3   v_first_name employees.last_name%TYPE;
4   v_end_date job_history.end_date%TYPE;
5   CURSOR c_employees IS
6     SELECT e.employee_id, e.first_name, jh.end_date
7     FROM employees e
8     JOIN job_history jh ON e.employee_id = jh.employee_id;
9 BEGIN
10   OPEN c_employees;
11   FETCH c_employees INTO v_employee_id, v_first_name, v_end_date;
12   WHILE c_employees%FOUND LOOP
13     DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id);
14     DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_first_name);

```
- Results Tab:** Active tab, showing the output:

```
Employee ID: 125
Employee Name: Emily
End Date: 04/21/1999
-----
Employee ID: 125
Employee Name: Emily
End Date: 03/21/1997
-----
Statement processed.
```

RESULT:

The program is executed successfully.

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

PROCEDURES AND FUNCTIONS

EX_NO: 17

DATE:8.5.24

1.)Factorial of a number using a function.

PROCEDURE

Here's the algorithm for the provided PL/SQL code that calculates the factorial of a user-provided number:

1. Variable Declaration:

- Declare two numeric variables:
 - `fac`: This variable is initialized to 1 (type: `NUMBER`). It will be used to accumulate the factorial product.
 - `n`: This variable will store the user-provided number for which the factorial needs to be calculated (type: `NUMBER`). Its value is taken as input using the positional parameter `:1`.

2. Factorial Calculation Loop:

- Use a `WHILE` loop to iterate as long as `n` is greater than 0:
 - Inside the loop:
 - Multiply the current value of `fac` by `n` and store the result back in `fac`. This accumulates the product for the factorial.
 - Decrement `n` by 1 to move towards the base case (0).

3. Displaying the Result:

- After the loop exits (when `n` becomes 0), use `DBMS_OUTPUT.PUT_LINE` to print the final value of `fac`, which represents the factorial of the user-provided number.

4. Termination:

- The `BEGIN` and `END` keywords mark the beginning and end of the code block, respectively.

Time Complexity:

This algorithm has a time complexity of $O(n)$, where n is the user-provided number. The `WHILE` loop iterates n times, performing a constant-time multiplication and decrement operation within each iteration. The exact number of iterations depends on the user's input.

QUERY:

```
DECLARE
    fac NUMBER := 1;
    n NUMBER := :1;
BEGIN
    WHILE n > 0 LOOP
        fac := n * fac;
        n := n - 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(fac);
END;
```

OUTPUT:



The screenshot shows the Oracle SQL Developer interface with the following details:

- Toolbar:** Includes buttons for Undo, Redo, Find, Replace, and Save.
- Language Selector:** Set to PL/SQL.
- Schema:** WKSP_BHARATH.
- Run Button:** A green button labeled "Run".
- Code Area:** Displays the PL/SQL block provided in the "QUERY:" section.
- Results Tab:** Active tab, showing the output of the executed code.
- Output:**
 - Statement processed.
 - 0.00 seconds

RESULT:

The program is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL code snippet that demonstrates calling a procedure to retrieve book information:

1. Procedure `get_book_info`:

- **Parameters:**

- Define four parameters:
 - `p_book_id` (IN NUMBER): Input parameter for the book ID.
 - `p_title` (IN OUT VARCHAR2): IN/OUT parameter for book title (can be pre-populated and potentially modified).
 - `p_author` (OUT VARCHAR2): OUT parameter for author name.
 - `p_year_published` (OUT NUMBER): OUT parameter for year published.

- **Logic:**

- Execute a `SELECT` statement:
 - Retrieve book information (title, author, year_published) from the `books` table based on the provided `p_book_id`.
- Assign retrieved values to OUT parameters:
 - Set `p_title`, `p_author`, and `p_year_published` to the corresponding values from the database query.
- **Optional Modification (`p_title`):**
 - Append the string "- Retrieved" to the `p_title` value to indicate successful retrieval.

- **Exception Handling:**

- Catch the `NO_DATA_FOUND` exception:
 - If no matching book is found, set all OUT parameters (`p_title`, `p_author`, and `p_year_published`) to `NULL`.

2. Main Program Block:

- Declare variables:
 - `v_book_id` (NUMBER): Stores the book ID to be queried (e.g., 1).
 - `v_title` (VARCHAR2(100)): Variable to hold the book title (initially set to "Initial Title").
 - `v_author` (VARCHAR2(100)): Variable to hold the author name.
 - `v_year_published` (NUMBER): Variable to hold the year published.
- **Optional Initialization:**
 - Assign an initial value (e.g., "Initial Title") to `v_title`.
- **Calling the Procedure:**
 - Call the `get_book_info` procedure, passing the following arguments:
 - `p_book_id => v_book_id`: Pass the book ID for retrieval.
 - `p_title => v_title`: Pass the initial title value (potentially modified by the procedure).
 - `p_author => v_author`: Pass the author name variable (to be populated by the procedure).
 - `p_year_published => v_year_published`: Pass the year published variable (to be populated by the procedure).
- **Displaying Results:**
 - Use `DBMS_OUTPUT.PUT_LINE` statements to print the retrieved information:
 - Title (using `v_title`)
 - Author (using `v_author`)
 - Year Published (using `v_year_published`)

Overall Flow:

1. The main program defines variables and optionally initializes the title.
2. It calls the `get_book_info` procedure, passing the book ID and variables to hold the retrieved information.
3. The procedure retrieves book data based on the ID and populates the passed variables with the results (or `NULL` if no data is found).

4. The main program displays the retrieved title, author, and year published using the populated variables.

QUERY:

```
CREATE OR REPLACE PROCEDURE get_book_info (
    p_book_id IN NUMBER,
    p_title IN OUT VARCHAR2,
    p_author OUT VARCHAR2,
    p_year_published OUT NUMBER
)
AS
BEGIN
    SELECT title, author, year_published INTO p_title, p_author, p_year_published
    FROM books
    WHERE book_id = p_book_id;
```

```
    p_title := p_title || ' - Retrieved';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_title := NULL;
        p_author := NULL;
        p_year_published := NULL;
END;
```

```
DECLARE
    v_book_id NUMBER := 1;
    v_title VARCHAR2(100);
    v_author VARCHAR2(100);
    v_year_published NUMBER;
BEGIN
    v_title := 'Initial Title';

    get_book_info(p_book_id => v_book_id, p_title => v_title, p_author => v_author,
    p_year_published => v_year_published);

    DBMS_OUTPUT.PUT_LINE('Title: ' || v_title);
```

```
DBMS_OUTPUT.PUT_LINE('Author: ' || v_author);
DBMS_OUTPUT.PUT_LINE('Year Published: ' || v_year_published);
END;
```

OUTPUT:

The screenshot shows the Oracle SQL developer interface. The code area contains a PL/SQL block:

```
1  DECLARE
2      v_book_id NUMBER := 1;
3      v_title VARCHAR2(100);
4      v_author VARCHAR2(100);
5      v_year_published NUMBER;
6  BEGIN
7      v_title := 'Initial title';
8
9      get_book_info(p_book_id => v_book_id, p_title => v_title, p_author => v_author, p_year_published => v_year_published);
10
11     DBMS_OUTPUT.PUT_LINE('Title: ' || v_title);
12     DBMS_OUTPUT.PUT_LINE('Author: ' || v_author);
13     DBMS_OUTPUT.PUT_LINE('Year Published: ' || v_year_published);
14 END;
```

The results tab shows the output:

```
Title: To Kill a Mockingbird - Retrieved
Author: Harper Lee
Year Published: 1960

Statement processed.

0.01 seconds
```

RESULT:

The program is executed successfully.

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

TRIGGER

EX_NO: 18

DATE: 8.5.24

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

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `prevent_parent_deletion`:

1. Trigger Firing:

- The trigger fires before each DELETE operation on a row in the `parent_table`.

2. Checking for Child Records:

- The trigger declares a custom exception (`child_exists`) to signal the presence of child records.
- It then retrieves the `parent_id` of the row being deleted from the `parent_table` using the pseudo-record variable `:OLD`.
- A `SELECT` statement is executed to count the number of child records in the `child_table` that have a `parent_id` matching the one being deleted.

3. Preventing Deletion (if Applicable):

- An `IF` statement checks if the retrieved child record count (`v_child_count`) is greater than 0.
 - If `v_child_count` is greater than 0 (meaning child records exist), the `child_exists` exception is raised.

4. Exception Handling:

- An `EXCEPTION` block is defined to catch the `child_exists` exception raised within the trigger logic.

- Inside the exception block, another exception (`RAISE_APPLICATION_ERROR`) is raised with a user-defined error code (`-20001`) and an informative message indicating that the deletion is blocked due to existing child records.

5. Interaction with Main Program:

- The trigger itself doesn't directly interact with the main program that initiated the DELETE operation.
- When the `child_exists` exception is raised within the trigger, the database transaction attempting to delete the parent record fails due to the unhandled exception.
- The user or application that initiated the DELETE will likely receive an error message associated with the raised exception code (`-20001`) and its message (referential integrity constraint violation or similar).

Overall Logic:

The trigger acts as an enforcement mechanism to prevent data inconsistencies. It ensures that parent records cannot be deleted from the `parent_table` if there are child records referencing them in the `child_table`. The raised exception with a user-friendly message provides feedback to the user about the reason for the failed deletion.

QUERY:

```
CREATE OR REPLACE TRIGGER prevent_parent_deletion
BEFORE DELETE ON parent_table
FOR EACH ROW
DECLARE
    child_exists EXCEPTION;
    PRAGMA EXCEPTION_INIT(child_exists, -20001);
    v_child_count NUMBER;
BEGIN
```

```

SELECT COUNT(*) INTO v_child_count FROM child_table WHERE parent_id = :OLD.parent_id;
IF v_child_count > 0 THEN
    RAISE child_exists;
END IF;
EXCEPTION
    WHEN child_exists THEN
        RAISE_APPLICATION_ERROR(-20001, 'Cannot delete parent record while child records
exist.');
END;

```

OUTPUT:

The screenshot shows a SQL editor window with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Undo, Redo, Find, Clear Command, Find Tables.
- Code:**

```

1 CREATE OR REPLACE TRIGGER prevent_parent_deletion
2 BEFORE DELETE ON parent_table
3 FOR EACH ROW
4 DECLARE
5     child_exists EXCEPTION;
6     PRAGMA EXCEPTION_INIT(child_exists, -20001);
7     v_child_count NUMBER;
8 BEGIN
9     SELECT COUNT(*) INTO v_child_count FROM child_table WHERE parent_id = :OLD.parent_id;
10    IF v_child_count > 0 THEN
11        | RAISE child_exists;
12    END IF;
13 EXCEPTION
14    WHEN child_exists THEN
15        | RAISE_APPLICATION_ERROR(-20001, 'Cannot delete parent record while child records exist.');
16 END;
17 /

```
- Results Tab:** Trigger created.
- Timing:** 0.04 seconds

RESULT:

The trigger is executed successfully.

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

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `check_duplicates`:

1. Trigger Firing:

- The trigger fires before each INSERT or UPDATE operation on a row in the `unique_values_table`.

2. Checking for Duplicate Values:

- The trigger declares a custom exception (`duplicate_found`) to signal the presence of a duplicate value in the `unique_col`.
- It retrieves the `unique_col` value from the new row being inserted/updated using the pseudo-record variable `:NEW.unique_col`.
- A `SELECT` statement is executed to count the number of existing rows in the `unique_values_table` that have:
 - A `unique_col` value matching the new row's `unique_col` value.
 - An `id` value different from the new row's `id` (achieved using `id != :NEW.id`). This ensures the current row itself isn't counted as a duplicate.

3. Preventing Insert/Update (if Duplicate Found):

- An `IF` statement checks if the retrieved duplicate count (`v_count`) is greater than 0.
 - If `v_count` is greater than 0 (meaning a duplicate value exists), the `duplicate_found` exception is raised.

4. Exception Handling:

- An `EXCEPTION` block is defined to catch the `duplicate_found` exception raised within the trigger logic.
- Inside the exception block, another exception (`RAISE_APPLICATION_ERROR`) is raised with a user-defined error code (`-20002`) and an informative message indicating that the insert/update is blocked due to a duplicate value in the `unique_col`.

5. Interaction with Main Program:

- The trigger itself doesn't directly interact with the main program that initiated the `INSERT` or `UPDATE` operation.
- When the `duplicate_found` exception is raised within the trigger, the database transaction attempting to insert/update the row fails due to the unhandled exception.

- The user or application that initiated the INSERT or UPDATE will likely receive an error message associated with the raised exception code (-20002) and its message (referring to a duplicate value violation).

Overall Logic:

The trigger enforces a constraint on the `unique_values_table` to maintain data integrity. It prevents insert or update operations that would introduce duplicate values in the `unique_col`, excluding the row being modified itself (assuming a unique identifier column `id` exists). The raised exception with a user-friendly message provides feedback about the reason for the failed insert/update attempt.

QUERY:

```

CREATE OR REPLACE TRIGGER check_duplicates
BEFORE INSERT OR UPDATE ON unique_values_table
FOR EACH ROW
DECLARE
    duplicate_found EXCEPTION;
    PRAGMA EXCEPTION_INIT(duplicate_found, -20002);
    v_count NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_count FROM unique_values_table
    WHERE unique_col = :NEW.unique_col AND id != :NEW.id;
    IF v_count > 0 THEN
        RAISE duplicate_found;
    END IF;
EXCEPTION
    WHEN duplicate_found THEN
        RAISE_APPLICATION_ERROR(-20002, 'Duplicate value found in unique_col.');
END;

```

OUTPUT:

The screenshot shows a SQL editor window with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Clear Command, Find Tables
- Text Area Content:**

```

4  DECLARE
5      duplicate_found EXCEPTION;
6      PRAGMA EXCEPTION_INIT(duplicate_found, -20002);
7      v_count NUMBER;
8  BEGIN
9      SELECT COUNT(*) INTO v_count FROM unique_values_table
10     WHERE unique_col = :NEW.unique_col AND id != :NEW.id;
11     IF v_count > 0 THEN
12         RAISE duplicate_found;
13     END IF;
14  EXCEPTION
15     WHEN duplicate_found THEN
16         RAISE_APPLICATION_ERROR(-20002, 'Duplicate value found in unique_col.');
17  END;
18 /

```
- Results Tab:** Trigger created.
- Timing:** 0.04 seconds

RESULT:

The trigger is executed successfully..

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

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `check_threshold`:

1. Trigger Firing:

- The trigger fires before each INSERT or UPDATE operation on a row in the `threshold_table`.

2. Threshold Checking:

- The trigger declares a custom exception (`threshold_exceeded`) to signal exceeding the defined threshold.
- It defines a variable (`v_threshold`) to hold the threshold value (modifiable based on your needs).

3. Calculating Potential Sum (Before Insert/Update):

- A `SELECT` statement retrieves the current sum of the `value_col` from all existing rows in the `threshold_table`.
- The value being inserted/updated (`NEW.value_col`) is then added to the retrieved sum, storing the result in `v_sum`. This simulates the potential impact on the total sum after the operation.

4. Preventing Insert/Update (if Threshold Exceeded):

- An `IF` statement checks if the calculated `v_sum` (including the new/updated value) is greater than the `v_threshold`.
 - If true, the `threshold_exceeded` exception is raised.

5. Exception Handling:

- An `EXCEPTION` block is defined to catch the `threshold_exceeded` exception raised within the trigger logic.
- Inside the exception block, another exception (`RAISE_APPLICATION_ERROR`) is raised with a user-defined error code (`-20003`) and an informative message indicating that the insert/update is blocked because it would cause the sum to exceed the threshold.

6. Interaction with Main Program:

- The trigger itself doesn't directly interact with the main program that initiated the `INSERT` or `UPDATE` operation.
- When the `threshold_exceeded` exception is raised within the trigger, the database transaction attempting to insert/update the row fails due to the unhandled exception.
- The user or application that initiated the `INSERT` or `UPDATE` will likely receive an error message associated with the raised exception code (`-20003`) and its message (referring to exceeding the threshold for `value_col`).

Overall Logic:

The trigger enforces a business rule on the `threshold_table`. It prevents insert or update operations that would cause the sum of the `value_col` to exceed the predefined threshold. The raised exception with a user-friendly message provides feedback about the reason for the failed insert/update attempt.

QUERY:

```
CREATE OR REPLACE TRIGGER check_threshold
BEFORE INSERT OR UPDATE ON threshold_table
FOR EACH ROW
DECLARE
    threshold_exceeded EXCEPTION;
    PRAGMA EXCEPTION_INIT(threshold_exceeded, -20003);
    v_sum NUMBER;
    v_threshold NUMBER := 10000; -- Set your threshold here
BEGIN
    SELECT SUM(value_col) INTO v_sum FROM threshold_table;
    v_sum := v_sum + :NEW.value_col;
    IF v_sum > v_threshold THEN
        RAISE threshold_exceeded;
    END IF;
EXCEPTION
    WHEN threshold_exceeded THEN
        RAISE_APPLICATION_ERROR(-20003, 'Threshold exceeded for value_col.');
END;
```

OUTPUT:



The screenshot shows a SQL editor interface with the following details:

- Toolbar:** Language (SQL), Rows (10), Clear Command, Find Tables, Save, Run.
- Code Area:** The trigger definition is pasted into the code area. Lines 1 through 17 are numbered on the left.
- Status Bar:** Trigger created. 0.04 seconds.

RESULT:

The trigger is executed successfully.

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.

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `log_changes`:

1. Trigger Firing:

- The trigger fires after each update operation on a row in the `main_table`.

2. Data Retrieval:

- The trigger doesn't explicitly retrieve data from the `main_table` beyond the values available through pseudo-record variables.

3. Audit Trail Entry Creation:

- The trigger defines how a new row will be inserted into the `audit_table`.

4. Populating Audit Table Columns:

- `audit_id`:

- The trigger likely retrieves the next value from a sequence named `audit_seq` (assuming it exists) to generate a unique identifier for the audit record.

- `changed_id`:

- This is set to the `id` of the updated row in the `main_table`, obtained using the pseudo-record variable `:OLD.id`.

- `old_col1, new_col1`:

- These capture the old and new values of the `col1` column from the updated row, accessed using `:OLD.col1` and `:NEW.col1` respectively.

- `old_col2, new_col2`:

- Similar to `col1`, these capture old and new values of the `col2` column using `:OLD.col2` and `:NEW.col2`.

- `change_time`:
 - This is set to the current system timestamp using `SYSTIMESTAMP` to record the exact time of the update.
5. Inserting Audit Entry:
- An `INSERT` statement is executed to insert the constructed new row containing the audit information into the `audit_table`.

6. Trigger Completion:

- After the `INSERT` statement is executed, the trigger logic completes without any additional actions.

Overall Logic:

The trigger intercepts update operations on the `main_table` and leverages pseudo-record variables (`:OLD` and `:NEW`) to access the original and updated values of specific columns (`col1` and `col2`). It then constructs a new row containing this information along with a unique identifier (`audit_id`) and the current timestamp (`change_time`) to be inserted into the `audit_table`. This effectively creates an audit trail for tracking data modifications in the `main_table`.

QUERY:

```
CREATE OR REPLACE TRIGGER log_changes
AFTER UPDATE ON main_table
FOR EACH ROW
BEGIN
    INSERT INTO audit_table (audit_id, changed_id, old_col1, new_col1, old_col2, new_col2,
change_time)
    VALUES (audit_seq.NEXTVAL, :OLD.id, :OLD.col1, :NEW.col1, :OLD.col2, :NEW.col2,
SYSTIMESTAMP);
END;
```

OUTPUT:

The screenshot shows a SQL editor interface with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Undo, Redo, Find, Clear Command, Find Tables.
- SQL Code:**

```
1 CREATE OR REPLACE TRIGGER log_changes
2 AFTER UPDATE ON main_table
3 FOR EACH ROW
4 BEGIN
5   INSERT INTO audit_table (audit_id, changed_id, old_col1, new_col1, old_col2, new_col2, change_time)
6   VALUES (audit_seq.NEXTVAL, :OLD.id, :OLD.col1, :NEW.col1, :OLD.col2, :NEW.col2, SYSTIMESTAMP);
7 END;
8 /
9
```
- Results Tab:** The tab is selected, showing the output of the trigger creation command.
- Output:**

```
Trigger created.
```
- Time:** 0.03 seconds

RESULT:

The trigger is executed successfully.

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.

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `log_user_activity`:

1. Trigger Firing:

- The trigger fires after each insert, update, or delete operation on a row in the `activity_table`.

2. Action Identification:

- The trigger uses an `IF` statement block to determine the type of operation that occurred based on built-in PL/SQL variables:
 - `INSERTING`: True after an insert operation.
 - `UPDATING`: True after an update operation.
 - `DELETING`: True after a delete operation.

3. Log Entry Creation (Conditional):

- Within each `IF` block (depending on the operation type), an `INSERT` statement is used to create a new row in the `user_activity_log` table.
 - `log_id`: Likely retrieved using `activity_log_seq.NEXTVAL` (assuming it's a sequence for generating unique identifiers).
 - `action`: Set to 'INSERT', 'UPDATE', or 'DELETE' based on the operation type.
 - `table_name`: Set to a constant value 'activity_table' (assuming the trigger is specific to this table).
 - `record_id`:
 - For inserts and updates, this is set to the `id` of the new/updated row using `:NEW.id`.
 - For deletes, it's set to the `id` of the deleted row using `:OLD.id` (accessed from the pseudo-record variable).
 - `change_time`: Set to the current system timestamp using `SYSTIMESTAMP` to record the time of the operation.

4. Trigger Completion:

- After the conditional `INSERT` statement (based on the operation type), the trigger logic completes without any further actions.

Overall Logic:

The trigger acts as an auditing mechanism to capture user activity related to the `activity_table`. It leverages built-in PL/SQL variables to identify the operation type and access relevant data (record ID) based on insert, update, or delete. The trigger then constructs and inserts a corresponding log entry into the `user_activity_log` table, providing a detailed record of user interactions with the `activity_table`.

QUERY:

```
CREATE OR REPLACE TRIGGER log_user_activity
```

```

AFTER INSERT OR UPDATE OR DELETE ON activity_table
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
    VALUES (activity_log_seq.NEXTVAL, 'INSERT', 'activity_table', :NEW.id, SYSTIMESTAMP);
  ELSIF UPDATING THEN
    INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
    VALUES (activity_log_seq.NEXTVAL, 'UPDATE', 'activity_table', :NEW.id, SYSTIMESTAMP);
  ELSIF DELETING THEN
    INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
    VALUES (activity_log_seq.NEXTVAL, 'DELETE', 'activity_table', :OLD.id, SYSTIMESTAMP);
  END IF;
END;

```

OUTPUT:

The screenshot shows a SQL editor interface with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Clear Command, Find Tables.
- Toolbar:** Includes icons for Undo, Redo, Search, and others.
- Code Area:**

```

1 CREATE OR REPLACE TRIGGER log_user_activity
2 AFTER INSERT OR UPDATE OR DELETE ON activity_table
3 FOR EACH ROW
4 BEGIN
5   IF INSERTING THEN
6     INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
7     VALUES (activity_log_seq.NEXTVAL, 'INSERT', 'activity_table', :NEW.id, SYSTIMESTAMP);
8   ELSIF UPDATING THEN
9     INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
10    VALUES (activity_log_seq.NEXTVAL, 'UPDATE', 'activity_table', :NEW.id, SYSTIMESTAMP);
11   ELSIF DELETING THEN
12     INSERT INTO user_activity_log (log_id, action, table_name, record_id, change_time)
13     VALUES (activity_log_seq.NEXTVAL, 'DELETE', 'activity_table', :OLD.id, SYSTIMESTAMP);

```
- Results Tab:** Shows the message "Trigger created." and a execution time of "0.03 seconds".

RESULT:

The trigger is executed successfully.

6.) 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

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `update_running_total`:

1. Trigger Firing:
 - The trigger fires before each insert operation on a row in the `running_total_table`.
2. Variable Initialization:
 - A numeric variable `v_total` is declared to store the calculated running total.
3. Retrieving Existing Total (if any):
 - A `SELECT` statement is executed to query the existing total from the `running_total_table`.
 - The `NVL(SUM(amount), 0)` expression calculates the sum of the `amount` column in all existing rows. The `NVL` function ensures `v_total` is set to 0 if the sum is null (no existing records).
4. Calculating Updated Total:
 - The trigger retrieves the `amount` value from the new row being inserted (presumably using a column named `amount`).
 - It calculates the updated running total by adding the new row's `amount` to the `v_total` which holds the existing total (or 0 if no previous records).
5. Updating New Row (if applicable):
 - The trigger modifies the `:NEW.running_total` pseudo-record variable. This variable represents the values being inserted in the new row.
 - It assigns the calculated updated running total (from step 4) to the `:NEW.running_total` pseudo-record, effectively updating the `running_total` value for the new row before insertion.
6. Trigger Completion:
 - After updating the new row's `running_total` (if applicable), the trigger logic completes without any further actions.

Overall Logic:

The trigger intercepts insert operations on the `running_total_table` and calculates an updated running total by considering the `amount` value from the new row being inserted and the potentially existing total from the table. It then updates the `running_total` value in the new row itself (assuming

the table structure doesn't already have a `running_total` column) before the insertion occurs. This ensures the `running_total_table` always reflects the cumulative total amount.

QUERY:

```
CREATE OR REPLACE TRIGGER update_running_total
BEFORE INSERT ON running_total_table
FOR EACH ROW
DECLARE
    v_total NUMBER;
BEGIN
    SELECT NVL(SUM(amount), 0) INTO v_total FROM running_total_table;
    :NEW.running_total := v_total + :NEW.amount;
END;
```

OUTPUT:



The screenshot shows a SQL editor interface with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Undo, Redo, Search, Find Tables
- Code Area:** The code area displays the PL/SQL trigger definition with line numbers 1 through 11.
- Status Bar:** Trigger created.
0.03 seconds

RESULT:

The trigger is executed successfully.

7.) 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

PROCEDURE

Here's the algorithm for the provided PL/SQL trigger `validate_order`:

1. Trigger Firing:
 - The trigger fires before each insert operation on a row in the `orders` table (representing a new order).
2. Variable Declaration:
 - A variable `v_stock` is declared to hold the retrieved stock quantity for the ordered item.
 - A custom exception `insufficient_stock` is declared to signal insufficient stock for the order.
3. Exception Initialization:
 - The `PRAGMA EXCEPTION_INIT` statement associates the `insufficient_stock` exception with a specific error code (`-20004`).
4. Checking Stock Availability:
 - A `SELECT` statement queries the stock quantity for the item being ordered. It retrieves the `stock_quantity` from the `items` table where the `item_id` in the new order row (`NEW.item_id`) matches the item's ID in the `items` table.
 - The retrieved stock quantity is stored in the `v_stock` variable.
5. Order Quantity Validation:
 - An `IF` statement checks if the available stock (`v_stock`) is less than the quantity ordered (`NEW.order_quantity`).
 - If true (insufficient stock):
 - The `insufficient_stock` exception is raised, causing the trigger to move to the exception handling block.
6. Updating Stock Level (if sufficient):
 - This step assumes the `IF` statement condition (insufficient stock) wasn't met, implying sufficient stock is available.
 - An `UPDATE` statement is executed to decrease the stock quantity in the `items` table.

- The `stock_quantity` is reduced by the `NEW.order_quantity` (ordered amount) for the item identified by `NEW.item_id`.

7. Exception Handling:

- The `EXCEPTION` block defines how to handle the `insufficient_stock` exception that might be raised within the trigger logic (step 5).
- When the `insufficient_stock` exception is raised:
 - Another exception (`RAISE_APPLICATION_ERROR`) is raised with a user-defined error code (`-20004`) and an informative message explaining the insufficient stock issue. This prevents order insertion and provides feedback to the user.

8. Trigger Completion:

- If no exceptions are raised (sufficient stock and successful update), the trigger logic completes without any further actions.

Overall Logic:

The trigger enforces a business rule by ensuring sufficient stock exists for an order before creating a new order record. It checks stock availability, updates the stock level if sufficient, and raises an exception with a user-friendly message if there's insufficient stock. This helps maintain data integrity and prevents processing orders that would exceed available inventory.

QUERY:

```

CREATE OR REPLACE TRIGGER validate_order
BEFORE INSERT ON orders
FOR EACH ROW
DECLARE
  v_stock NUMBER;
  insufficient_stock EXCEPTION;
  PRAGMA EXCEPTION_INIT(insufficient_stock, -20004);
BEGIN
  SELECT stock_quantity INTO v_stock FROM items WHERE item_id = :NEW.item_id;
  IF v_stock < :NEW.order_quantity THEN
    RAISE_APPLICATION_ERROR(-20004, 'Insufficient stock for item ' || :NEW.item_id || '.');
  END IF;
END;
  
```

```

    RAISE insufficient_stock;
END IF;
UPDATE items SET stock_quantity = stock_quantity - :NEW.order_quantity WHERE item_id =
:NEW.item_id;
EXCEPTION
WHEN insufficient_stock THEN
    RAISE_APPLICATION_ERROR(-20004, 'Insufficient stock for the item.');
END;

```

OUTPUT:

The screenshot shows a SQL editor window with the following details:

- Language:** SQL
- Rows:** 10
- Buttons:** Save, Run, Clear Command, Find Tables
- Code:**

```

1 CREATE OR REPLACE TRIGGER validate_order
2 BEFORE INSERT ON orders
3 FOR EACH ROW
4 DECLARE
5     v_stock NUMBER;
6     insufficient_stock EXCEPTION;
7     PRAGMA EXCEPTION_INIT(insufficient_stock, -20004);
8 BEGIN
9     SELECT stock_quantity INTO v_stock FROM items WHERE item_id = :NEW.item_id;
10    IF v_stock < :NEW.order_quantity THEN
11        RAISE insufficient_stock;
12    END IF;
13    UPDATE items SET stock_quantity = stock_quantity - :NEW.order_quantity WHERE item_id = :NEW.item_id;
14 EXCEPTION
15     WHEN insufficient_stock THEN

```
- Results Tab:** Trigger created.
- Timing:** 0.05 seconds

RESULT:

The trigger is executed successfully.

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

MONGODB

EX_NO: 19

DATE:22.5.24

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

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({$and:[{ $or: [{cuisine: { $nin: ["American", "Chinese"] }}], {name: /^Wil/}}]}, {restaurant_id: 1, name: 1, borough: 1, cuisine: 1})
[
  {
    _id: ObjectId('6652df15df1fcfc37b3cdcdf6'),
    borough: 'Bronx',
    cuisine: 'Bakery',
    name: 'Morris Park Bake Shop',
    restaurant_id: '30075445'
  }
]
user> |
```

RESULT:

The query is executed successfully.

2.)

```
mongosh mongodb://127.0.0.1:27017/
user> db.restaurants.find({ "grades": { $elemMatch: { "grade": "A", "score": 11, "date": ISODate("2014-08-11T00:00:00Z") } } }, { restaurant_id: 1, name: 1, grades: 1 })
[
  {
    _id: ObjectId('6652e2d7df1fcfc37b3cdcdfa'),
    grades: [
      {
        date: ISODate('2014-08-11T00:00:00.000Z'),
        grade: 'A',
        score: 11
      },
      {
        date: ISODate('2013-09-11T00:00:00.000Z'),
        grade: 'B',
        score: 14
      }
    ],
    name: 'Example Chinese Restaurant',
    restaurant_id: '50012345'
  },
  {
    _id: ObjectId('6652e2d7df1fcfc37b3cdcdfb'),
    grades: [
      {
        date: ISODate('2014-08-11T00:00:00.000Z'),
        grade: 'A',
        score: 11
      },
      {
        date: ISODate('2014-07-10T00:00:00.000Z'),
        grade: 'B',
        score: 12
      }
    ],
    name: 'Sample Italian Bistro',
    restaurant_id: '50012346'
  },
  {
    _id: ObjectId('6652e2d7df1fcfc37b3cdcdfc'),
    grades: [
      {
        date: ISODate('2014-08-11T00:00:00.000Z'),
        grade: 'A',
        score: 11
      },
      {
        date: ISODate('2014-06-15T00:00:00.000Z'),
        grade: 'A',
        score: 8
      }
    ]
  }
]
```

QUERY:

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

OUTPUT:

RESULT:

The query is executed successfully.

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".

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ "grades.1.grade": "A", "grades.1.score": 9, "grades.1.date": ISODate("2014-08-11T00:00:00Z") }, { restaurant_id: 1, name: 1, grades: 1 });
[
  {
    _id: ObjectId('66531e17323c3d16f1cdcdf6'),
    grades: [
      {
        date: ISODate('2014-07-10T00:00:00.000Z'),
        grade: 'B',
        score: 12
      },
      {
        date: ISODate('2014-08-11T00:00:00.000Z'),
        grade: 'A',
        score: 9
      }
    ],
    name: 'Sample Italian Bistro',
    restaurant_id: '50012346'
  }
]
user>
```

RESULT:

The query is executed successfully.

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

QUERY:

```
db.restaurants.find({$and : [{"address.coord.1": {$gt : 42}}, {"address.coord.1": {$lte : 52}}]}, {_id:0, restaurant_id:1, name:1, address:1})
```

OUTPUT:

```

user> db.restaurants.find({$and : [{"address.coord.1": {$gt : 42}}, {"address.coord.1": {$lte : 52}}]}, {_id:0, restaurant_id:1, name:1, address:1})
[{"address: { building: '789', coord: [ -73.856077, 45.848447 ], street: 'Test Blvd', zipcode: '10036' }, name: 'Test Mexican Grill', restaurant_id: '50012347'}, {"address: { building: '789', coord: [ -73.856077, 45.848447 ], street: 'Test Blvd', zipcode: '10036' }, name: 'Test Mexican Grill', restaurant_id: '50012347'}]
user> |

```

RESULT:

The query is executed successfully.

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

QUERY:

```
db.restaurants.find({}, {_id: 0 }).sort({ name: 1 });
```

OUTPUT:

```

user> db.restaurants.find({}, {_id: 0 }).sort({ name: 1 })
[{"address: { building: '123', coord: [ -73.856077, 40.848447 ], street: 'Sample Ave', zipcode: '10123' }, borough: 'Brooklyn', cuisine: 'Italian', grades: [ { date: ISODate('2019-08-11T00:00:00Z'), grade: 'A', score: 11 }, { date: ISODate('2018-07-10T00:00:00Z'), grade: 'B', score: 12 } ], name: 'A Sample Italian Bistro', restaurant_id: '50012349'}, {"address: { building: '400', coord: [ -73.984381, 40.759145 ], street: 'Broadway', zipcode: '10036' }, borough: 'Manhattan', cuisine: 'Steakhouse', grades: [ { date: ISODate('2014-03-03T00:00:00Z'), grade: 'A', score: 9 }, { date: ISODate('2013-09-11T00:00:00Z'), grade: 'A', score: 8 } ], name: 'Broadway Steakhouse', restaurant_id: '40356402'}, {"address: { building: '101', coord: [ -73.927, 40.829 ], street: 'Demo Rd', zipcode: '10451' },

```

RESULT:

The query is executed successfully.

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

QUERY:

```
db.restaurants.find({}, { _id: 0 }).sort({ name: -1 })
```

OUTPUT:

```
user> db.restaurants.find({}, { _id: 0 }).sort({ name: -1 })
[{"_id": "5e012350", "name": "Zebra Mexican Grill", "grades": [{"date": ISODate('2022-08-11T00:00:00Z'), "grade": "A", "score": 10}, {"date": ISODate('2021-07-10T00:00:00Z'), "grade": "B", "score": 15}], "address": {"building": "456", "coord": [-73.856077, 40.848447], "street": "Last St", "zipcode": "10001"}, "borough": "Manhattan", "cuisine": "Mexican"}, {"_id": "5e012347", "name": "Test Mexican Grill", "grades": [{"date": ISODate('2014-08-11T00:00:00Z'), "grade": "A", "score": 11}, {"date": ISODate('2014-06-15T00:00:00Z'), "grade": "A", "score": 8}], "address": {"building": "789", "coord": [-73.980381, 40.759145], "street": "Test Blvd", "zipcode": "10036"}, "borough": "Manhattan", "cuisine": "Mexican"}, {"_id": "5e012346", "name": "Test Mexican Grill", "grades": [{"date": ISODate('2014-08-11T00:00:00Z'), "grade": "A", "score": 11}, {"date": ISODate('2014-06-15T00:00:00Z'), "grade": "A", "score": 8}], "address": {"building": "789", "coord": [-73.980381, 40.759145], "street": "Test Blvd", "zipcode": "10036"}]
```

RESULT:

The query is executed successfully.

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

QUERY:

```
db.restaurants.find({}, { _id: 0 }).sort({ cuisine: 1, borough: -1 })
```

OUTPUT:

```

user> db.restaurants.find({}, { _id: 0 }).sort({ cuisine: 1, borough: -1 })
[ {
  address: {
    building: '1007',
    coord: [ -73.856077, 40.848447 ],
    street: 'Morris Park Ave',
    zipcode: '10462'
  },
  borough: 'Bronx',
  cuisine: 'Bakery',
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: 'A',
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: 'A',
      score: 9
    },
    {
      date: ISODate('2011-03-18T00:00:00.000Z'),
      grade: 'B',
      score: 14
    }
  ],
  name: 'Morris Park Bake Shop',
  restaurant_id: '30075445'
},
{
  address: {
    building: '101',
    coord: [ -73.927, 40.829 ],
    street: 'Domo Rd',
    zipcode: '10451'
  },
  borough: 'Bronx',
  cuisine: 'Bakery',
  grades: [
    {
      date: ISODate('2014-08-11T00:00:00.000Z'),
      grade: 'A',
      score: 11
    }
  ]
}
]

```

RESULT:

The query is executed successfully.

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

QUERY:

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

OUTPUT:

```

user> db.restaurants.find({ "address.street": { $exists: true, $ne: "" } })
[ {
  _id: ObjectId('6652df15df1fcfc37b3cdcdf6'),
  address: {
    building: '1007',
    coord: [ -73.856077, 40.848447 ],
    street: 'Morris Park Ave',
    zipcode: '10462'
  },
  borough: 'Bronx',
  cuisine: 'Bakery',
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: 'A',
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: 'A',
      score: 9
    },
    {
      date: ISODate('2011-03-18T00:00:00.000Z'),
      grade: 'B',
      score: 14
    }
  ],
  name: 'Morris Park Bake Shop',
  restaurant_id: '30075445'
},
{
  _id: ObjectId('6652e1addf1fcfc37b3cdcdf7'),
  address: {
    building: '200',
    coord: [ -73.961704, 40.662942 ],
    street: 'Flatbush Ave',
    zipcode: '11225'
  },
  borough: 'Brooklyn',
  cuisine: 'Pizza',
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 10
    }
  ]
}
]

```

RESULT:

The query is executed successfully.

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

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ "address.coord": { $elemMatch: { $type: "double" } } })
[{"_id": ObjectId("6652df15df1fc37b3cdcdf6"),
  address: {
    building: "1087",
    coord: [-73.856077, 40.848447],
    street: "Morris Park Ave",
    zipcode: "10462"
  },
  borough: "Bronx",
  cuisine: "Bakery",
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: "A",
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: "A",
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: "A",
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: "A",
      score: 9
    },
    {
      date: ISODate('2011-03-10T00:00:00.000Z'),
      grade: "B",
      score: 14
    }
  ],
  name: "Morris Park Bake Shop",
  restaurant_id: "30075045"
},
{"_id": ObjectId("6652e1addf1fc37b3cdcdf7"),
  address: {
    building: "200",
    coord: [-73.961704, 40.662942],
    street: "Flatbush Ave",
    zipcode: "11225"
  },
  borough: "Brooklyn",
  cuisine: "Pizza",
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: "B"
    }
  ]
}]
```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```

user> db.restaurants.find({ "grades.score": { $mod: [7, 0] } }, { restaurant_id: 1, name: 1, grades: 1 })
[{
  _id: ObjectId('6652df15df1cfc37b3cdcdf6'),
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: 'A',
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: 'A',
      score: 9
    },
    {
      date: ISODate('2011-03-10T00:00:00.000Z'),
      grade: 'B',
      score: 14
    }
  ],
  name: 'Morris Park Bake Shop',
  restaurant_id: '30075045'
},
{
  _id: ObjectId('6652e1c2df1cfc37b3cdcdf9'),
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 7
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 8
    }
  ],
  name: 'Broadway Steakhouse',
  restaurant_id: '40356042'
},
{
  _id: ObjectId('6652e2d7df1cfc37b3cdcdfa'),
  grades: [
    {
      date: ISODate('2014-08-11T00:00:00.000Z'),
      grade: 'A',
      score: 10
    }
  ]
}
]

```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```

user> db.restaurants.find({ name: /mon/i }, { name: 1, borough: 1, "address.coord": 1, cuisine: 1 })
[{
  _id: ObjectId('66530613323c3d16f1cdcdf'),
  address: { coord: [ -73.856077, 40.848447 ] },
  borough: 'Manhattan',
  cuisine: 'French',
  name: 'Mon Amour Bistro'
}]
user>

```

RESULT:

The query is executed successfully.

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

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ name: /"Mad/i }, { name: 1, borough: 1, "address.coord": 1, cuisine: 1 })
[
  {
    _id: ObjectId('665326af923c3d16f1cdce00'),
    address: { coord: [-73.856077, 40.848447] },
    borough: 'Brooklyn',
    cuisine: 'American',
    name: 'Madison Grill'
  }
]
user>
```

RESULT:

The query is executed successfully.

13. Write a MongoDB query to find the restaurants that have at least one grade with a score of less than 5.

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ "grades": { $elemMatch: { "score": { $lt: 5 } } } })
[
  {
    _id: ObjectId('6652df15df1fc97b3cdcdf6'),
    address: {
      building: '1007',
      coord: [-73.856077, 40.848447],
      street: 'Morris Park Ave',
      zipcode: '10662'
    },
    borough: 'Bronx',
    cuisine: 'Bakery',
    grades: [
      {
        date: ISODate('2014-03-03T00:00:00Z'),
        grade: 'A',
        score: 2
      },
      {
        date: ISODate('2013-09-11T00:00:00Z'),
        grade: 'A',
        score: 6
      },
      {
        date: ISODate('2013-01-24T00:00:00Z'),
        grade: 'A',
        score: 18
      },
      {
        date: ISODate('2011-11-23T00:00:00Z'),
        grade: 'A',
        score: 9
      },
      {
        date: ISODate('2011-03-18T00:00:00Z'),
        grade: 'B',
        score: 14
      }
    ],
    name: 'Morris Park Bake Shop',
    restaurant_id: '30075445'
  },
  {
    _id: ObjectId('665326fb923c3d16f1cdce01'),
    address: {
      building: '789',
      coord: [-73.856077, 40.848447],
      street: 'Example Street',
      zipcode: '10005'
    },
    borough: 'Queens',
    cuisine: 'Japanese',
    grades: [
      {
        date: ISODate('2023-05-10T00:00:00Z'),
        grade: 'E'
      }
    ]
  }
]
```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```

user> db.restaurants.find({ "grades": { $elemMatch: { "score": { $lt: 5 } } }, "borough": "Manhattan" })
[ {
  _id: ObjectId('665327af323c3d16f1cdce02'),
  address: {
    building: '123',
    coord: [-73.856077, 40.848447],
    street: 'Example Street',
    zipcode: '10006'
  },
  borough: 'Manhattan',
  cuisine: 'American',
  grades: [
    {
      date: ISODate('2023-05-10T00:00:00.000Z'),
      grade: 'C',
      score: 3
    },
    {
      date: ISODate('2022-03-15T00:00:00.000Z'),
      grade: 'B',
      score: 7
    }
  ],
  name: 'Manhattan Deli',
  restaurant_id: '50012357'
}
]
user>

```

RESULT:

The query is executed successfully.

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.

QUERY:

```

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

```

OUTPUT:

```

user> db.restaurants.find({ "grades": { $elemMatch: { "score": { $lt: 5 } } }, $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }] })
[ {
  _id: ObjectId('665327af323c3d16f1cdce02'),
  address: {
    building: '123',
    coord: [-73.856077, 40.848447],
    street: 'Example Street',
    zipcode: '10006'
  },
  borough: 'Manhattan',
  cuisine: 'American',
  grades: [
    {
      date: ISODate('2023-05-10T00:00:00.000Z'),
      grade: 'C',
      score: 3
    },
    {
      date: ISODate('2022-03-15T00:00:00.000Z'),
      grade: 'B',
      score: 7
    }
  ],
  name: 'Manhattan Deli',
  restaurant_id: '50012357'
},
{
  _id: ObjectId('66532836323c3d16f1cdce03'),
  address: {
    building: '156',
    coord: [-73.856077, 40.848447],
    street: 'Sample Street',
    zipcode: '10007'
  },
  borough: 'Brooklyn',
  cuisine: 'Italian',
  grades: [
    {
      date: ISODate('2023-05-10T00:00:00.000Z'),
      grade: 'C',
      score: 3
    },
    {
      date: ISODate('2022-03-15T00:00:00.000Z'),
      grade: 'B',
      score: 7
    }
  ],
  name: 'Brooklyn Pizza',
  restaurant_id: '50012358'
}
]
user>

```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ "grades": { $elemMatch: { "score": { $lt: 5 } } }, $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }], "cuisine": { $ne: "American" } })
[{"_id": ObjectId('66532836323c3d16f1cdce03'), "address": {"building": '1456', "coord": [-73.856077, 40.848447], "street": 'Sample Street', "zipcode": '10007'}, "borough": 'Brooklyn', "cuisine": 'Italian', "grades": [{"date": ISODate('2023-05-10T00:00:00.000Z'), "grade": 'C', "score": 3}, {"date": ISODate('2022-03-15T00:00:00.000Z'), "grade": 'B', "score": 7}], "name": 'Brooklyn Pizza', "restaurant_id": '50012358'}, {"_id": ObjectId('665328bf323c3d16f1cdce04'), "address": {"building": '789', "coord": [-73.856077, 40.848447], "street": 'Sample Avenue', "zipcode": '10008'}, "borough": 'Manhattan', "cuisine": 'Italian', "grades": [{"date": ISODate('2023-05-10T00:00:00.000Z'), "grade": 'C', "score": 3}, {"date": ISODate('2022-03-15T00:00:00.000Z'), "grade": 'B', "score": 7}], "name": 'Manhattan Pasta', "restaurant_id": '50012359'}]
user> |
```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```

