Department of Computer Science & Engineering

&

Information Technology

LAB MANUAL

Session 2013-2014

Subject Name :- DBMS Lab

Subject Code :- ICS-454

Branch :- CSE/IT

Year :- 2nd

Semester :- 4th

INTEGRAL UNIVERSITY LUCKNOW

Dsauli, Kursi Road, PO Basha-226026

Revision History

REVISION HISTORY					
REVISION #	**AUTHOR(S)**	**REVIEWER(S)**	**DATE OF RELEASE**	**OWNER**	**SUMMARY OF CHANGES**
Revision - 1	**Mr. Isha Mansoori** **Ms. Nida Khan**	Internal Reviewers	01/01/2014	HOD - CSE	Initial Release
Revision -2		Departmental QAC	31/01/2014	HOD - CSE	Major revisions made. Incorporated review comments from departmental QAC into Revision-2
Revision -3		External Reviewers	28/02/2013	HOD - CSE	

List of Lab Exercises

Experiment-1

OBJECTIVE

- Introduction of oracle and database design using E-R model.

Oracle workgroup or server is the largest selling RDBMS product.it is estimated that the combined sales of both these oracle database product account for aroud 80% of the RDBMSsystems sold worldwide.
These products are constantly undergoing change and evolving. The natural language of this RDBMS product is ANSI SQL,PL/SQL a superset of ANSI SQL.oracle 8i and 9i also understand SQLJ.

Oracle corp has also incorporated a full-fledged java virtual machine into its database engine.since both executable share the same memory space the JVM can communicate With the database engine with ease and has direct access to oracle tables and their data.

SQL is structure query language.SQL contains different data types those are

1. char(size)
2. varchar2(size)
3. date
4. number(p,s)
5. long
6. raw/long raw

How to Write and execute sql, pl/sql commands/programs:

1). Open your oracle application by the following navigation
Start->all programs->oracle orahome.->application development->sql.

2). You will be asked for user name, pass word and host string
You have to enter user name, pass word and host string as given by the administrator. It will be different from one user to another user.

3). Upon successful login you will get SQL prompt (SQL>).
In two ways you can write your programs:
a). directly at SQL prompt
b). or in sql editor.

If you type your programs at sql prompt then screen will look like follow:

```
SQL> SELECT ename,empno,
  2         sal from
  3         emp;
```

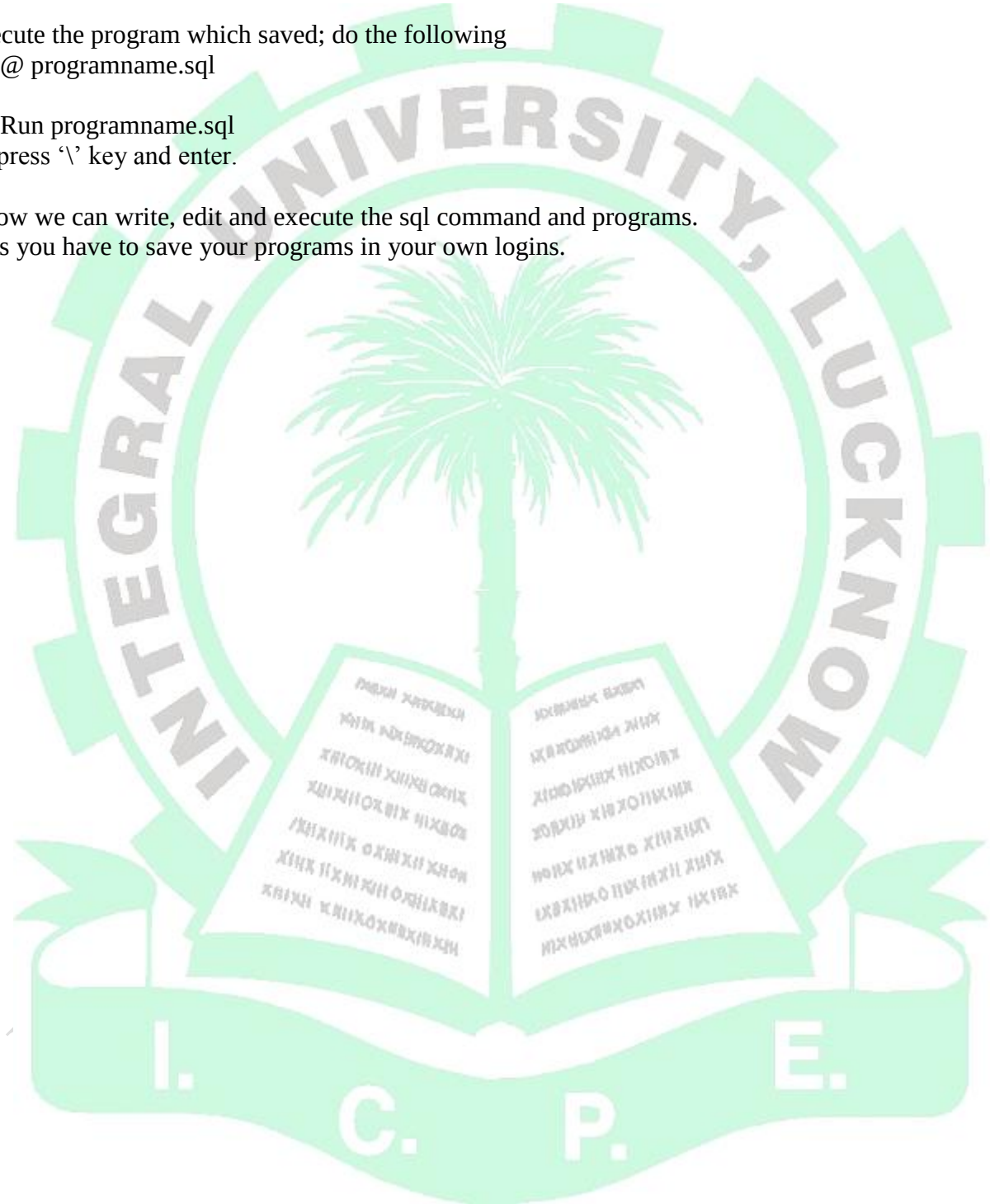
where 2 and 3 are the line numbers and rest is the command /program……

to execute above program/command you have to press '/' then enter.

Here editing the program is somewhat difficult; if you want to edit the previous command then you have to open sql editor (by default it displays the sql buffer contents). By giving 'ed' at sql prompt.(this is what I mentioned as a second method to type/enter the program).
in the sql editor you can do all the formatting/editing/file operations directly by selecting menu options provided by it.

To execute the program which saved; do the following
SQL> @ programname.sql
Or
SQL> Run programname.sql
Then press '\' key and enter.

This how we can write, edit and execute the sql command and programs.
Always you have to save your programs in your own logins.



Entity Relationship Model

Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.

For example, student and school -- they are two entities. Students study in school. So, these two entities are associated with a relationship "Studies in".

As another example, consider a system where some job runs every night, which updates the database. Here, job and database could be two entities. They are associated with the relationship "Updates".

Entity Set and Relationship Set

An entity set is a collection of all similar entities. For example, "Student" is an entity set that abstracts all students. Ram, John are specific entities belonging to this set. Similarly, a "Relationship" set is a set of similar relationships.

Attributes of Entity

Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.

For example, any student has got a name, age, an address. At any given time a student can study only at one school. In the school he would have a roll number, and of course a grade in which he studies. These data are the attributes of the entity set Student.

Weak Entity

An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by it's attributes. In other words, it doesn't have a super key.

For example, consider a company that allows employees to have travel allowance for their immediate family. So, here we have two entity sets: employee and family, related by "Can claim for". However, family doesn't have a super key. Existence of a family is entirely dependent on the concerned employee. So, it is meaningful only with reference to employee.

Entity Generalization and Specialization

Once we have identified the entity sets, we might find some similarities among them. For example, multiple person interacts with a banking system. Most of them are customers, and rest employees or other service providers. Here, customers, employees are persons, but with certain specializations. Or in other way, person is the generalized form of customer and employee entity sets.

ER model uses the "ISA" hierarchy to depict specialization (and thus, generalization).

Mapping Cardinalities

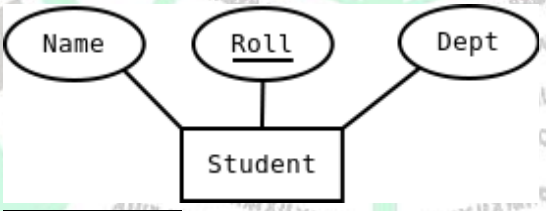
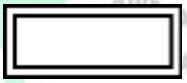
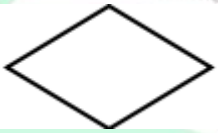
One of the main tasks of ER modeling is to associate different entity sets. Let's consider two entity sets E1 and E2 associated by a relationship set R. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:

- **One to one:** An entity in E1 is related to at most a single entity in E2, and vice versa
- **One to many:** An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1.
- **Many to one:** Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1.
- **Many to many:** Any number of entities could be related to any number of entities in E2, including zero, and vice versa.

ER Diagram

From a given problem statement we identify the possible entity sets, their attributes, and relationships among different entity sets. Once we have these information, we represent them pictorially, called an entity-relationship (ER) diagram.

Graphical Notations for ER Diagram

Term	Notation	Remarks
Entity set		Name of the set is written inside the rectangle
Attribute		Name of the attribute is written inside the ellipse
Entity with attributes		Roll is the primary key; denoted with an underline
Weak entity set		
Relationship set		Name of the relationship is written inside the diamond
Related enity sets		
Relationship cardinality		A person can own zero or more cars but no two persons can own the same car

Term	Notation	Remarks
Relationship with weak entity set		

Importance of ER modeling

Figure - 01 shows the different steps involved in implementation of a (relational) database.

Figure - 01: Steps to implement a RDBMS

Given a problem statement, the first step is to identify the entities, attributes and relationships. We represent them using an ER diagram. Using this ER diagram, table structures are created, along with required constraints. Finally, these tables are normalized in order to remove redundancy and maintain data integrity. Thus, to have data stored efficiently, the ER diagram is to be drawn as much detailed and accurate as possible.

