DBMS LAB MANUAL

1. **Queries to facilitate acquaintance of Built-In Functions, String Functions, Numeric Functions, Date Functions and Conversion Functions.**

Built-In functions are used in SQL SELECT expressions to calculate values and manipulate data. These functions can be used anywhere expressions are allowed and examples include string functions, numeric functions, date functions, conversion functions etc.

STRING FUNCTIONS:

1) Concatenation: Combines Two Strings

```
SQL> SELECT CONCAT ('VIGNAN','LARA') FROM DUAL;

                    CONCAT ('VI
                    --------------------

                    VIGNANLARA
```

2)LPAD(CHAR1,N,CHAR2): Char1 left padded to length N by char2

```
SQL> SELECT LPAD('ANIL',10,'KUMAR') FROM DUAL;

                    LPAD('ANIL')

                    ----------

                    KUMARKANIL
```

3)RPAD(CHAR1,N,CHAR2): Right pad the char1 to length with sequence of characters in char2

```
SQL> SELECT RPAD('ANIL',10,'KUMAR') FROM DUAL;

                    RPAD('ANIL

                    ----------

                    ANILKUMARK
```

4)LTRIM(CHAR,SET): Trims character from the left side.

```
SQL> SELECT LTRIM('NANDINI','N') FROM DUAL;

          LTRIM(

          ------

          ANDINI
```

5)LOWER:converts all characters into lower case

```
SQL> SELECT LOWER('VIGNAN') FROM DUAL;
```

LOWER

---

vignan

6)UPPER: Converts all characters into upper case

SQL> SELECT UPPER('VIGNAN') FROM DUAL;

UPPER

VIGNAN

7)INITCAP(CHAR):first letter of word is capitalized.

SQL> SELECT INITCAP('VIGNAN') FROM DUAL;

INITCA

Vignan

8)LENGTH(CHAR):LENGTH of string

SQL> SELECT LENGTH('ANIL KUMAR')FROM DUAL;

LENGTH('ANILKUMAR')

10

9)SUBSTRING: The SUBSTR function returns part of a string.

SQL> SELECT SUBSTR('11-MAY-2015',4,3) FROM DUAL;

SUB

MAY

10)INSTRING:

SQL> SELECT INSTR('ABCABDEBD','BD',1,2) FROM DUAL;

INSTR('ABCABDEBD','BD',1,2)

8

DATE FUNCTIONS:

1) SYSDATE: Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format.

```
SQL> SELECT TO_CHAR(SYSDATE) FROM DUAL;

        TO_CHAR(SYSDATE)

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

        12-NOV-15
```

2)ADD MONTHS: Adding months to SYSDATE

```
SQL> SELECT TO_CHAR(ADD_MONTHS(SYSDATE,3)) FROM DUAL;

            TO_CHAR(ADD_MONTHS

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

            12-FEB-16
```

3)LAST DAY: Returns the date of the last day of the month that contains ***date***.

```
SQL> SELECT TO_CHAR(LAST_DAY(SYSDATE)) FROM DUAL;

            TO_CHAR(LAST_DAY(S

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

        30-NOV-15
```

4)NEXT DAY: This function returns the date of the day of the week following a particular date.

```
SQL> SELECT TO_CHAR(NEXT_DAY('27-AUG-15','MON')) FROM DUAL;

            TO_CHAR(NEXT_DAY('

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

            31-AUG-15
```

5)MONTHS BETWEEN: To find the time difference between two dates, use the MONTHS_BETWEEN function.

The MONTHS_BETWEEN function returns fractional months.

```
SQL> SELECT TO_CHAR(MONTHS_BETWEEN('06-AUG-15','06-AUG-16')) FROM DUAL;

        TO_CHAR(MONTHS_BETWEEN('06-AUG-15','06-A

                                -12
```

CONVERSION FUNCTIONS:

1)TO_CHAR: Convert a numeric or date expression to a character String.

```
SQL> SELECT TO_CHAR(SYSDATE,'YYYY/MM/DD') FROM DUAL;

TO_CHAR(SY
----------
2015/11/12

SQL> SELECT TO_CHAR(1000,'$9999') FROM DUAL;

TO_CHA
------
 $1000

SQL> SELECT TO_CHAR(1010,'9999') FROM DUAL;

TO_CH
-----
 1010
```