user> db.restaurants.find({ "grades": { $elemMatch: { "score": { $lt: 5 } } } }, { $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }], "cuisine": { $in: ["American", "Chinese"] } })
[ {
  _id: ObjectId('66532836323c3d16f1cdce03'),
  address: {
    building: '456',
    coord: [ -73.856077, 40.848447 ],
    street: 'Sample Street',
    zipcode: '10007'
  },
  borough: 'Brooklyn',
  cuisine: 'Italian',
  grades: [
    {
      date: ISODate('2023-05-10T00:00:00.000Z'),
      grade: 'C',
      score: 3
    },
    {
      date: ISODate('2022-03-15T00:00:00.000Z'),
      grade: 'B',
      score: 7
    }
  ],
  name: 'Brooklyn Pizza',
  restaurant_id: '50012358'
},
{
  _id: ObjectId('6653280f323c3d16f1cdce04'),
  address: {
    building: '789',
    coord: [ -73.856077, 40.848447 ],
    street: 'Sample Avenue',
    zipcode: '10008'
  },
  borough: 'Manhattan',
  cuisine: 'Italian',
  grades: [
    {
      date: ISODate('2023-05-10T00:00:00.000Z'),
      grade: 'C',
      score: 3
    },
    {
      date: ISODate('2022-03-15T00:00:00.000Z'),
      grade: 'B',
      score: 7
    }
  ],
  name: 'Manhattan Pasta',
  restaurant_id: '50012359'
},
{
  _id: ObjectId('66532948323c3d16f1cdce05'),
  address: {

```

RESULT:

The query is executed successfully.

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.

QUERY:

```
db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }] })
```

OUTPUT:

```

[ {
  _id: ObjectId('664f3c798752f54dc3cdcf7'),
  address: {
    building: '1007',
    coord: [ -73.856077, 40.848447 ],
    street: 'Morris Park Ave',
    zipcode: '10462'
  },
  borough: 'Bronx',
  cuisine: 'Bakery',
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: 'A',
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: 'A',
      score: 9
    },
    {
      date: ISODate('2011-03-10T00:00:00.000Z'),
      grade: 'B',
      score: 14
    }
  ],
  name: 'Morris Park Bake Shop',
  restaurant_id: '50075445'
}
```

RESULT:

The query is executed successfully.

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.

QUERY:

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

OUTPUT:

```
user> db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], "borough": "Manhattan" })
[
  {
    _id: ObjectId('6654323a0bcc80d32cdcdf6'),
    name: 'Example Restaurant',
    cuisine: 'American',
    borough: 'Manhattan',
    address: {
      street: '123 Example St',
      zipcode: '10001',
      building: '100',
      coord: [ -73.856077, 40.848447 ]
    },
    grades: [
      {
        date: ISODate('2023-05-01T00:00:00.000Z'),
        grade: 'A',
        score: 2
      },
      {
        date: ISODate('2023-06-01T00:00:00.000Z'),
        grade: 'A',
        score: 6
      }
    ],
    _id: ObjectId('665432ad0bcc80d32cdcdf7'),
    name: 'Sample Restaurant',
    cuisine: 'Italian',
    borough: 'Manhattan',
    address: {
      street: '456 Sample Rd',
      zipcode: '10002',
      building: '200',
      coord: [ -73.955741, 40.78496 ]
    },
    grades: [
      {
        date: ISODate('2024-01-15T00:00:00.000Z'),
        grade: 'A',
        score: 2
      },
      {
        date: ISODate('2024-02-15T00:00:00.000Z'),
        grade: 'A',
        score: 6
      }
    ]
  }
]
user> |
```

RESULT:

The query is executed successfully.

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.

QUERY:

```
db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }] })
```

OUTPUT:

```

user> db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }] })
[{
  _id: ObjectId('6654323a0bcc80d32cdcdf6'),
  name: 'Example Restaurant',
  cuisine: 'American',
  borough: 'Manhattan',
  address: {
    street: '123 Example ST',
    zipcode: '10001',
    building: '100',
    coord: [ -73.866077, 40.848447 ]
  },
  grades: [
    {
      date: ISODate('2023-05-01T00:00:00Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2023-06-01T00:00:00Z'),
      grade: 'A',
      score: 6
    }
  ],
  _id: ObjectId('665432ad0bcc80d32cdcdf7'),
  name: 'Sample Restaurant',
  cuisine: 'Italian',
  borough: 'Manhattan',
  address: {
    street: '456 Sample Rd',
    zipcode: '10002',
    building: '200',
    coord: [ -73.955701, 40.78496 ]
  },
  grades: [
    {
      date: ISODate('2024-01-15T00:00:00Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2024-02-15T00:00:00Z'),
      grade: 'A',
      score: 6
    }
  ],
  _id: ObjectId('665433020bcc80d32cdcdf8'),
  name: 'Example Restaurant',
  cuisine: 'Italian',
  borough: 'Manhattan',
}
]

```

RESULT:

The query is executed successfully.

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.

QUERY:

```
db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }], "cuisine": { $ne: "American" } })
```

OUTPUT:

```

user> db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }], "cuisine": { $ne: "American" } })
[
  {
    _id: ObjectId('665432ad8bcc80d32cdcdf7'),
    name: 'Sample Restaurant',
    cuisine: 'Italian',
    borough: 'Manhattan',
    address: {
      street: '123 Sample Rd',
      zipcode: '10002',
      building: '200',
      coord: [ -73.955741, 40.78496 ]
    },
    grades: [
      {
        date: ISODate('2024-01-15T00:00:00Z'),
        grade: 'A',
        score: 2
      },
      {
        date: ISODate('2024-02-15T00:00:00Z'),
        grade: 'A',
        score: 6
      }
    ],
    _id: ObjectId('665433020bcc80d32cdcdf8'),
    name: 'Example Restaurant',
    cuisine: 'Italian',
    borough: 'Manhattan',
    address: {
      street: '789 Example Ave',
      zipcode: '10003',
      building: '300',
      coord: [ -73.991069, 40.73061 ]
    },
    grades: [
      {
        date: ISODate('2024-03-01T00:00:00Z'),
        grade: 'A',
        score: 2
      },
      {
        date: ISODate('2024-04-01T00:00:00Z'),
        grade: 'A',
        score: 6
      }
    ],
    _id: ObjectId('665433580bcc80d32cdcdf9'),
    name: 'Example Restaurant',
    cuisine: 'Italian',
    borough: 'Brooklyn',
  }
]

```

RESULT:

The query is executed successfully.

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.

QUERY:

```

db.restaurants.find({ $and: [{ "grades.grade": "A", "grades.score": 2 }, { "grades.grade": "A", "grades.score": 6 }], $or: [{ "borough": "Manhattan" }, { "borough": "Brooklyn" }], "cuisine": { $nin: ["American", "Chinese"] } })

```

OUTPUT:

RESULT:

The query is executed successfully.

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.

QUERY:

```

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

```

OUTPUT:

```
{
  _id: ObjectId('664f3c798752f54dc3cdcdf7'),
  address: {
    building: '1007',
    coord: [ -73.856077, 40.848447 ],
    street: 'Morris Park Ave',
    zipcode: '10462'
  },
  borough: 'Bronx',
  cuisine: 'Bakery',
  grades: [
    {
      date: ISODate('2014-03-03T00:00:00.000Z'),
      grade: 'A',
      score: 2
    },
    {
      date: ISODate('2013-09-11T00:00:00.000Z'),
      grade: 'A',
      score: 6
    },
    {
      date: ISODate('2013-01-24T00:00:00.000Z'),
      grade: 'A',
      score: 10
    },
    {
      date: ISODate('2011-11-23T00:00:00.000Z'),
      grade: 'A',
      score: 9
    },
    {
      date: ISODate('2011-03-10T00:00:00.000Z'),
      grade: 'B',
      score: 14
    }
  ],
  name: 'Morris Park Bake Shop',
  restaurant_id: '50075445'
}
```

RESULT:

The query is executed successfully.

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

MONGODB

EX_NO: 20

DATE:22.5.24

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

QUERY:

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

OUTPUT:

```
user> db.movies.find({ year: 1893 })
[
  {
    _id: ObjectId('665434940bcc80d32cdcdfb'),
    title: 'Sample Movie',
    year: 1893,
    genre: 'Drama',
    director: 'Sample Director',
    actors: [ 'Actor 1', 'Actor 2' ],
    plot: 'This is a sample plot for the movie.',
    awards: { wins: 5, nominations: 10 }
  }
]
user> |
```

RESULT:

The query is executed successfully.

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

QUERY:

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

OUTPUT:

```
user> db.movies.find({ runtime: { $gt: 120 } })
[
  {
    _id: ObjectId('665434db0bcc80d32cdcdfe'),
    title: 'Sample Movie',
    year: 2023,
    genre: 'Action',
    director: 'Sample Director',
    actors: [ 'Actor A', 'Actor B', 'Actor C' ],
    plot: 'This is a sample plot for the movie.',
    runtime: 130
  }
]
user> |
```

RESULT:

The query is executed successfully.

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

QUERY:

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

OUTPUT:

```

user> db.movies.find({ genres: 'Short' })
[
  {
    _id: ObjectId('573a1390f29313caabcd42e8'),
    plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on their heels.',
    genres: [ 'Short', 'Western' ],
    runtime: 11,
    cast: [
      'A.C. Abadie',
      'Gilbert M. 'Broncho Billy' Anderson',
      'George Barnes',
      'Justus D. Barnes'
    ],
    poster: 'https://m.media-amazon.com/images/M/MV5BMTU3NjE5NzYtYTtyNS00MDVmLWIwYjgtMnYwYWIxZDVyNzU2XkEyXkFqcGdeQXVyNzQzNzQxNzI0..V1_SY1000_SK677_AL_.jpg',
    title: 'The Great Train Robbery',
    fullplot: 'Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.',
    languages: [ 'English' ],
    released: ISODate('1903-12-01T00:00:00.000Z'),
    directors: [ 'Edwin S. Porter' ],
    rated: 'TV-G',
    awards: { wins: 1, nominations: 0, text: '1 win.' },
    lastupdated: '2015-08-13 00:27:59.177000000',
    year: 1903,
    imdb: { rating: 7.4, votes: 9847, id: 439 },
    countries: [ 'USA' ],
    type: 'Movie',
    tomatoes: {
      viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
      fresh: 6,
      critic: { rating: 7.6, numReviews: 6, meter: 100 },
      rotten: 0,
      lastUpdated: ISODate('2015-08-08T19:16:10.000Z')
    }
  },
  {
    _id: ObjectId('665435100bcc80d32cdcdfd'),
    title: 'Sample Short Film',
    year: 2024,
    genres: [ 'short' ],
    director: 'Sample Director',
    actors: [ 'Actor X', 'Actor Y' ],
    plot: 'This is a sample plot for the short film.'
  }
]
user>

```

RESULT:

The query is executed successfully.

4.) Retrieve all movies from the 'movies' collection that were directed by William K.L. Dickson and include complete information for each movie.

QUERY:

```
db.movies.find({ directors: 'William K.L. Dickson' })
```

OUTPUT:

```

user> db.movies.find({ directors: 'William K.L. Dickson' })
[
  {
    _id: ObjectId('6654355d0bhccc80d32cdcdfd'),
    title: 'Sample Movie',
    year: 1894,
    genres: [ 'Short', 'Documentary' ],
    director: [ 'William K.L. Dickson' ],
    actors: [ 'Actor A', 'Actor B' ],
    plot: 'This is a sample plot for the movie directed by William K.L. Dickson.'
  }
]
user>

```

RESULT:

The query is executed successfully.

5.) Retrieve all movies from the 'movies' collection that were released in the USA and include complete information for each movie.

QUERY:

```
db.movies.find({ countries: 'USA' })
```

OUTPUT:

```

user> db.movies.find({ countries: 'USA' })
[
  {
    _id: ObjectId('573a1390f29313caabcd42e8'),
    plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on their heels.',
    genres: [ 'Short', 'Western' ],
    runtime: 11,
    cast: [
      'A.C. Abadie',
      'Gilbert M. 'Broncho Billy' Anderson',
      'George Barnes',
      'Justus D. Barnes'
    ],
    poster: 'https://m.media-amazon.com/images/M/MVSBMTU3NjE8NzYLyTyyNS80NDVmLWImYjgtHmYwYIxDYyNzU2XkEyXkFqcGdeQxVvHzQzNzQxNzI@._V1_SY1000_SX677_AL_.jpg',
    title: 'The Great Train Robbery',
    fullplot: "Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",
    passengers: "They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",
    languages: [ 'English' ],
    released: ISODate('1903-12-01T00:00:00Z'),
    directors: [ 'Edwin S. Porter' ],
    rated: 'TV-G',
    awards: { wins: 1, nominations: 0, text: '1 win.' },
    lastUpdated: '2015-08-13 08:27:59.177000000',
    year: 1903,
    imdb: { rating: 7.4, votes: 9847, id: 439 },
    countries: [ 'USA' ],
    type: 'movie',
    tomatoes: {
      viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
      fresh: 6,
      critic: { rating: 7.6, numReviews: 6, meter: 100 },
      rotten: 0,
      lastUpdated: ISODate('2015-08-08T19:16:10.000Z')
    }
  },
  {
    _id: ObjectId('665435a10bcc80d32cdcff'),
    title: 'Sample Movie',
    year: 2624,
    genres: [ 'Drama', 'Thriller' ],
    directors: [ 'Sample Director' ],
    actors: [ 'Actor X', 'Actor Y' ],
    countries: [ 'USA' ],
    plot: 'This is a sample plot for the movie set in the USA.'
  }
]
user> |

```

RESULT:

The query is executed successfully.

6.) Retrieve all movies from the 'movies' collection that have complete information and are rated as "UNRATED".

QUERY:

```
db.movies.find({ rated: 'UNRATED' })
```

OUTPUT:

```

user> db.movies.find({ rated: 'UNRATED' })
[
  {
    _id: ObjectId('665435d60bcc80d32cdce00'),
    title: 'Sample Movie',
    year: 2625,
    genres: [ 'Comedy', 'Drama' ],
    directors: [ 'Sample Director' ],
    actors: [ 'Actor A', 'Actor B' ],
    rated: 'UNRATED',
    plot: 'This is a sample plot for the unrated movie.'
  }
]
user> |

```

RESULT:

The query is executed successfully.

7.) Retrieve all movies from the 'movies' collection that have complete information and have received more than 1000 votes on IMDb.

QUERY:

```
db.movies.find({ 'imdb.votes': { $gt: 1000 } })
```

OUTPUT:

```

user> db.movies.find({ 'imdb.votes': { $gt: 1000 } })
[ {
  _id: ObjectId('573a1390ef29313caabcc42e8'),
  plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on their heels.',
  genres: [ 'Short', 'Western' ],
  runtime: 11,
  cast: [
    'A.C. Abadie',
    'Gilbert M. 'Broncho Billy' Anderson',
    'George Barnes',
    'Justus D. Barnes'
  ],
  poster: 'https://m.media-amazon.com/images/M/MV5BMTU3NjE5NzYtYTyNS00MDVmLWIwYjgtMmYwYWIxZDyNzU2XkEyXkFqcGdeQXvNyNzQzNzQzI@._V1_SV1000_SX677_AL_.jpg',
  title: 'The Great Train Robbery',
  fullplot: "Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",
  languages: [ 'English' ],
  released: ISODate('1903-12-01T00:00:00.000Z'),
  directors: [ 'Edwin S. Porter' ],
  rated: 'TV-G',
  awards: { wins: 1, nominations: 0, text: '1 win.' },
  lastupdated: '2015-08-13 00:27:59.177000000',
  year: 1903,
  imdb: { rating: 7.4, votes: 9847, id: 439 },
  countries: [ 'USA' ],
  type: 'movie',
  tomatoes: {
    viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
    fresh: 6,
    critic: { rating: 7.6, numReviews: 6, meter: 100 },
    rotten: 0,
    lastupdated: ISODate('2015-08-08T19:16:10.000Z')
  }
},
{
  _id: ObjectId('665436090bcc88d32cdce01'),
  title: 'Sample Movie',
  year: 2026,
  genres: [ 'Action', 'Adventure' ],
  directors: [ 'Sample Director' ],
  actors: [ 'Actor X', 'Actor Y' ],
  imdb: { rating: 7.5, votes: 1200 },
  plot: 'This is a sample plot for the movie with more than 1000 IMDb votes.'
}
]
user> |

```

RESULT:

The query is executed successfully.

8.) Retrieve all movies from the 'movies' collection that have complete information and have an IMDb rating higher than 7.

QUERY:

```
db.movies.find({ 'imdb.rating': { $gt: 7 } })
```

OUTPUT:

```

user> db.movies.find({ 'imdb.rating': { $gt: 7 } })
[ {
  _id: ObjectId('573a1390ef29313caabcc42e8'),
  plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on their heels.',
  genres: [ 'Short', 'Western' ],
  runtime: 11,
  cast: [
    'A.C. Abadie',
    'Gilbert M. 'Broncho Billy' Anderson',
    'George Barnes',
    'Justus D. Barnes'
  ],
  poster: 'https://m.media-amazon.com/images/M/MV5BMTU3NjE5NzYtYTyNS00MDVmLWIwYjgtMmYwYWIxZDyNzU2XkEyXkFqcGdeQXvNyNzQzNzQzI@._V1_SV1000_SX677_AL_.jpg',
  title: 'The Great Train Robbery',
  fullplot: "Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",
  languages: [ 'English' ],
  released: ISODate('1903-12-01T00:00:00.000Z'),
  directors: [ 'Edwin S. Porter' ],
  rated: 'TV-G',
  awards: { wins: 1, nominations: 0, text: '1 win.' },
  lastupdated: '2015-08-13 00:27:59.177000000',
  year: 1903,
  imdb: { rating: 7.4, votes: 9847, id: 439 },
  countries: [ 'USA' ],
  type: 'movie',
  tomatoes: {
    viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
    fresh: 6,
    critic: { rating: 7.6, numReviews: 6, meter: 100 },
    rotten: 0,
    lastupdated: ISODate('2015-08-08T19:16:10.000Z')
  }
},
{
  _id: ObjectId('665436090bcc88d32cdce01'),
  title: 'Sample Movie',
  year: 2026,
  genres: [ 'Action', 'Adventure' ],
  directors: [ 'Sample Director' ],
  actors: [ 'Actor X', 'Actor Y' ],
  imdb: { rating: 7.5, votes: 1200 },
  plot: 'This is a sample plot for the movie with more than 1000 IMDb votes.'
},
{
  _id: ObjectId('665436090bcc88d32cdce02'),
  title: 'Sample High Rated Movie',
  year: 2027,
  genres: [ 'Drama', 'Romance' ],
  directors: [ 'Sample Director' ],
  actors: [ 'Actor A', 'Actor B' ],
  imdb: { rating: 7.5, votes: 1500 },
  plot: 'This is a sample plot for a highly rated movie with an IMDb rating greater than 7.'
}
]

```

RESULT:

The query is executed successfully.

9.) Retrieve all movies from the 'movies' collection that have complete information and have a viewer rating higher than 4 on Tomatoes.

QUERY:

```
db.movies.find({ 'tomatoes.viewer.rating': { $gt: 4 } })
```

OUTPUT:

```
user> db.movies.find({ 'tomatoes.viewer.rating': { $gt: 4 } })
[ {
  _id: ObjectId('665436950bccc88d32cdce03'),
  title: 'Sample Movie',
  year: 2028,
  genres: [ 'Comedy', 'Adventure' ],
  directors: [ 'Sample Director' ],
  actors: [ 'Actor X', 'Actor Y' ],
  tomatoes: { viewer: { rating: 4.5 } },
  plot: 'This is a sample plot for the movie with a Rotten Tomatoes viewer rating greater than 4.'
}
]
user> |
```

RESULT:

The query is executed successfully.

10.) Retrieve all movies from the 'movies' collection that have received an award.

QUERY:

```
db.movies.find({ 'awards.wins': { $gt: 0 } })
```

OUTPUT:

```
user> db.movies.find({ 'awards.wins': { $gt: 0 } })
[ {
  _id: ObjectId('573a1390f29313caabed42e8'),
  plot: 'A group of bandits stage a brazen train hold-up, only to find a determined posse hot on their heels.',
  genres: [ 'Short', 'Western' ],
  runtime: 11,
  cast: [
    'A.C. Abadie',
    'Gilbert M. 'Broncho Billy' Anderson',
    'George Barnes',
    'Justus D. Barnes'
  ],
  poster: 'https://m.media-amazon.com/images/M/MV5BNTU3NjESNzYtYTYYNS00MDVmLWIwYjgtNmVnYIxZDyNzU2XkEyXkFqcGdeQXVyNzQzhzQxNzI@._V1_SX677_AL_.jpg',
  title: 'The Great Train Robbery',
  fullplot: "Among the earliest existing films in American cinema - notable as the first film that presented a narrative story to tell - it depicts a group of cowboy outlaws who hold up a train and rob the passengers. They are then pursued by a Sheriff's posse. Several scenes have color included - all hand tinted.",
  languages: [ 'English' ],
  released: ISODate('1903-12-01T00:00:00.000Z'),
  directors: [ 'Edwin S. Porter' ],
  rated: 'TV-G',
  awards: { wins: 1, nominations: 0, text: '1 win.' },
  lastupdated: '2015-08-13 00:27:59.177000000',
  year: 1903,
  imdb: { rating: 7.4, votes: 9847, id: 439 },
  countries: [ 'USA' ],
  type: 'movie',
  tomatoes: {
    viewer: { rating: 3.7, numReviews: 2559, meter: 75 },
    fresh: 6,
    critic: { rating: 7.6, numReviews: 6, meter: 100 },
    rotten: 0,
    lastupdated: ISODate('2015-08-08T19:16:10.000Z')
  }
},
{
  _id: ObjectId('66543480bccc88d32cdcfb'),
  title: 'Sample Movie',
  year: 1893,
  genre: 'Drama',
  director: 'Sample Director',
  actors: [ 'Actor 1', 'Actor 2' ],
  plot: 'This is a sample plot for the movie.',
  awards: { wins: 5, nominations: 10 }
},
{
  _id: ObjectId('665436df0bccc88d32cdce04'),
  title: 'Sample Award-Winning Movie',
  year: 2029,
  genres: [ 'Drama', 'Biography' ],
  directors: [ 'Sample Director' ],
  actors: [ 'Actor A', 'Actor B' ],
  awards: { wins: 3, nominations: 5 },
  plot: 'This is a sample plot for the award-winning movie with wins greater than 0.'
}
```

RESULT:

The query is executed successfully.

11.) Find all movies with title, languages, released, directors, writers, awards, year, genres, runtime, cast, countries from the 'movies' collection in MongoDB that have at least one nomination.

QUERY:

```
db.movies.find( { 'awards.nominations': { $gt: 0 } }, { title: 1, languages: 1, released: 1, directors: 1, writers: 1, awards: 1, year: 1, genres: 1, runtime: 1, cast: 1, countries: 1 } )
```

OUTPUT:

```
user> db.movies.find(
...   { 'awards.nominations': { $gt: 0 } },
...   { title: 1, languages: 1, released: 1, directors: 1, writers: 1, awards: 1, year: 1, genres: 1, runtime: 1, cast: 1, countries: 1 }
... )
[
  {
    _id: ObjectId('665434840bcc880d32cdcdfb'),
    title: 'Sample Movie',
    year: 1892,
    awards: { wins: 5, nominations: 10 }
  },
  {
    _id: ObjectId('665436df0bcc880d32cdce04'),
    title: 'Sample Award-Winning Movie',
    year: 2020,
    genres: [ 'Drama', 'Biography' ],
    directors: [ 'Sample Director' ],
    awards: { wins: 3, nominations: 5 }
  },
  {
    _id: ObjectId('6654371d0bcc880d32cdce05'),
    title: 'Sample Nominated Movie',
    year: 2030,
    languages: [ 'English' ],
    released: ISODate('2030-06-15T00:00:00Z'),
    directors: [ 'Sample Director' ],
    writers: [ 'Writer A', 'Writer B' ],
    awards: { wins: 0, nominations: 2 },
    genres: [ 'Drama', 'Mystery' ],
    runtime: 120,
    cast: [ 'Actor A', 'Actor B', 'Actor C' ],
    countries: [ 'USA' ]
  }
]
user> |
```

RESULT:

The query is executed successfully.

12.) Find all movies with title, languages, released, directors, writers, awards, year, genres, runtime, cast, countries from the 'movies' collection in MongoDB with cast including "Charles Kayser".

QUERY:

```
db.movies.find( { cast: 'Charles Kayser' }, { title: 1, languages: 1, released: 1, directors: 1, writers: 1, awards: 1, year: 1, genres: 1, runtime: 1, cast: 1, countries: 1 } )
```

OUTPUT:

```
user> db.movies.find(
...   { cast: 'Charles Kayser' },
...   { title: 1, languages: 1, released: 1, directors: 1, writers: 1, awards: 1, year: 1, genres: 1, runtime: 1, cast: 1, countries: 1 }
... )
[
  {
    _id: ObjectId('665437640bcc880d32cdce06'),
    title: 'Sample Movie',
    year: 2031,
    languages: [ 'English' ],
    released: ISODate('2031-08-20T00:00:00Z'),
    directors: [ 'Sample Director' ],
    writers: [ 'Writer A', 'Writer B' ],
    awards: { wins: 2, nominations: 5 },
    genres: [ 'Drama', 'Thriller' ],
    runtime: 110,
    cast: [ 'Charles Kayser', 'Actor B', 'Actor C' ],
    countries: [ 'USA' ]
  }
]
user> |
```

RESULT:

The query is executed successfully.

13.) Retrieve all movies with title, languages, releases, directors, writers, countries from the 'movies' collection in MongoDB that was released on May 9, 1893.

QUERY:

```
db.movies.find( { released: ISODate("1893-05-09T00:00:00.000Z") }, { title: 1, languages: 1, released: 1, directors: 1, writers: 1, countries: 1 })
```

OUTPUT:

```
user> db.movies.find(
...   { released: ISODate("1893-05-09T00:00:00.000Z") },
...   { title: 1, languages: 1, released: 1, directors: 1, writers: 1, countries: 1 }
...
{
  _id: ObjectId('66b437af0bcc80d32cdce07'),
  title: 'Sample Movie',
  released: ISODate('1893-05-09T00:00:00.000Z'),
  languages: ['English'],
  directors: ['Sample Director'],
  writers: ['Sample Writer'],
  countries: ['USA']
}
] user> |
```

RESULT:

The query is executed successfully.

14.) Retrieve all movies with title, languages, released, directors, writers, countries from the 'movies' collection in MongoDB that have a word "scene" in the title.

QUERY:

```
db.movies.find( { title: /scene/i }, { title: 1, languages: 1, released: 1, directors: 1, writers: 1, countries: 1 } )
```

OUTPUT:

```
user> db.movies.find(
...   { title: /scene/i },
...   { title: 1, languages: 1, released: 1, directors: 1, writers: 1, countries: 1 }
...
{
  _id: ObjectId('66b437ec0bcc80d32cdce08'),
  title: 'Sample Scene Movie',
  languages: ['English'],
  released: ISODate('2032-10-15T00:00:00.000Z'),
  directors: ['Sample Director'],
  writers: ['Sample Writer'],
  countries: ['USA']
}
] user> |
```

RESULT:

The query is executed successfully.

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