Experiment-2

OBJECTIVE

- Write the queries for DDL, DML & DCL

Different types of commands in SQL:

A). **DDL commands: -** To create a database objects
B). **DML commands: -** To manipulate data of a database objects
C). **DQL command: -** To retrieve the data from a database.
D). **DCL/DTL commands: -** To control the data of a database…

DDL commands:

1. The Create Table Command: - it defines each column of the table uniquely. Each column has minimum of three attributes, a name , data type and size.

Syntax:
Create table <table name> (<col1> <datatype>(<size>),<col2> <datatype><size>));

Ex:
create table emp(empno number(4) primary key, ename char(10));

2. Modifying the structure of tables.
a)add new columns

Syntax:
Alter table <tablename> add(<new col><datatype(size),<new col>datatype(size));

Ex:
alter table emp add(sal number(7,2));

3. Dropping a column from a table.

Syntax:
Alter table <tablename> drop column <col>;

Ex:
alter table emp drop column sal;

4. Modifying existing columns.

Syntax:
Alter table <tablename> modify(<col><newdatatype>(<newsize>));

Ex:
alter table emp modify(ename varchar2(15));

5. Renaming the tables

Syntax:
Rename <oldtable> to <new table>;

Ex:
rename emp to emp1;

6. truncating the tables.

Syntax:
Truncate table <tablename>;

Ex:
trunc table emp1;

7. Destroying tables.

Syntax:
Drop table <tablename>;

Ex:
drop table emp;

DML commands:

8. Inserting Data into Tables: - once a table is created the most natural thing to do is load this table with data to be manipulated later.

Syntax:
insert into <tablename> (<col1>,<col2>) values(<exp>,<exp>);

9. Delete operations.

a) remove all rows
Syntax:
delete from <tablename>;

b) removal of a specified row/s
Syntax:
delete from <tablename> where <condition>;

10. Updating the contents of a table.

a) updating all rows
Syntax:
Update <tablename> set <col>=<exp>,<col>=<exp>;

b) updating seleted records.

Syntax:

Update<tablename>set<col>=<exp>,<col>=<exp>
where <condition>;

11. Types of data constrains.

a) not null constraint at column level.

Syntax:

<col><datatype>(size)not null

b) unique constraint

Syntax:

Unique constraint at column level.
<col><datatype>(size)unique;

c) unique constraint at table level:

Syntax:

Create table tablename(col=format,col=format,unique(<col1>,<col2>);

d) primary key constraint at column level

Syntax:

<col><datatype>(size)primary key;

e) primary key constraint at table level.

Syntax:

Create table tablename(col=format,col=format
primary key(col1>,<col2>);

f) foreign key constraint at column level.

Syntax:

<col><datatype>(size>) references <tablename>[<col>];

g) foreign key constraint at table level

Syntax:

foreign key(<col>[,<col>])references <tablename>[(<col>,<col>)

h) check constraint

check constraint constraint at column level.
Syntax: <col><datatype>(size) check(<logical expression>)

i) check constraint constraint at table level.

Syntax: check(<logical expression>)

DQL Commands:

12. Viewing data in the tables: - once data has been inserted into a table, the next most logical operation would be to view what has been inserted.

a) all rows and all columns

Syntax:

Select <col> to <col n> from tablename;

Select * from tablename;

13. Filtering table data: - while viewing data from a table, it is rare that all the data from table will be required each time. Hence, sql must give us a method of filtering out data that is not required data.

a) Selected columns and all rows:

Syntax:

select <col1>,<col2> from <tablename>;

b) selected rows and all columns:

Syntax:

select * from <tablename> where <condition>;

c) selected columns and selected rows

Syntax:

select <col1>,<col2> from <tablename> where<condition>;

14. Sorting data in a table.

Syntax:

Select * from <tablename> order by <col1>,<col2> <[sortorder]>;

DCL commands:

Oracle provides extensive feature in order to safeguard information stored in its tables from unauthoraised viewing and damage.The rights that allow the user of some or all oracle resources on the server are called privileges.

a) Grant privileges using the GRANT statement

The grant statement provides various types of access to database objects such as tables,views and sequences and so on.

Syntax:

GRANT <object privileges>
ON <objectname>
TO<username>
[WITH GRANT OPTION];

b) Reoke permissions using the REVOKE statement:

The REVOKE statement is used to deny the Grant given on an object.

Syntax:

REVOKE<object privilege>
ON
FROM<user name>;

Experiment No 3 & 4

OBJECTIVE

- Write queries using Logical Operators (=, <,> etc.)
- Write queries using SQL Operators (BETWEEN, AND, IN, LIKE, ISNULL and along with Negation expressions.)

1. **Get the description of EMP table.**

SQL> desc emp;

RESULT:

```
Name Null? Type
----------------------------- ---------------------- ------------------------
EMPNO NOT NULL NUMBER(4)
ENAME VARCHAR2(10)
JOB VARCHAR2(9)
MGR NUMBER(4)
HIREDATE DATE
SAL NUMBER(7,2)
COMM NUMBER(7,2)
DEPTNO NUMBER(3)
AGE NUMBER(3)
ESAL NUMBER(10)
```

2. **Get the description DEPT table.**

SQL>desc dept;

RESULT:

```
Name Null? Type
------------------------------- -------------------- ----------------------
DEPTNO NOT NULL NUMBER(2)
DNAME VARCHAR2(14)
LOC VARCHAR2(13)
```

3. List all employee details.

SQL>select * from emp;

4. List all employee names and their salaries, whose salary lies between 1500/- and 3500/- both inclusive.

SQL>select ename from emp where sal between 1500 and 3500;

5. List all employee names and their and their manager whose manager is 7902 or 7566 0r 7789.

SQL>select ename from emp where mgr in(7602,7566,7789);

6. List all employees which starts with either J or T.

SQL>select ename from emp where ename like „J%" or ename like „T%";

7. List all employee names and jobs, whose job title includes M or P.

SQL>select ename,job from emp where job like „M%" or job like „P%";

8. List all jobs available in employee table.

SQL>select distinct job from emp;

9. List all employees who belongs to the department 10 or 20.

SQL>select ename from emp where deptno in (10,20);

10. List all employee names , salary and 15% rise in salary.

SQL>select ename , sal , sal+0.15* sal from emp;

11. List minimum , maximum , average salaries of employee.

SQL>select min(sal),max(sal),avg(sal) from emp;

12. Find how many job titles are available in employee table.

SQL>select count (distinct job) from emp;

13. What is the difference between maximum and minimum salaries of employees in the organization?

SQL>select max(sal)-min(sal) from emp;

14. Display all employee names and salary whose salary is greater than minimum salary of the company and job title starts with 'M'.

SQL>select ename,sal from emp where job like „M%" and sal > (select min (sal) from emp);

15. Find how much amount the company is spending towards salaries.

SQL>select sum (sal) from emp;

16. Display name of the dept. with deptno 20.

SQL>select ename from emp where deptno = 20;

17. List ename whose commission is NULL.

SQL>select ename from emp where comm is null;

18. Find no.of dept in employee table.

SQL>select count (distinct ename) from emp;

19. List ename whose manager is not NULL.

SQL>select ename from emp where mgr is not null;

20. Display all the details of all 'Mgrs'

SQL>Select * from emp where empno in (select mgr from emp) ;

21. List the emps who joined before 1981.

SQL>select * from emp where hiredate < ('01-jan-81');

22. List the Empno, Ename, Sal, Daily sal of all emps in the asc order of Annsal.

SQL>select empno ,ename ,sal,sal/30,12*sal annsal from emp order by annsal asc;

23. Display the Empno, Ename, job, Hiredate, Exp of all Mgrs

SQL>select empno,ename ,job,hiredate, months_between(sysdate,hiredate) exp

from emp where empno in (select mgr from emp);

24. List the Empno, Ename, Sal, Exp of all emps working for Mgr 7369.

SQL>select empno,ename,sal,exp from emp where mgr = 7369;

25. Display all the details of the emps whose Comm. Is more than their Sal.

SQL>select * from emp where comm. > sal;

26. List the emps along with their Exp and Daily Sal is more than Rs.100.

SQL>select * from emp where (sal/30) >100;

27. List the emps who are either 'CLERK' or 'ANALYST' in the Desc order.

SQL>select * from emp where job = 'CLERK' or job = 'ANALYST' order by job desc;

28. List the emps who joined on 1-MAY-81,3-DEC-81,17-DEC-81,19-JAN-80 in asc order of seniority.

SQL>select * from emp where hiredate in ('01-may-81','03-dec-81','17-dec-81','19-jan-80') order by hiredate asc;

29. List the emp who are working for the Deptno 10 or20.

SQL>select * from emp where deptno = 10 or deptno = 20 ;

30. List the emps who are joined in the year 81.

SQL>select * from emp where hiredate between '01-jan-81' and '31-dec-81';

31. List the emps who are joined in the month of Aug 1980.

SQL>select * from emp where hiredate between '01-aug-80' and '31-aug-80'; (OR)
select * from emp where to_char(hiredate,'mon-yyyy') ='aug-1980;

32. List the emps Who Annual sal ranging from 22000 and 45000.

SQL>select * from emp where 12*sal between 22000 and 45000;

33. List the emps those are having four chars and third character must be 'r'.

SQL>A) select * from emp where length(ename) = 4 and ename like '__R%';

34. List the emps whose names having a character set 'll' together.

SQL>select * from emp where ename like '%LL%';

35. List the emps who does not belong to Deptno 20.

SQL>select * from emp where deptno not in (20); (OR)
SQL>select * from emp where deptno != 20; (OR)
SQL>select * from emp where deptno <>20; (OR)
SQL>select * from emp where deptno not like '20';

36. List the emps whose Empno not starting with digit78.

SQL>select * from emp where empno not like '78%';

37. List all the Clerks of Deptno 20.
SQL>select * from emp where job ='CLERK' and deptno = 20;

38. Display the details of SMITH.
SQL>select * from emp where ename = 'SMITH' ;

Experiment No 5&6

Objectives:

- Write SQL Queries using Character, Number, Date and group Functions.
- Write SQL Queries for Relational Algebra (UNION,INTERSECT and MINUS etc.)

Order by : The order by clause is used to display the results in sorted order.

Group by : The attribute or attributes given in the clauses are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

Having: SQL applies predicates (conditions) in the having clause after groups have been formed, so aggregate function be used.