2)TO_NUMBER: Convert a string expression to a number

```
SQL> SELECT TO_NUMBER(1234.98,'9999.99') FROM DUAL;

                    TO_NUMBER(1234.98,'9999.99')

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

                                        1234.98
```

3)TO_DATE: Convert an expression to a date value.

```
SQL> SELECT TO_DATE('20100811','YYYYMMDD') FROM DUAL;

                    TO_DATE('

                      ---------

                     11-AUG-10

SQL> SELECT TO_DATE('2005 120 0540','YYYY DDD SSSSS') FROM DUAL;

                         TO_DATE('

                           ---------

                          30-APR-05
```

NUMERIC FUNCTIONS:

SQL numeric functions are used primarily for numeric manipulation and/or mathematical calculations.

1) ABS: Absolute value of the number
```
SELECT ABS(12) FROM DUAL;
  ABS(12)
---------
      12
```

2) CEIL: Integer value that is Greater than or equal to the number
```
SQL> SELECT CEIL(48.99) FROM DUAL;
CEIL(48.99)
-----------
        49
SQL> SELECT CEIL(48.11) FROM DUAL;
CEIL(48.11)
-----------
        49
```
3) FLOOR: Integer value that is Less than or equal to the number
```
SQL> SELECT FLOOR(49.99) FROM DUAL;
FLOOR(49.99)
------------
        49
SQL> SELECT FLOOR(49.11) FROM DUAL;
FLOOR(49.11)
------------
        49
```

4) ROUND: Rounded off value of the number 'x' up to the number 'y' decimal places
```
SQL> SELECT ROUND(49.11321,2) FROM DUAL;
ROUND(49.11321,2)
-----------------
          49.11
SQL> SELECT ROUND(49.11321,3) FROM DUAL;
ROUND(49.11321,3)
-----------------
         49.113
```
5)
```
SQL> SELECT ROUND(49.11321,4) FROM DUAL;
ROUND(49.11321,4)
-----------------
        49.1132
```

2. Queries using operators in SQL.

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Book table:

SQL> select * from book;

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20
124	MAHABARATH	500	EPIC	100	25

PUBLISHER TABLE:

SQL> select * from publisher;

PID	PNAME	ADDRESS
12	CHETAN	BANGLORE
15	BHAGAT	BANGLORE
18	MAHI	HYDERABAD
20	TEJA	DELHI

ANY OERATOR: ANY operator compares a value with any of values written by sub query. This operator returns a false value if the sub query returns a tuple.

Q:Retrive the details of book with price equal to any of the books belonging to the novel category

SQL> SELECT * FROM BOOK WHERE PRICE=ANY(SELECT PRICE FROM BOOK WHERE CATEGORY='NOVEL');

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17

ALL OPERATOR:ALL operator compares a value to every value in a list returned by the sub query

Q: Retrive the details of books with price greater than the price of all books belonging to epic category

SQL> SELECT * FROM BOOK WHERE PRICE > ALL(SELECT PRICE FROM BOOK WHERE CATEGORY=

'EPIC');

no rows selected

IN OPERATOR: The IN operator is used to specify the list of values.the IN operator selects values that match any value in the given list of values.

Q:Retrive the book details belonging to the category novel

SQL> SELECT * FROM BOOK WHERE CATEGORY IN('NOVEL');

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17

Q: Retrive the details of books belonging to the category with the p_count>50

SQL> SELECT * FROM BOOK WHERE CATEGORY IN(SELECT CATEGORY FROM BOOK WHERE P_COUNT>50);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
124	MAHABARATH	500	EPIC	100	25
123	NIGHRT	280	EPIC	50	20

EXIST OPERATOR: It evaluates true if a sub query returns atleast one tuple as a result otherwise it returns a false value.

Q:Retrive the details of publishers having atleast one book publish

SQL> SELECT * FROM PUBLISHER WHERE EXISTS(SELECT * FROM BOOK WHERE PUBLISHER.PID

=BOOK.PID);

O/P:

PID	PNAME	ADDRESS
12	CHETAN	BANGLORE
20	TEJA	DELHI

NOT EXIST OPERATOR: It evaluates true if a subquery returns no tuple as a result.