1. List the emps in the asc order of their Salaries?

A) select * from emp order by sal asc;

2. List the details of the emps in asc order of the Dptnos and desc of Jobs?

A)select * from emp order by deptno asc,job desc;

3. Display all the unique job groups in the descending order?

A)select distinct job from emp order by job desc;

4. List the emps in the asc order of Designations of those joined after the second half of 1981.

A) select * from emp where hiredate > ('30-jun-81') and to_char(hiredate,'YYYY') = 1981 order by job asc;

5. List all the emps except 'PRESIDENT' & 'MGR" in asc order of Salaries.

A) Select * from emp where job not in ('PRESIDENT','MANAGER') order by sal asc;

B) Select * from emp where job not like 'PRESIDENT' and job not like 'MANAGER' order by sal asc;

C) Select * from emp where job != 'PRESIDENT' and job <> 'MANAGER' order by sal asc;

6. List the Enames those are having five characters in their Names.

A) select ename from emp where length (ename) = 5;

7. List the Enames those are starting with 'S' and with five characters.

A) select ename from emp where ename like 'S%' and length (ename) = 5;

8. List the Five character names starting with 'S' and ending with 'H'.

A) select * from emp where length(ename) = 5 and ename like 'S%H';

9. List the emps who joined in January.

A) select * from emp where to_char (hiredate,'mon') = 'jan';

10. List the emps who joined in the month of which second character is 'a'.

A) select * from emp where to_char(hiredate,'mon') like '_a_';
(OR)
B) select * from emp where to_char(hiredate,'mon') like '_a%';

11. List the emps whose Sal is four digit number ending with Zero.

A) select * from emp where length (sal) = 4 and sal like '%0';

12. List the emps those who joined in 80's.

A) select * from emp where to_char(hiredate,'yy') like '8%';

13. List all the emps who joined before or after 1981.

A) select * from emp where to_char (hiredate,'YYYY') not in ('1981'); (OR)
B) select * from emp where to_char (hiredate,'YYYY') != '1981'; (OR)
C) select * from emp where to_char(hiredate,'YYYY') <> '1981' ; (OR)

D) select * from emp where to_char (hiredate ,'YYYY') not like '1981';

14. List the emps who are working under 'MGR'.

A) select e.ename || ' works for ' || m.ename from emp e ,emp m where e.mgr = m.empno ; (OR)
B) select e.ename || ' has an employee '|| m.ename from emp e , emp m where e.empno = m.mgr;

15. List the emps who joined in any year but not belongs to the month of March.

A) select * from emp where to_char (hiredate,'MON') not in ('MAR'); (OR)
B) select * from emp where to_char (hiredate,'MON') != 'MAR'; (OR)
C) select * from emp where to_char(hiredate,'MONTH') not like 'MAR%' ; (OR)
D) select * from emp where to_char(hiredate,'MON') <> 'MAR';

16. Give a count of how many employees are working in each department

A) select count(empid),deptid from paydet group by deptid;

17. Display total salary spent for each job category

A) select job,sum (sal) from emp group by job;

The Helpful Dual

Oracle Database provides a single-row, single-column table called DUAL that is useful for many purposes, not the least of which is learning about Oracle functions. DUAL is an Oracle system table owned by the SYS user, not the SQL_101 schema. Many Oracle system tables are made available to all users via *public synonyms*. Synonyms will be discussed in subsequent articles in this series.
The DUAL table contains no data that's useful in and of itself. (It has one row with one column—called the DUMMY column—that contains the value X.) You can use DUAL to try out functions that work on string literals and, as you'll see in subsequent articles in this series, on number literals and even on today's date.

The following demonstrates the single-row, single-column output of a SELECT statement executed against the DUAL table:

SQL> select * from dual;

D
-
X

1 row selected.

To display the current date, you can query the DUAL table as follows:

SQL> select sysdate from dual;

SYSDATE
_____

18-APR-12

1 row selected.

And finally, the following example shows how you can practice any function in the SELECT clause of a SQL statement, using the DUAL table:

Stringing Strings Together

Sometimes it makes sense to combine certain strings, such as the FIRST_NAME and LAST_NAME values from the EMPLOYEE table, in the result set display. You can use *concatenation* to accomplish this task—with either the CONCAT function, illustrated in Listing 6, or the (more commonly used) concatenation operator || (two pipe characters), illustrated in Listing 7.

Code Listing 6: Query that demonstrates the CONCAT function

SQL> select CONCAT(first_name, last_name) employee_name from employee order by employee_name;

EMPLOYEE_NAME
_____

BetsyJames
DonaldNewton
EmilyEckhardt
FrancesNewton
MatthewMichaels
RogerFriedli
markleblanc
michaelpeterson

8 rows selected.

1: **Query that demonstrates the concatenation operator, ||**

```
SQL> select first_name||' '||last_name employee_name
  2    from employee
  3  order by employee_name;

EMPLOYEE_NAME
_____________________________________________

Betsy James
Donald Newton
Emily Eckhardt
Frances Newton
Matthew Michaels
Roger Friedli
mark leblanc
michael peterson

8 rows selected.
```

The CONCAT function takes two parameters and concatenates them. You can also nest multiple CONCAT function calls, as shown in Listing 8. The queries in Listings 7 and 8 concatenate literal strings with column data values. (I prefer the concatenation operator, because it has unlimited input parameters and makes the concatenated output more readable.)

2: Query that demonstrates nested CONCAT calls

```
SQL> select CONCAT(first_name, CONCAT(' ', last_name)) employee_name
  2    from employee
  3  order by employee_name;

EMPLOYEE_NAME
_____________________________________________

Betsy James
Donald Newton
Emily Eckhardt
Frances Newton
Matthew Michaels
Roger Friedli
mark leblanc
michael peterson

8 rows selected.
```

Giving Your Data a Trim

Sometimes you want to remove unwanted spaces or characters from data when you display it. For example, data inserted into a table column via a form application might include extraneous characters or spaces—preceding and/or following the actual data value—that the form input field doesn't trim.

Listing 9 shows a query that trims extra spaces from string values. The TRIM function in Listing 9 takes two parameters. The first parameter is the character, symbol, or space (delimited by single quotes) to be removed. The second parameter specifies the string literal or column value to be trimmed. The TRIM function supports three keywords: LEADING, TRAILING, and BOTH. The example in Listing 9 uses the TRAILING keyword to right-trim the FIRST_NAME value. The TRIM function applied to the LAST_NAME value specifies the LEADING keyword to left-trim the spaces from that value. And, as you can see, the spaces to the right of the LAST_NAME value remain and are included in the output.

3: Query that trims extra spaces

```
SQL> select "" ||TRIM(TRAILING ' ' FROM 'Ashton    ') || "" first_name,
 "" || TRIM(LEADING ' ' FROM '  Cinder   ') || "" last_name
  2    from dual;

FIRST_NA  LAST_NAME
————————  ————————————————————

'Ashton'  'Cinder   '

1 row selected.
```

Compare the output in Listing 9 with that in Listing 10, which trims the rightmost extra spaces from the LAST_NAME value. When no keyword is specified, the default behavior for the TRIM function is to trim leading as well as trailing characters. The older RTRIM and LTRIM functions are available for backward compatibility.

4: Query that trims extra spaces, including rightmost extra spaces

```
SQL> select "" || TRIM(TRAILING ' ' FROM 'Ashton ') || "" first_name,
"" || TRIM(' Cinder ') || "" last_name
2 from dual;

FIRST_NA LAST_NAM
———————— ———————————

'Ashton' 'Cinder'

1 row selected.
```

Searching for Strings Within Strings

When you need to search column values for similar string pattern values, you can do so with the INSTR character function. INSTR—which stands for *in string*—returns the position of a substring within a string value. Listing 11 demonstrates the INSTR function applied to the LAST_NAME column of the EMPLOYEE table to locate all occurrences of the “ton“ substring. As you can see, the INSTR function takes as input the literal or column value you want to search, followed by the substring pattern to search for. In Listing 11, the INSTR function finds the “ton“ pattern in only two column data values—both of them Newton—and returns 4 as their position. Because it did not find the search string in any other values, the output for those values is 0.

Code Listing 11: Query that demonstrates the INSTR character function

```
SQL> select last_name, INSTR(last_name, 'ton') ton_starting_point
  2    from employee
  3  order by last_name;

LAST_NAME            TON_STARTING_POINT
-------------------- ------------------
Eckhardt                             0
Friedli                              0
James                                0
Michaels                             0
Newton                               4
Newton                               4
leblanc                              0
peterson                             0

8 rows selected.
```

Two additional parameters—*starting position* and *occurrence*—are optional. The starting position specifies the character in the string from which to begin your search. The default behavior is for the search to begin at the first character—otherwise known as character position 1. The occurrence parameter lets you specify which occurrence of the substring you'd like to find. For example, the word *Mississippi* includes two occurrences of the "issi" substring. To search for the starting-position location of the second occurrence of this pattern, you must provide the INSTR function with an occurrence parameter of 2:

```
SQL> select INSTR('Mississippi', 'issi', 1, 2)
  2    from dual;

INSTR('MISSISSIPPI','ISSI',1,2)
-------------------------------
                              5
1 row selected.
```

Extracting Strings from Strings

Sometimes you need to extract a portion of a string for your desired output. The SUBSTR (for *substring*) character function can assist you with this task. Listing 12 shows a query that uses the SUBSTR function to extract the first three characters of every LAST_NAME value from the EMPLOYEE table. The SUBSTR function takes two required parameters and one optional input parameter. The first parameter is the literal or column value on which you want the SUBSTR function to operate. The second parameter is the position of the starting character for the substring, and the optional third parameter is the number of characters to be included in the substring. If the third parameter is not specified, the SUBSTR function will return the remainder of the string.

5: Query that demonstrates the SUBSTR character function

```
SQL> select last_name, SUBSTR(last_name, 1, 3)
  2    from employee
  3  order by last_name;

LAST_NAME                 SUB
_____________________________________________ ________

Eckhardt                  Eck
Friedli                   Fri
James                     Jam
Michaels                  Mic
Newton                    New
Newton                    New
leblanc                   leb
peterson                  pet

8 rows selected.
```

Listing 13 demonstrates the SUBSTR and INSTR functions working together to display the portion of every LAST_NAME value from the EMPLOYEE table that contains the “ton“ substring. In this example, the output from the INSTR function provides the value for the input parameter that specifies the position for the SUBSTR function’s starting character. In the LAST_NAME values in which the substring “ton“ is not found, the entire LAST_NAME value is returned, for two reasons: SUBSTR treats a starting position of 0 the same as a starting position of 1 (that is, as the first position in the string), and because the query omits the optional length parameter, the full remainder of the string is returned.

6: **Query that demonstrates the INSTR and SUBSTR character functions**

```
SQL> select last_name, INSTR(last_name, 'ton') ton_position, SUBSTR(last_name,
INSTR(last_name, 'ton')) substring_ton
2 from employee
3 order by last_name;
LAST_NAME TON_POSITION SUBSTRING_TON
_____________ ____________________ ____________________________________

Eckhardt                  0                  Eckhardt
Friedli                   0                  Friedli
James                     0                  James
Michaels                  0                  Michaels
Newton                    4                  ton
Newton                    4                  ton
leblanc                   0                  leblanc
peterson                  0                  peterson

8 rows selected.
```

When Size Matters

Occasionally you need to determine a string's length—for example, to determine the maximum number of characters a form entry field should permit. Listing 14 shows a query that uses the LENGTH function to display the length of all FIRST_NAME values from the EMPLOYEE table.

7: Query that demonstrates the LENGTH function

```
SQL> select first_name, LENGTH(first_name) length
  2    from employee
  3  order by length desc, first_name;

FIRST_NAME          LENGTH
_____________________  _____________________________

Frances                7
Matthew                 7
michael                7
Donald                  6
Betsy                  5
Emily                   5
Roger                   5
mark                    4

8 rows selected.
```

Character functions can also be placed in WHERE and ORDER BY clauses, as illustrated in Listings 15 and 16.

In Listing 15, the total length of those employee first and last names concatenated together, separated by a single space, is calculated with the LENGTH function. And only values that are more than 15 characters long are returned in the result set. In Listing 16, the WHERE clause uses the INSTR function nested inside the SUBSTR function to return only those employees whose last names contain the substring "ton"—the resultant employee first and last name values are concatenated and separated with a space. Finally, the query's ORDER BY clause sorts the concatenated first and last name values from the SELECT list by character length in descending order, by using the LENGTH character function.

8: Query that demonstrates a function in a WHERE clause

```
SQL> select first_name||' '||last_name employee_name
  2    from employee
  3   where LENGTH(first_name||' '||last_name) > 15
  4  order by employee_name;

EMPLOYEE_NAME
_____________________________________

Matthew Michaels
michael peterson

2 rows selected.
```

9: Query that demonstrates functions in a WHERE and an ORDER BY clause

```
SQL> select first_name||' '||last_name employee_name
  2    from employee
  3   where SUBSTR(last_name, INSTR(last_name, 'ton')) = 'ton'
  4  order by LENGTH(employee_name) desc;

EMPLOYEE_NAME
_____________________________________

Frances Newton
Donald Newton

2 rows selected.
```

10: Rounding Numeric Data

```
SELECT LAST_NAME,
ROUND (((SALARY * 12)/365), 2) "Daily Pay"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 Daily Pay
------------------------- ----------
Chen                       269.59
Faviet                     295.89
Greenberg                    394.52
Popp                       226.85
Sciarra                    253.15
Urman                       256.44

6 rows selected.
```

101: Truncating Numeric Data

```
SELECT LAST_NAME,
TRUNC ((SALARY * 12)/365) "Daily Pay"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 Daily Pay
```

---------- ----------
Chen 269
Faviet 295
Greenberg 394
Popp 226
Sciarra 253
Urman 256

6 rows selected.

12: Concatenating Character Data

SELECT **FIRST_NAME || ' ' || LAST_NAME** "Name"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;

Result:

Name

John Chen
Daniel Faviet
Nancy Greenberg
Luis Popp
Ismael Sciarra
Jose Manuel Urman

6 rows selected.

13: Changing the Case of Character Data

SELECT **UPPER(LAST_NAME)** "Last",
INITCAP(FIRST_NAME) "First",
LOWER(EMAIL) "E-Mail"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY EMAIL;

Result:

Last First E-Mail
---------- ---------- ----------
FAVIET Daniel dfaviet
SCIARRA Ismael isciarra
CHEN John jchen
URMAN Jose Manuel jmurman
POPP Luis lpopp

GREENBERG Nancy ngreenbe

6 rows selected.

14: Trimming Character Data

```
SELECT LAST_NAME,
RTRIM(JOB_ID, '_CLERK') "Clerk Type"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%_CLERK'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME            Clerk Type
-------------------- ----------
Atkinson             ST
Baida                PU
Bell                 SH
Bissot               ST
Bull                 SH
Cabrio               SH
Chung                SH
Colmenares           PU
Davies               ST
Dellinger            SH
Dilly                SH

LAST_NAME            Clerk Type
-------------------- ----------
Everett              SH
Feeney               SH
...
LAST_NAME            Clerk Type
-------------------- ----------
Walsh                SH

45 rows selected.
```

15: Padding Character Data

```
SELECT LPAD(FIRST_NAME,15) "First",
RPAD(LAST_NAME,15) "Last"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY FIRST_NAME;
```

Result:

```
First          Last
-------------- --------------
        Daniel Faviet
        Ismael Sciarra
          John Chen
    Jose Manuel Urman
          Luis Popp
         Nancy Greenberg

6 rows selected.
```

16: Extracting Substrings from Character Data

```
SELECT SUBSTR(FIRST_NAME, 1, 1) || '. ' || LAST_NAME "Name",
SUBSTR(PHONE_NUMBER, 5, 8) "Phone"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
Name                        Phone
--------------------------- --------
J. Chen                    124.4269
D. Faviet                   124.4169
N. Greenberg                  124.4569
L. Popp                    124.4567
I. Sciarra                 124.4369
J. Urman                    124.4469

6 rows selected.
```

17: Replacing Substrings in Character Data

```
COLUMN "Job" FORMAT A15;
SELECT LAST_NAME,
REPLACE(JOB_ID, 'SH', 'SHIPPING') "Job"
FROM EMPLOYEES
WHERE SUBSTR(JOB_ID, 1, 2) = 'SH'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 Job
------------------------- ---------------
Bell                      SHIPPING_CLERK
```

```
Bull                 SHIPPING_CLERK
Cabrio               SHIPPING_CLERK
Chung                SHIPPING_CLERK
Dellinger            SHIPPING_CLERK
Dilly                SHIPPING_CLERK
Everett              SHIPPING_CLERK
Feeney               SHIPPING_CLERK
Fleaur               SHIPPING_CLERK
Gates                SHIPPING_CLERK
Geoni                SHIPPING_CLERK

LAST_NAME            Job
------------------------ ---------------
Grant                SHIPPING_CLERK
Jones                SHIPPING_CLERK
McCain               SHIPPING_CLERK
OConnell             SHIPPING_CLERK
Perkins              SHIPPING_CLERK
Sarchand             SHIPPING_CLERK
Sullivan             SHIPPING_CLERK
Taylor               SHIPPING_CLERK
Walsh                SHIPPING_CLERK

20 rows selected.
```

Using Datetime Functions in Queries

Example 4-30 Displaying System Date and Time

```
SELECT EXTRACT(HOUR FROM SYSTIMESTAMP) || ':' ||
EXTRACT(MINUTE FROM SYSTIMESTAMP) || ':' ||
ROUND(EXTRACT(SECOND FROM SYSTIMESTAMP), 0) || ', ' ||
EXTRACT(MONTH FROM SYSTIMESTAMP) || '/' ||
EXTRACT(DAY FROM SYSTIMESTAMP) || '/' ||
EXTRACT(YEAR FROM SYSTIMESTAMP) "System Time and Date"
FROM DUAL;
```

Results depend on current **SYSTIMESTAMP** value, but have this format:

```
System Time and Date
-------------------------------------------------------------
18:47:33, 6/19/2008
```

Using Conversion Functions in Queries

Conversion functions convert one data type to another. The query in Example uses the **TO_CHAR** function to convert **HIRE_DATE** values (which are of type **DATE**) to character values that have the format **FMMonth DD YYYY**. **FM**removes leading and trailing blanks from the month name. **FMMonth DD YYYY** is an example of a **datetime format model**.

Example 4-31 Converting Dates to Characters Using a Format Template

```
SELECT LAST_NAME,
HIRE_DATE,
TO_CHAR(HIRE_DATE, 'FMMonth DD YYYY') "Date Started"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME            HIRE_DATE Date Started
-------------------- --------- -----------------
Chen                 28-SEP-05 September 28 2005
Faviet               16-AUG-02 August 16 2002
Greenberg              17-AUG-02 August 17 2002
Popp                 07-DEC-07 December 7 2007
Sciarra              30-SEP-05 September 30 2005
Urman                 07-MAR-06 March 7 2006

6 rows selected.
```

The query in Example 4-32 uses the **TO_CHAR** function to convert **HIRE_DATE** values to character values that have the two standard formats **DS** (Date Short) and **DL** (Date Long).

1: Converting Dates to Characters Using Standard Formats

```
SELECT LAST_NAME,
TO_CHAR(HIRE_DATE, 'DS') "Short Date",
TO_CHAR(HIRE_DATE, 'DL') "Long Date"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME            Short Date Long Date
-------------------- ---------- ----------------------------
Chen                 9/28/2005  Sunday, September 28, 2005
Faviet               8/16/2002  Tuesday, August 16, 2002
Greenberg              8/17/2002  Wednesday, August 17, 2002
Popp                 12/7/2007  Tuesday, December 07, 2007
Sciarra              9/30/2005  Tuesday, September 30, 2005
Urman                 3/7/2006   Saturday, March 07, 2006

6 rows selected.
```

The query in Example 4-33 uses the **TO_CHAR** function to convert **SALARY** values (which are of type **NUMBER**) to character values that have the format **$99,999.99**.