Q:Retrive the details of publishers having not published any book

SQL> SELECT * FROM PUBLISHER WHERE NOT EXISTS(SELECT * FROM BOOK WHERE PUBLISHER

.PID=BOOK.PID);

O/P:

PID	PNAME	ADDRESS
18	MAHI	HYDERABAD
15	BHAGAT	BANGLORE

UNION OPERATOR: It is used to retrieve tuple from more than one relation and it also eliminate duplicate tuples

Q:Find the union of all tuples with price greater than 200 and all the tuples wuth price less than 450 from book

SQL> (SELECT * FROM BOOK WHERE PRICE>250) UNION (SELECT * FROM BOOK WHERE PRICE<

450);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20
124	MAHABARATH	500	EPIC	100	25

INTERSECT OPERATOR:IT Is used to retrieve the common tuples from more than one relation.

Q:Find the intersection of all the tuples with price>20 and all the tuples with price<450 from book relation.

SQL> (SELECT * FROM BOOK WHERE PRICE>200) INTERSECT(SELECT * FROM BOOK WHERE PRI

CE<450);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20

3. **Queries to Retrieve and Change Data: Select, Insert, Delete, and Update.**

SQL DML(Data Manipulation Language) commands:

SELECT – retrieve data from the a database.
INSERT – insert data into a table.
UPDATE – updates existing data within a table.
DELETE – Delete all records from a database table.

Before using the above commands in queries, one should create database(table). The following query is used to create a table:

Syntax: CREATE TABLE tablename (column_name data_ type constraints, …);
Example :
SQL> CREATE TABLE STU123(SNO NUMBER,NAME VARCHAR2(10),BRANCH VARCHAR2(10),AGE INT);

Table created.

SQL> **INSERT INTO** STU123 VALUES(1,'A','CSE',19);

1 row created.

SQL> INSERT INTO STU123 VALUES(2,'B','CSE','');

INSERT INTO STU123 VALUES(2,'B','CSE','')

*ERROR at line 1:

ORA-01400: cannot insert NULL into ("13FE1A0552"."STU123"."AGE")

SQL> **SELECT** * FROM STU123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19

SQL> INSERT INTO STU123 VALUES(2,'B','CSE','20');

1 row created.

SQL> INSERT INTO STU123 VALUES(3,'C','',21);

1 row created.

SQL> SELECT * FROM STU123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19
2	B	CSE	20
3	C		21

UPDATE:

Syntax: UPDATE table_name SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

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

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

DELETE:

4. Queries

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

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 |
+----+----------+-----+-----------+----------+
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS
WHERE ID = 6;
```

using Group By, Order By, and Having Clauses.

Book table:

SQL> select * from book;

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20
124	MAHABARATH	500	EPIC	100	25

EMPLOYEE TABLE:

SQL> SELECT * FROM EMPL;

NAME	IDNO	SALARY
BHANU	3	25000
CHANDU	8	30000
NAVYA	9	10000
CHAKRI	7	15000
VALI	10	25000

GROUP BY: The GROUP BY clause when used in a select command divides the relation into groupson the basis of value of one or more attributes.

Q1) Calculate the avg price for each category of book in book relation?

SQL> SELECT AVG(PRICE) FROM BOOK GROUP BY CATEGORY;

AVG(PRICE)
390
225

Q2)calculate the group by salary,count,avg(salary) from empl table group by salary?

SQL> select salary,count(*),avg(salary) from empl group by salary;

SALARY	COUNT(*)	AVG(SALARY)
10000	1	10000
30000	1	30000
15000	1	15000
25000	2	25000

ORDER BY: used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Q3) Select *from empl order by name;

NAME	IDNO	SALARY

BHANU	3	25000
CHAKRI	7	15000
CHANDU	8	30000
NAVYA	9	10000
VALI	10	25000

HAVING: Conditions can be placed on the groups using the HAVING clause.

Q4) Retrive the details of salary,avg(salary),count from empl group by salary where salary,count>=2?

SQL> SELECT SALARY,COUNT(*),AVG(SALARY) FROM EMPL GROUP BY SALARY HAVING COUNT(S

ALARY)>=2;

SALARY	COUNT(*)	AVG(SALARY)
---	---	---
25000	2	25000

5. Queries on Controlling Data: Commit, Rollback, and Save point

Transaction control statements manage changes made by DML statements. A transaction is a set of SQL statements which Oracle treats as a Single Unit. i.e. all the statements should execute successfully or none of the statements should execute.

To control transactions Oracle does not made permanent any DML statements unless you commit it. If you don't commit the transaction and power goes off or system crashes then the transaction is roll backed.

TCL Statements available in Oracle are:

COMMIT : Make changes done in transaction permanent.
ROLLBACK : Rollbacks the state of database to the last commit point.
SAVEPOINT : Use to specify a point in transaction to which later you can rollback.

Example

insert into emp (empno,ename,sal) values (101,'Abid',2300);

commit;

ROLLBACK

To rollback the changes done in a transaction give rollback statement. Rollback restore the state of the database to the last commit point.

Example :