2: Converting Numbers to Characters Using a Format Template

```
SELECT LAST_NAME,
TO_CHAR(SALARY, '$99,999.99') "Salary"
FROM EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY SALARY;
```

Result:

```
LAST_NAME            Salary
-------------------- ----------
Popp                 $6,900.00
Sciarra              $7,700.00
Urman                 $7,800.00
Chen                 $8,200.00
Faviet               $9,000.00
Greenberg             $12,000.00

6 rows selected.
```

The query in Example 4-34 uses the **TO_NUMBER** function to convert **POSTAL_CODE** values (which are of type **VARCHAR2**) to values of type **NUMBER**, which it uses in calculations.

3: Converting Characters to Numbers

```
SELECT CITY,
POSTAL_CODE "Old Code",
TO_NUMBER(POSTAL_CODE) + 1 "New Code"
FROM LOCATIONS
WHERE COUNTRY_ID = 'US'
ORDER BY POSTAL_CODE;
```

Result:

```
CITY                      Old Code      New Code
------------------------- ------------ ----------
Southlake                  26192          26193
South Brunswick             50090          50091
Seattle                   98199          98200
South San Francisco          99236          99237

4 rows selected.
```

The query in Example 4-35 uses the **TO_DATE** function to convert a string of characters whose format is **Month dd, YYYY, HH:MI A.M.** to a **DATE** value.

4: Converting a Character String to a Date

```
SELECT TO_DATE('January 5, 2007, 8:43 A.M.',
'Month dd, YYYY, HH:MI A.M.') "Date"
FROM DUAL;
```

Result:

```
Date
---------
05-JAN-07
```

The query in Example 4-36 uses the **TO_TIMESTAMP** function to convert a string of characters whose format is **DD-Mon-RR HH24:MI:SS.FF** to a **TIMESTAMP** value.

5: Converting a Character String to a Time Stamp

```
SELECT TO_TIMESTAMP('May 5, 2007, 8:43 A.M.',
'Month dd, YYYY, HH:MI A.M.') "Timestamp"
FROM DUAL;
```

Result:

```
Timestamp
---------------------------------------------------------------------------
05-MAY-07 08.43.00.000000000 AM
```

Using Aggregate Functions in Queries

An aggregate function returns a single result row, based on a group of rows. The group of rows can be an entire table or view.

Aggregate functions are especially powerful when used with the **GROUP BY** clause, which groups query results by one or more columns, with a result for each group.

The query in Example 4-37 uses the **COUNT** function and the **GROUP BY** clause to show how many people report to each manager. The wildcard character, *****, represents an entire record.

6: Counting the Number of Rows in Each Group

```
SELECT MANAGER_ID "Manager",
COUNT(*) "Number of Reports"
FROM EMPLOYEES
GROUP BY MANAGER_ID;
```

Result:

```
Manager Number of Reports
---------- ----------------
       100               14
                          1
```

123	8
120	8
121	8
147	6
205	1
108	5
148	6
149	6
201	1

Manager	Number of Reports
102	1
101	5
114	5
124	8
145	6
146	6
103	4
122	8

19 rows selected.

Example shows that one employee does not report to a manager. The following query selects the first name, last name, and job title of that employee:

```
COLUMN FIRST_NAME FORMAT A10;
COLUMN LAST_NAME FORMAT A10;
COLUMN JOB_TITLE FORMAT A10;

SELECT e.FIRST_NAME,
e.LAST_NAME,
j.JOB_TITLE
FROM EMPLOYEES e, JOBS j
WHERE e.JOB_ID = j.JOB_ID
AND MANAGER_ID IS NULL;
```

Result:

FIRST_NAME	LAST_NAME	JOB_TITLE
Steven	King	President

When used with the **DISTINCT** option, the **COUNT** function shows how many distinct values are in a data set.

The two queries in Example 4-38 show the total number of departments and the number of departments that have employees.

Example 7: Counting the Number of Distinct Values in a Set

```
SELECT COUNT(*) FROM DEPARTMENTS;
```

Result:

```
COUNT(*)
----------
        27
```

```
SELECT COUNT(DISTINCT DEPARTMENT_ID) "Number of Departments"
FROM EMPLOYEES;
```

Result:

```
Number of Departments
---------------------
                   11
```

The query in Example 4-39 uses several aggregate functions to show statistics for the salaries of each **JOB_ID**.

Example 8: Using Aggregate Functions for Statistical Information

```
SELECT JOB_ID,
COUNT(*) "#",
MIN(SALARY) "Minimum",
ROUND(AVG(SALARY), 0) "Average",
MEDIAN(SALARY) "Median",
MAX(SALARY) "Maximum",
ROUND(STDDEV(SALARY)) "Std Dev"
FROM EMPLOYEES
GROUP BY JOB_ID
ORDER BY JOB_ID;
```

Result:

```
JOB_ID           #    Minimum    Average     Median    Maximum    Std Dev
---------- ---------- ---------- ---------- ---------- ---------- ----------
AC_ACCOUNT         1       8300       8300       8300       8300          0
AC_MGR             1      12000      12000      12000      12000          0
AD_ASST            1       4400       4400       4400       4400          0
AD_PRES            1      24000      24000      24000      24000          0
AD_VP              2      17000      17000      17000      17000          0
FI_ACCOUNT         5       6900       7920       7800       9000        766
FI_MGR             1      12000      12000      12000      12000          0
HR_REP             1       6500       6500       6500       6500          0
IT_PROG            5       4200       5760       4800       9000       1926
MK_MAN             1      13000      13000      13000      13000          0
MK_REP             1       6000       6000       6000       6000          0
```

```
JOB_ID          #    Minimum    Average    Median    Maximum    Std Dev
---------- ---------- ---------- ---------- ---------- ---------- ----------
PR_REP          1      10000      10000      10000      10000          0
PU_CLERK        5       2500       2780       2800       3100        239
PU_MAN          1      11000      11000      11000      11000          0
SA_MAN          5      10500      12200      12000      14000       1525
SA_REP         30       6100       8350       8200      11500       1524
SH_CLERK       20       2500       3215       3100       4200        548
ST_CLERK       20       2100       2785       2700       3600        453
ST_MAN          5       5800       7280       7900       8200       1066

19 rows selected.
```

To have the query return only rows where aggregate values meet specified conditions, use the **HAVING** clause.

The query in Example 4-40 shows how much each department spends annually on salaries, but only for departments for which that amount exceeds $1,000,000.

Example 9: Limiting Aggregate Functions to Rows that Satisfy a Condition

```
SELECT DEPARTMENT_ID "Department",
SUM(SALARY*12) "All Salaries"
FROM EMPLOYEES
HAVING SUM(SALARY * 12) >= 1000000
GROUP BY DEPARTMENT_ID;
```

Result:

```
Department All Salaries
---------- ------------
        50      1876800
        80      3654000
```

The **RANK** function returns the relative ordered rank of a number, and the **PERCENT_RANK** function returns the percentile position of a number.

The query in Example 4-41 shows that a salary of $3,000 is the 20th highest, and is in the 42nd percentile, among all clerks.

Example 10: Showing the Rank and Percentile of a Number Within a Group

```
SELECT RANK(3000) WITHIN GROUP
  (ORDER BY SALARY DESC) "Rank",
ROUND(100 * (PERCENT_RANK(3000) WITHIN GROUP
  (ORDER BY SALARY DESC)), 0) "Percentile"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%CLERK';
```

Result:

```
Rank Percentile
---------- ----------
      20         42
```

The **DENSE_RANK** function is like the **RANK** function, except that the identical values have the same rank, and there are no gaps in the ranking. Using the **DENSE_RANK** function, $3,000 is the 12th highest salary for clerks, as Example 4-42 shows.

Example 11: Showing the Dense Rank of a Number Within a Group

```
SELECT DENSE_RANK(3000) WITHIN GROUP (ORDER BY salary DESC) "Rank"
FROM EMPLOYEES
WHERE JOB_ID LIKE '%CLERK';
```

Result:

```
Rank
----------
      12
```

Using NULL-Related Functions in Queries

The **NULL**-related functions facilitate the handling of **NULL** values..

The query in Example 4-43 returns the last name and commission of the employees whose last names begin with **'B'**. If an employee receives no commission (that is, if**COMMISSION_PCT** is **NULL**), the **NVL** function substitutes "Not Applicable" for **NULL**.

Example 12: Substituting a String for a NULL Value

```
SELECT LAST_NAME,
NVL(TO_CHAR(COMMISSION_PCT), 'Not Applicable') "COMMISSION"
FROM EMPLOYEES
WHERE LAST_NAME LIKE 'B%'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .1
Bates                     .15
Bell                      Not Applicable
Bernstein                 .25
Bissot                    Not Applicable
```

Bloom .2
Bull Not Applicable

9 rows selected.

The query in Example 4-44 returns the last name, salary, and income of the employees whose last names begin with **'B'**, using the **NVL2** function: If **COMMISSION_PCT** is not**NULL**, the income is the salary plus the commission;
if **COMMISSION_PCT** is **NULL**, income is only the salary.

Example 13: Specifying Different Expressions for NULL and Not NULL Values

```
SELECT LAST_NAME, SALARY,
NVL2(COMMISSION_PCT, SALARY + (SALARY * COMMISSION_PCT), SALARY) INCOME
FROM EMPLOYEES WHERE LAST_NAME LIKE 'B%'
ORDER BY LAST_NAME;
```

Result:

```
LAST_NAME                 SALARY    INCOME
------------------------- ---------- ----------
Baer                       10000      10000
Baida                       2900       2900
Banda                       6200       6882
Bates                       7300       8468
Bell                       4000       4000
Bernstein                   9500      11970
Bissot                      3300       3300
Bloom                       10000      12100
Bull                       4100       4100

9 rows selected.
```

UNION Example

The following statement combines the results with the **UNION** operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the **TO_CHAR** function) when columns do not exist in one or the other table:

```
SELECT location_id, department_name "Department",
  TO_CHAR(NULL) "Warehouse"  FROM departments
   UNION
   SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
   FROM warehouses;

LOCATION_ID Department          Warehouse
```

---------- -------------------- -------------------------
1400 IT
1400 Southlake, Texas
1500 Shipping
1500 San Francisco
1600 New Jersey
1700 Accounting
1700 Administration
1700 Benefits
1700 Construction
.
.
.

UNION ALL Example

The **UNION** operator returns only distinct rows that appear in either result, while the **UNION ALL** operator returns all rows. The **UNION ALL** operator does not eliminate duplicate selected rows:

```
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories;

SELECT location_id  FROM locations
UNION ALL
SELECT location_id  FROM departments;
```

A **location_id** value that appears multiple times in either or both queries (such as '**1700**') is returned only once by the **UNION** operator, but multiple times by the **UNION ALL** operator.

INTERSECT Example

The following statement combines the results with the **INTERSECT** operator, which returns only those rows returned by both queries:

```
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items;
```

MINUS Example

The following statement combines results with the **MINUS** operator, which returns only rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items;
```

Suppose that you want to select the **FIRST_NAME**, **LAST_NAME**, and **DEPARTMENT_NAME** of every employee. **FIRST_NAME** and **LAST_NAME** are in the **EMPLOYEES** table, and**DEPARTMENT_NAME** is in the **DEPARTMENTS** table. Both tables have **DEPARTMENT_ID**. You can use the query in Example 4-16. Such a query is called a **join**.

Experiment No 7&8

Objectives:

- Queries for (Equi-join,Non-Equi join,outer join)
- Sub queries, Nested queries

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's **FROM** clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain **WHERE** clause conditions that compare two columns, each from a different table. Such a condition is called a **join condition**. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to **TRUE**. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, in the case of the cost-based optimization approach, statistics for the tables.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns.

Example 4-16 Selecting Data from Two Tables (Joining Two Tables)

```
SELECT EMPLOYEES.FIRST_NAME "First",
EMPLOYEES.LAST_NAME "Last",
```

```
DEPARTMENTS.DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

Result:

```
First                Last                   Dept. Name
-------------------- ---------------------- ------------------------------
Jennifer             Whalen                 Administration
Michael              Hartstein              Marketing
Pat                  Fay                    Marketing
Den                  Raphaely               Purchasing
Karen                Colmenares             Purchasing
Alexander            Khoo                   Purchasing
Shelli               Baida                  Purchasing
Sigal                Tobias                 Purchasing
Guy                  Himuro                 Purchasing
Susan                Mavris                 Human Resources
Donald               OConnell               Shipping

First                Last                   Dept. Name
-------------------- ---------------------- ------------------------------
Douglas              Grant                  Shipping
...
Shelley              Higgins                Accounting

106 rows selected.
```

Table-name qualifiers are optional for column names that appear in only one table of a join, but are required for column names that appear in both tables. The following query is equivalent to the query in Example 4-16:

```
SELECT FIRST_NAME "First",
LAST_NAME "Last",
DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

To make queries that use qualified column names more readable, use table aliases, as in the following example:

```
SELECT FIRST_NAME "First",
LAST_NAME "Last",
DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES e, DEPARTMENTS d
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

Although you create the aliases in the **FROM** clause, you can use them earlier in the query, as in the following example:

```
SELECT e.FIRST_NAME "First",
e.LAST_NAME "Last",
d.DEPARTMENT_NAME "Dept. Name"
FROM EMPLOYEES e, DEPARTMENTS d
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

Create the tables with the appropriate integrity constraints and Insert around 10 records in each of the tables

SQL> create table customer1 (cust_id number(5) primary key, cust_name varchar2(15));
Output: Table created.

SQL> desc customer1;

Output:

Name	Null?	Type
CUST_ID	NOT NULL	NUMBER(5)
CUST_NAME		VARCHAR2(15)

Valid Test Data

b) SQL> insert into customer1 values(&custid,'&custname');
SQL> select * from customer1;
Output:

CUST_ID	CUST_NAME
100	ramu
101	kamal
102	raju
103	raju sundaram
104	lawrence

SQL> create table item(item_id number(4) primary key,
item_name varchar2(15),price number(6,2));
SQL> dsec item

Output:

Name	Null?	Type
Cust_id	NOT NULL	NUMBER(4)
Item_name		VARCHAR2(15)
PRICE		NUMBER(6,2)

SQL>insert into item values(&item_id,'&item_name',&price);

SQL> select * from item;

Output:

ITEM_ID	ITEM_NAME	PRICE
2334	geera	6.25
4532	corn soup	34.65
2124	lays chips	20
4531	setwet	99.99
2319	duracell	45.5

SQL>create table sale(bill_no number(5) primary key,bill_date date, cust_id number(5) references customer(cust_id), item_id number(4) references item(item_id),qty_sold number(4));

Out put: Table Created.

SQL>dsec sale

Output:

Name	Null?	Type
BILL_NO	NOT NULL	NUMBER(4)
BILL_DATE		DATE
CUST_ID		NUMBER(5)
ITEM_ID		NUMBER(4)
QTY_SOLD		NUMBER(4)

SQL>insert into Sale values(&bill_no, '&bill_date',
&cust_id, &item_id, &qty_sold);

SQL>select * from sale;

Output:

BILL_NO	BILL_DATE	CUST_ID	ITEM_ID	QTY_SOLD
1450	04-JAN-06	100	2124	2
1451	04-JAN-06	101	2319	1
1452	04-JAN-06	103	4531	2
1453	04-JAN-06	102	2334	3
1454	04-JAN-06	104	4532	3

c) List all the bills for the current date with the customer names and item numbers

SQL> select c.custname, i.itemid, s.billno from customer c, item I, sale s
where c.custid=s.custid and
s.billdate=to_char(sysdate);

CUSTNAME	ITEMID	BILLNO
John	5001	332

d) List the total Bill details with the quantity sold, price of the item and the final amount
SQL> select i.price, s.qty,(i.price*s.qty) total from item I, sale s where i.itemid=s.itemid;

PRICE	QTY	TOTAL
120	2	240
20	3	60
5	2	10
10	1	10
350	4	1400

e) List the details of the customer who have bought a product which has a price>200
SQL> select c.custid, c.custname from customer c, sale s, item i where i.price>200 and c.custid=s.custid and i.itemid=s.itemid;

CUSTID	CUSTNAME
4	duffy

f) Give a count of how many products have been bought by each customer
SQL> select custid, count(itemid) from sale group by custid;

CUSTID	COUNT(ITEMID)
1	2
3	1
4	1
5	1

g) Give a list of products bought by a customer having cust_id as 5
SQL> select i.itemname from item i, sale s where s.custid=5 and i.itemid-s.itemid;

ITEMNAME
Pens

h) List the item details which are sold as of today
SQL> select i.itemid, i.itemname from item I, sale s where i.itemid=s.itemid and s.billdate=to_char(sysdate);

ITEMID	ITEMNAME
1234	Pencil

Outer join

An **outer join** does not require each record in the two joined tables to have a matching record. The joined table retains each record—even if no other matching record exists. Outer joins subdivide further into left outer joins, right outer joins, and full outer joins, depending on which table's rows are retained (left, right, or both).

(In this case *left* and *right* refer to the two sides of the JOIN keyword.)

No implicit join-notation for outer joins exists in standard SQL.

Left outer join

The result of a *left outer join* (or simply **left join**) for tables A and B always contains all records of the "left" table (A), even if the join-condition does not find any matching record in the "right" table (B). This means that if the ON clause matches 0 (zero) records in B (for a given record in A), the join will still return a row in the result (for that record)—but with NULL in each column from B. A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table.

For example, this allows us to find an employee's department, but still shows the employee(s) even when they have not been assigned to a department (contrary to the inner-join example above, where unassigned employees were excluded from the result).

Example of a left outer join (the **OUTER** keyword is optional), with the additional result row (compared with the inner join) italicized:

```
SELECT *
FROM employee LEFT OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
John	NULL	NULL	NULL
Heisenberg	33	Engineering	33

Alternate syntaxes

Oracle supports the deprecated syntax:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID = department.DepartmentID(+)
```

Sybase supports the syntax:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID *= department.DepartmentID
```

IBM Informix supports the syntax:

```
SELECT *
FROM employee, OUTER department
WHERE employee.DepartmentID = department.DepartmentID
```

Right outer join

A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those records that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Below is an example of a right outer join (the **OUTER** keyword is optional), with the additional result row italicized:

```
SELECT *
FROM employee RIGHT OUTER JOIN department
  ON employee.DepartmentID = department.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

Right and left outer joins are functionally equivalent. Neither provides any functionality that the other does not, so right and left outer joins may replace each other as long as the table order is switched.

Alternate syntaxes

Oracle supports the deprecated syntax:

```
SELECT *
FROM employee, department
WHERE employee.DepartmentID(+) = department.DepartmentID
```

Self-join

A self-join is joining a table to itself.

Example

A query to find all pairings of two employees in the same country is desired. If there were two separate tables for employees and a query which requested employees in the first table having the same country as employees in the second table, a normal join operation could be used to find the answer table. However, all the employee information is contained within a single large table.

Consider a modified Employee table such as the following:

Employee Table

EmployeeID	LastName	Country	DepartmentID
123	Rafferty	Australia	31
124	Jones	Australia	33
145	Heisenberg	Australia	33
201	Robinson	United States	34
305	Smith	Germany	34
306	John	Germany	NULL

An example solution query could be as follows:

```
SELECT F.EmployeeID, F.LastName, S.EmployeeID, S.LastName, F.Country
FROM Employee F INNER JOIN Employee S ON F.Country = S.Country
WHERE F.EmployeeID < S.EmployeeID
ORDER BY F.EmployeeID, S.EmployeeID;
```

Which results in the following table being generated.

Employee Table after Self-join by Country

EmployeeID	LastName	EmployeeID	LastName	Country
123	Rafferty	124	Jones	Australia
123	Rafferty	145	Heisenberg	Australia
124	Jones	145	Heisenberg	Australia
305	Smith	306	John	Germany

For this example:

- F and S are aliases for the first and second copies of the employee table.
- The condition F.Country = S.Country excludes pairings between employees in different countries. The example question only wanted pairs of employees in the same country.
- The condition F.EmployeeID < S.EmployeeID excludes pairings where the EmployeeID of the first employee is greater than or equal to the EmployeeID of the second employee. In other words, the

effect of this condition is to exclude duplicate pairings and self-pairings. Without it, the following less useful table would be generated (the table below displays only the "Germany" portion of the result):

EmployeeID	LastName	EmployeeID	LastName	Country
305	Smith	305	Smith	Germany
305	Smith	306	John	Germany
306	John	305	Smith	Germany
306	John	306	John	Germany

Only one of the two middle pairings is needed to satisfy the original question, and the topmost and bottommost are of no interest at all in this example.

Subqueries

Using a Subquery to Solve a Problem
"Who has a salary greater than Jones'?"
"Which employees have a salary greater than Jones' salary?"
"What is Jones' salary?"
Sub queries
SELECT select list
FROM *table*
WHERE *expr operator*
(SELECT select_List
FROM *table*);
•The subquery (inner query) executes once before the main query.
•The result of the subquery is used by the main query (outerquery).

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.
You can place the subquery in a number of SQL clauses
•WHERE clauss
•HAVING clause
•FROM clause
ln the syntax;

operator includes a comparison operatör such as >, =, or IN
Note: Comparison operutors fall iıito two classes: single-row operators
(> , = , >= , < , < > , <=)

and multiple-row operators (IN , ANY , ALL).

Using a Sub query