```
delete from emp;

rollback;        /* undo the changes */
```

SAVEPOINT : Specify a point in a transaction to which later you can roll back.

Example

```
insert into emp (empno,ename,sal) values (109,'Sami',3000);
savepoint a;
insert into dept values (10,'Sales','Hyd');
savepoint b;
insert into salgrade values ('III',9000,12000);
```

Now if you give

```
rollback to a;
```

Then row from salgrade table and dept will be roll backed. At this point you can commit the row inserted into emp table or rollback the transaction.

If you give

```
rollback to b;
```

Then row inserted into salgrade table will be roll backed. At this point you can commit the row inserted into dept table and emp table or rollback to savepoint a or completely roll backed the transaction.

If you give

```
rollback;
```

Then the whole transactions is roll backed.

If you give

```
commit;
```

Then the whole transaction is committed and all savepoints are removed.

6. Queries to Build Report in SQL *PLUS

Creating Reports using Command-line SQL*Plus:

In addition to plain text output, the SQL*Plus command-line interface enables you to generate either a complete web page, HTML output which can be embedded in a web page, or data in CSV format. You can use SQLPLUS -MARKUP "HTML ON" or SET MARKUP HTML ON SPOOL ON to produce complete HTML pages automatically encapsulated with <HTML> and <BODY> tags. You can use SQLPLUS -MARKUP "CSV ON" or SET MARKUP CSV ON to produce reports in CSV format.

By default, data retrieved with MARKUP HTML ON is output in HTML, though you can optionally direct output to the HTML <PRE> tag so that it displays in a web browser exactly as it appears in SQL*Plus. See the SQLPLUS MARKUP Options and the SET MARKUP command for more information about these commands.

SQLPLUS -MARKUP "HTML ON" is useful when embedding SQL*Plus in program scripts. On starting, it outputs the HTML and BODY tags before executing any commands. All subsequent output is in HTML until SQL*Plus terminates.

The -SILENT and -RESTRICT command-line options may be effectively used with -MARKUP to suppress the display of SQL*Plus prompt and banner information, and to restrict the use of some commands.

SET MARKUP HTML ON SPOOL ON generates an HTML page for each subsequently spooled file. The HTML tags in a spool file are closed when SPOOL OFF is executed or SQL*Plus exits.

You can use SET MARKUP HTML ON SPOOL OFF to generate HTML output suitable for embedding in an existing web page. HTML output generated this way has no <HTML> or <BODY> tags.

You can enable CSV markup while logging into a user session, by using the -M[ARKUP] CSV ON option at the SQL*Plus command line. For more information, see SQL*Plus Program Syntax. While logged in to a user session, you can enable CSV markup by using the SET MARKUP CSV ON command.

You can specify the delimiter character by using the DELIMITER option. You can also output text without quotes by using QUOTE OFF.

You can suppress display of data returned by a query by using the ONLY option of the SET FEEDBACK command. The number of rows selected and returned by the query is displayed.

Creating HTML Reports

During a SQL*Plus session, use the SET MARKUP command interactively to write HTML to a spool file. You can view the output in a web browser.

SET MARKUP HTML ON SPOOL ON only specifies that SQL*Plus output will be HTML encoded, it does not create or begin writing to an output file. You must use the SQL*Plus SPOOL command to start generation of a spool file. This file then has HTML tags including <HTML> and </HTML>.

When creating a HTML file, it is important and convenient to specify a .html or .htm file extension which are standard file extensions for HTML files. This enables you to easily identify

the type of your output files, and also enables web browsers to identify and correctly display your HTML files. If no extension is specified, the default SQL*Plus file extension is used.

You use SPOOL OFF or EXIT to append final HTML tags to the spool file and then close it. If you enter another SPOOL filename command, the current spool file is closed as for SPOOL OFF or EXIT, and a new HTML spool file with the specified name is created.

You can use the SET MARKUP command to enable or disable HTML output as required.