```
SELECT ename
FROM EMP
WHERE sal >
(SELECT sal
FROM emp
WHERE empno=7566);
```

ENAME
FORD
SCOTT
KING
FORD

Using a Sub query

```
SELECT ename, sal, deptno, job
FROM EMP
WHERE job =
(SELECT job
FROM emp
WHERE empno=7369);
```

ENAME	SAL	DEPTNO	JOB
ADAMS	1100	20	CLERK
JAMES	950	30	CLERK
MILLER	1300	10	CLERK
SMITH	800	20	CLERK
ADAMS	1100	20	CLERK
JAMES	950	30	CLERK
MILLER	1300	10	CLERK

7 rows selected.

```
SELECT enamel, sal, deptno
FROM EMP
WHERE sal IN
( SELECT MIN(sal)
FROM emp
GROUP BY deptno );
```

ENAME	SAL	DEPTNO
JAMES	950	30
SMITH	800	20
MILLER	1300	10

```
SELECT empno, ename, job
FROM emp
WHERE sal < ANY
( SELECT sal
FROM emp
WHERE job = 'CLERK' );
```

EMPNO	ENAME	JOB
7369	SMITH	CLERK
7900	JAMES	CLERK

7876	ADAMS	CLERK
7521	WARD	SALESMAN
7654	MARTIN	SALESMAN

```
SELECT empno, ename, job
FROM emp
WHERE sal < ANY
( SELECT sal
FROM emp
WHERE job = 'CLERK' )
AND job <> 'CLERK' ;
```

EMPNO	ENAME	JOB
7521	WARD	SALESMAN
7654	MARTIN	SALESMAN

```
SELECT empno, ename, job
FROM emp
WHERE sal > ALL
( SELECT AVG(sal)
FROM emp
GROUP BY deptno ) ;
```

EMPNO	ENAME	JOB
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7839	KING	PRESIDENT
7902	FORD	ANALYST

1**Guidelines for Using Subqueries**

•Enclose subqueries in parentheses.
•Place subqueries on the right side of the comparison operator.
•Do not add an ORDER BY clause to a subquery.
•Use single-row operators with singlerow subqueries.
•Use multiple-row operators with

multiple-row subqueries.

Types of Subqueries
· **Single-row subquery**
· **Multiple-row subquery**
· **Multiple-column subquery**

Types of Subqueries

Single-row subgueries: Queries that return only one row from the inner SELECT stntenıent
Muliple-row subqueries: QUERIES that return more than one rows from the inner SELECT statement
Muliple-column subqueries: QUERIES that return more than one column from the inner SELECT statement.

Using Group Functions in a Subquery

```
SELECT ename, sal, deptno
FROM EMP
WHERE sal IN
( SELECT MIN(sal)
FROM emp
GROUP BY deptno ) ;
```

ENAME	SAL	DEPTNO
SMITH	800	20
JAMES	950	30
MILLER	1300	10

HAVING Clause with Subqueries

•The Oracle Server executes sub queries first.
•The Oracle Server returns results into The HAVING clause of the main query.

```
SELECT job, AVG (sal)
FROM emp
GROUP BY job
HAVING AVG(sal) =
( SELECT MIN(AVG(sal))
FROM emp
GROUP BY job);
```

JOB	AVG(SAL)
CLERK	1037,5

Multiple-Row Subqueries
· **Return more than one row**
· **Use multiple-row comparison operators**
Operator Meaning
IN Equal to any member in the list
ANY Compare value to each value returned by the subquery
ALL Compare value to every value returned by the subquery

```
SELECT ename, sal, deptno
FROM emp
WHERE sal IN (SELECT MIN(sal)
```

```
FROM emp
GROUP BY deptno);
```

ENAME	SAL	DEPTNO
SMITH	800	20
JAMES	950	30
MILLER	1300	10

Using ANY Operator in Multiple-Row Subqueries

```
SELECT ename, sal, job
FROM emp
WHERE sal < ANY
( SELECT sal
FROM emp
WHERE job = 'CLERK' )
AND
job <> 'CLERK' ;
```

ENAME	SAL	JOB
WARD	1250	SALESMAN
MARTIN	1250	SALESMAN

Using ALL Operator in Multiple-Row Subqueries

```
SELECT ename, sal, job
FROM emp
WHERE sal > ALL
( SELECT AVG(sal)
FROM emp
GROUP BY deptno );
```

ENAME	SAL	JOB
JONES	2975	MANAGER
SCOTT	3000	ANALYST
KING	5000	PRESIDENT
FORD	3000	ANALYST

1. Write a query to display the employee name and hiredate for all employees in the same department as Blake. Exclude Blake.

```
SELECT ename, hiredate
FROM emp
WHERE deptno =
( SELECT deptno
FROM emp
WHERE ename = 'BLAKE')
AND ename <> 'BLAKE';
```

2. Create a query to display the employee number and name for all employees who earn more than the average salary. Sort the results in descending order of salary.

```
SELECT empno, ename
```

```
FROM emp
WHERE sal >
( SELECT AVG(sal)
FROM emp );
```

3. Write a query to display the employee number and name for all employees who work in a department with any employee whose name contains a T. Save your SQL statemant in a file called p6q3.sql .

```
SELECT empno, ename
FROM emp
WHERE deptno IN
( SELECT deptno
FROM emp
WHERE ename LIKE '%T%' );
```

4. Display the employee name, department number, and job title for all employees whose department location is Dallas.

Solution with subquery:

```
SELECT ename, empno, job
FROM emp
WHERE deptno = (SELECT deptno
FROM dept
WHERE loc ='DALLAS') ;
```

Solution with equijoin:

```
SELECT ename, empno, job
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND d.loc='DALLAS';
```

ENAME	EMPNO	JOB
SMITH	7369	CLERK
JONES	7566	MANAGER
SCOTT	7788	ANALYST
ADAMS	7876	CLERK
FORD	7902	ANALYST

5. Display the employee name and salary of all employees who report to King.

Solution with self join:

```
SELECT e.ename, e.sal
FROM emp e , emp d
WHERE e.mgr = d.empno
AND
d.ename ='KING';
```

Solution with subquery:

```
SELECT ename, sal
FROM emp
WHERE mgr = (SELECT empno
```

```
FROM emp
WHERE ename = 'KING' );
```

6. Display the department number, name,, and job for all employees in the Sales department.

```
SELECT e.deptno, e.ename, e.job , d.dname
FROM emp e , dept d
WHERE e.deptno = d.deptno
AND
d.dname = 'SALES'
```

If yo u have time, complete the following exercises:

7. Modify *p6q3.sql* to display the employee number, name, and salary for all employees who earn more than the average salary and who work in a department with any employee with a T in their name. Rerun your query.

```
SELECT empno, ename , sal
FROM emp
WHERE sal > (SELECT AVG (sal)
FROM emp )
AND
deptno IN ( SELECT deptno
FROM emp
WHERE ename LIKE '%T%');
```

EMPNO	ENAME	SAL
7902	FORD	3000
7788	SCOTT	3000
7566	JONES	2975
7698	BLAKE	2850

Experiment No 9&10

Objective:

- Concept of Commit, Rollback and check points
- creation of views

Properties of Transactions:

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
- **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** enables transactions to operate independently of and transparent to each other.
- **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control:

There are following commands used to control transactions:

- **COMMIT:** to save the changes.
- **ROLLBACK:** to rollback the changes.
- **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK
- **SET TRANSACTION:** Places a name on a transaction.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They can not be used while creating tables or dropping them because these operations are automatically commited in the database.

The COMMIT Command:

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows:

COMMIT;

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example which would delete records from the table having age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
   WHERE AGE = 25;
SQL> COMMIT;
```

As a result, two rows from the table would be deleted and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for ROLLBACK command is as follows:

```
ROLLBACK;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example, which would delete records from the table having age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
    WHERE AGE = 25;
SQL> ROLLBACK;
```

As a result, delete operation would not impact the table and SELECT statement would produce the following result:

```
+----+----------+-----+-----------          +----------+
| ID   | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------          +----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan     |  25 | Delhi          |  1500.00 |
|  3 | kaushik   |  23  | Kota           |  2000.00 |
|  4 | Chaitali  |  25   | Mumbai      |  6500.00 |
|  5 | Hardik    |  27  | Bhopal        |  8500.00 |
|  6 | Komal     |  22  | MP             |  4500.00 |
|  7 | Muffy     |  24  | Indore         | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SAVEPOINT Command:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for SAVEPOINT command is as follows:

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among transactional statements. The ROLLBACK command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as follows:

ROLLBACK TO SAVEPOINT_NAME;

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, here is the series of operations:

```
SQL> SAVEPOINT SP1;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

SQL: VIEWS

The **SQL VIEW** is, in essence, a virtual table. It does not physically exist. Rather, it is created by a SQL query joining one or more tables.

Views can provide advantages over tables:

- Views can represent a subset of the data contained in a table; consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
- Views can join and simplify multiple tables into a single virtual table
- Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the calculated results as part of the data
- Views can hide the complexity of data; for example a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data which it presents
- Depending on the SQL engine used, views can provide extra security

CREATE VIEW

Use the **CREATE VIEW** statement to define a view, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called base tables.

Create a view which lists out the bill_no, bill_date, cust_id, item_id, price, qty_sold, amount

SQL>create view cust as (select s.billno, s.billdate, c.custid, i. iitemid, i.price, s.qty from customer c,sale s item I where c.custid=s.custid and i.iemid=s.itemid);

view created.

SQL>select * from cust;

BILLNO	BILLDATE	CUSTID	ITEMID	PRICE	QTY
3432	12-JAN-06	3	3244	120	2
4424	20-FEB-06	1	3456	20	3
332	13-MAR-06	1	1234	5	2
2343	10-MAR	5	5001	10	1
1331	11-MAR-06	4	76776	350	4

Experiment No 11&12

Objective:

- Use of PL/SQL(Procedures and Function).
- High-level language extension with Cursor and Triggers.

1). Write a program to find largest number from the given three numbers.

Aim: To find largest number from the given three numbers.

Algorithm:

Step 1: Declare the variable A, B, and C.
Step 2: Store the valid data.
Step 3: Compare variable A with B and A with C
Step 4: If the value stored in variable A is big, it displays "A is Big". (IF conditional statement should be used)
Step 5: Compare variable B with C
Step 6: If the value stored in variable B is big, it displays "B is Big".
Step 7: other wise it displays "C is Big"

```
Declare
        A number;
        B number;
        C number;
Begin
        A:=&a;
        B:=&b;
        C:=&c;
If a > b && a> c then
        Dbms_output.put_line(' A is big ');
Else
If( b>c && b> a ) then
```

```
        Dbms_output.put_line(‘ B is big ‘);
Else
        Dbms_output.put_line(‘ C is big ‘);
End if;
End if;
End;
```

Valid Data Sets:

Enter the value of a:
1
Enter the value of b:
2
Enter the value of c:
3

OUTPUT:

C is big

Invalid Data sets :

Enter the value of a:
y
Enter the value of b:
x
Enter the value of c:
a

Output:

Invalid data types.

2). Simple programs using loop, while and for iterative control statement.

a) To generate first 10 natural numbers using **loop, while** and **for**.

AIM: To generate first 10 natural numbers using loop, while and for.

Algorithm:

Step 1: Declare the variable I.
Step 2: Store the valid data 1 in I.
Step 3: Use **LOOP** statement
Step 4: Display the first value.

Step 5: Increment the value of I by 1 value.
Step 6: check the value up to 10 no. and repeat the loop
Step 7: If condition exceeds the given value 10, the loop will be terminated.

/* using loop statement */

```
Declare

I number;

Begin

        I:=1;

Loop

        Dbms_output.put_line(I);

        I:=I+1;

        Exit when I>10;

End loop;

End;
```

Algorithm: for WHILE loop

Step 1: Declare the variable I.
Step 2: Store the valid data 1 in I.
Step 3: Use **WHILE** statement
Step 4: Check the value of I with value 10.
Step 5: if the value of I reached to 10 the loop will be terminated
Step 6: otherwise display value of I
Step 7: increment the next value of I using I=I+1.

/* using while */

```
Declare

I number;

Begin

        I:=1;
```

```
While (I<=10)
loop
        Dbms_output.put_line(I);
        I:=I+1;
End loop;
End;
```

Algorithm:

Step 1: Declare the variable I.
Step 2: Store the value 1 in var. I.
Step 3: Use **For**… **LOOP** statement
Step 4: Display the first value of I.
Step 5: Increment the value of I by 1 value.
Step 6: check the value up to 10 no. and repeat the loop
Step 7: if the loop exceeds the value 10 then the loop will be terminated.

/* using for loop*/

```
Begin

For I in 1..10

loop

        Dbms_output.put_line(I);

End loop;

End;
```

Valid Test Data:

OUTPUT

1
2
3
4
5
6
7
8
9
10

3. Program to check whether given number is Armstrong or not.

AIM: to check whether given number is Armstrong or not.

Algorithm:

Step 1: Declare the variable N, S, D and DUP.
Step 2: Store the value in var. N and var. DUP..
Step 3: check for the value of N, which is not equal to 0.
Step 4: divide value stored in N by 10 and store it var. D. (D=n%10).
Step 5: the reminder will be multiply 3 times and store it in Var. S.
Step 6: The coefficient will be calculated using FLOOR function. And store it in var. N.
Step 7: repeat the Steps 3, 4, 5, and 6 till loop will be terminated.
Step 8: Check whether the stored value and calculated values are same
Step 9: if both the values are same, then display "The given number is Armstrong"
Step 10: Otherwise display "it is not Armstrong" and terminate the loop.

```
Declare

        N number;
        S number;
        D number;

Begin

        N:=&n;

        S:=0;

While(n!=0)

Loop

        D=n%10;
        S:=s+(D*D*D);
        N:=floor(n/10);

End loop;

If (DUP=S) then

        DBMS_output.put_line('number is armstrong');
```

Else

DBMS_output.put_line('number is not armstrong');

End if;

End;

Test Valid Data Set:

Enter value of n

153

OUTPUT:

number is Armstrong.

---

4. Write a program to generate all prime numbers below 100.

AIM: to generate all prime numbers below 100.

Declare

I number;

J number;

C number;

Begin

While(i<=100)

Loop

C:=0;
J:=1;

While(j<=i)

Loop

If(floor(i%j)=0) then

C:= C+1;

End if;

J:=j+1;

```
        End loop;
If(c=2) then
        Dbms_output.put_line(i);
End if;
Endloop;
End;
```

Valid Test Data

OUTPUT:

2

3

5

7

11

.

.

99

5. Create a Cursor which update the salaries of an Employee as follows.

1. if sal<1000then update the salary to 1500.

2. if sal>=1000 and <2000 then update the salary to 2500.

3. if sal>=2000 and <=3000 then update the salary to 4000.

And also count the no.of records have been updated.*/

```
Declare
Cursor my_cur is select empno,sal from emp;
Xno emp.empno%type;
Xsal emp.sal%type;
C number;
```

```
Begin
Open my_cur;
C:=0;
Loop
Fetch my_cur into xno,xsal;
If(xsal<1000) then
Update emp set sal=3000 where empno=xno;
C:=c+1;
Else if(xsal>=2000) and xsa<3000) then
Update emp set sal=4000 where empno=xno;
C:=c+1;
End if;
End if;
Exit when my_cur%NOTFOUND     ;
End loop;
Close my_cur;
Dbma_output.put_line(c||'records have been successfully updated');
End;
Sql>@a.sql;
records have been successfully updated
pl/sql procedure successfully completed.
```

Valid Test Data

Before executing the cursor, the records in emp table as follows

```
Sql>select * from emp;
```

OUTPUT:

EMPNO ENAME JOB MGR HIREDATE SAL COMMD EPTNO

7369 SMITH CLERK 7902 17-DEC-80 2000 20

7499 ALLEN SALESMAN 7698 20-FEB-81 1600 300 30

7521 WARD SALESMAN 7698 22-FEB-81 1250 500 30

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

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

7566 JONES MANAGER 7839 02-APR-81 2975 20

7654 MARTIN SALESMAN 7698 28-SEP-81 1250 1400 30

7698 BLAKE MANAGER 7839 01-MAY-81 2850 30

…

….

…

14 rows selected.

6. create a procedure which generate all the prime numbers below the given number and count the no.of prime numbers.

```
Create or replace procedure prime_proc(n IN number,tot OUT number) as
        i number;
        c number;
        j number;
Begin
        i:=1;
        tot:=0;
while(i<=n)
        loop
        j:=1;
```

```
        c:=0;
while(j<=i)
loop
        if(mod(I,j)=0) then
                c:=c+1;
        end if;
        j:=j+1;
end loop;
if(c=2) then
        dbms_output.put_line(i);
        tot:=tot+1;
end if;
i:=i+1;
end loop;
end;
/
Sql>procedure created.
declare
t number;
begin
prime_proc(10,t);
dbms_output.put_line('the total prime no .are'||t);
end;
```

Valid Test Data:

```
sql>set serveroutput on
```

OUTPUT

sql>/

2

3

5

7

The total prime no.are 4

Pl/sql procedure successfully completed.

7. create a procedure which updates the salaries of an employees as follows.

1.if sal<1000 then update the salry to 1500.

2.if sal>=1000 and <=2400 then update the salary to 2500.*/

Create or replace procedure myproc as

Cursor my_cur is select empno,sal from emp;

Xno emp.empno%type;

Xsal emp.sal%type;

C number;

Begin

Open my_cur;

C:=0;

Loop

Fetch my_cur into xno,xsal;

If(xsal<1000) then

Update emp set sal=1500 where empno=xno;

C:=c+1;

Else

```
Is(xsal>=1000 and xsal<=2400) then

Update emp set sal=2500 where empno=xno;

C:=c+1;

End if;

End if;

Exit when my_cur%NOTFOUND;

End loop;

Close my_cur;

Dbms_output.put_line(c||'records have been successfully updated');

End;

/
```

Valid Test Data:

Procedure created.

Sql>exec myproc;

OUTPUT:

Records have been successfully completed.

```
/* create function which add two given numbers. (Simple programs) */

Create or replace function add_fun(a number,b number) return

Number as

C number;

Begin

C:=a+b;

Return c;

End;
```

/

Function created.

/*add_fun specification*/

Declare

Result number;

Begin

Result:=add_fun(10,20);

Dbms_output.put_line('the sum of 10 and 20 is'||result);

End;

Sql>/

The sum of 10 and 20 is 30

Pl/sql procedure successfully completed.

/*create a function which count total no.of employees having salary less than 6000.*/

/*function body*/

Create or replace function count_emp(esal number)return number as

Cursor vin_cur as Select empno,sal from emp;

Xno emp.empno%type;

Xsal emp.sal%type;

C number;

Begin

Open vin_cur;

C:=0;

loop

```
fetch vin_cur into xno,xsal;
if(xsal<esal) then
c:=c+1;
end if;
   exit when vin_cur%notfound;
end loop;
close vin_cur;
return c;
end;
/
```

Function created.

```
/*function specification*/
Declare
Ne number;
Xsal number;
Begin
Ne:=count_emp(xsal);
Dbms_output.put_line(xsal);
Dbma_output.put_line('there are '||ne||';employees');
End;
/
```

OUTPUT

There are 8 employees.

Triggers:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers:

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.

- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Examples

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

Trigger created.

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.