```
SQL> SELECT '<A HREF="http://oracle.com/'||DEPARTMENT_NAME||'.html">'||DEPARTMENT_NAME||'</A>'
DEPARTMENT_NAME, CITY
2 FROM EMP_DETAILS_VIEW
3 WHERE SALARY>12000;
```

DEPARTMENT	CITY
Executive	Seattle
Executive	Seattle
Executive	Seattle
Sales	Oxford
Sales	Oxford
Marketing	Toronto

```
6 rows selected.

SQL> SPOOL OFF
```

In this example, the prompts and query text have not been suppressed. Depending on how you invoke a script, you can use SET ECHO OFF or command-line -SILENT options to do this.

The SQL*Plus commands in this example contain several items of usage worth noting:

- The hyphen used to continue lines in long SQL*Plus commands.
- The TABLE option to set table WIDTH and BORDER attributes.
- The COLUMN command to set ENTMAP OFF for the DEPARTMENT_NAME column to enable the correct formation of HTML hyperlinks. This makes sure that any HTML special characters such as quotes and angle brackets are not replaced by their equivalent entities, ", &, < and >.
- The use of quotes and concatenation characters in the SELECT statement to create hyperlinks by concatenating string and variable elements.

View the report.html source in your web browser, or in a text editor to see that the table cells for the Department column contain fully formed hyperlinks as shown:

```
<html>
<head>
<TITLE>Department Report</TITLE>  <STYLE type="text/css">
<!-- BODY {background: #FFFFC6} -->  </STYLE>
<meta name="generator" content="SQL*Plus 10.2.0.1">
</head>
<body TEXT="#FF00Ff">
SQL&gt; SELECT '&lt;A HREF=&quot;http://oracle.com/'
```

```
||DEPARTMENT_NAME||'.html&quot;&gt;'||DEPARTMENT_NAME
||'&lt;/A&gt;' DEPARTMENT_NAME, CITY
<br>
  2  FROM EMP_DETAILS_VIEW
<br>
  3* WHERE SALARY&gt;12000
<br>
<p>
<table WIDTH="90%" BORDER="5">
<tr><th>DEPARTMENT</th><th>CITY</th></tr>
<tr><td><A HREF="http://oracle.com/Executive.html">Executive</A></td>
<td>Seattle</td></tr>
<tr><td><A HREF="http://oracle.com/Executive.html">Executive</A></td>
<td>Seattle</td></tr>
<tr><td><A HREF="http://oracle.com/Executive.html">Executive</A></td>
<td>Seattle</td></tr>
<tr><td><A HREF="http://oracle.com/Sales.html">Sales</A></td>
<td>Oxford</td></tr>
<tr><td><A HREF="http://oracle.com/Sales.html">Sales</A></td>
<td>Oxford</td></tr>
<tr><td><A HREF="http://oracle.com/Marketing.html">Marketing</A></td>
<td>Toronto</td></tr>
</table>
<p>

6 rows selected.<br>

SQL&gt; spool off
<br>
</body>
</html>
```

DEPARTMENT_NAME	CITY
Executive	Seattle
Executive	Seattle
Executive	Seattle
Sales	Oxford
Sales	Oxford
Marketing	Toronto

6 rows selected.

7. Queries for Creating, Dropping, and Altering Tables, Views, and Constraints.

SQL CONSTRAINTS:

The following constraints are commonly used in SQL:

NOT NULL - Ensures that a column cannot have a NULL value.
UNIQUE - Ensures that all values in a column are different.
PRIMARY KEY - A combination of a NOT NULL and UNIQUE.
FOREIGN KEY - Uniquely identifies a row/record in another table.

1.Not null

2.unique

3.check

4.primary key

5.Foreign key

1.NOT NULL:

SQL> CREATE TABLE STU123(SNO NUMBER,NAME VARCHAR2(10),BRANCH VARCHAR2(10),AGE INT

NOT NULL);

Table created.

SQL> INSERT INTO STU123 VALUES(1,'A','CSE',19);

1 row created.

SQL> INSERT INTO STU123 VALUES(2,'B','CSE','');

INSERT INTO STU123 VALUES(2,'B','CSE','')

*ERROR at line 1:

ORA-01400: cannot insert NULL into ("13FE1A0552"."STU123"."AGE")

SQL> SELECT * FROM STU123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19

SQL> INSERT INTO STU123 VALUES(2,'B','CSE','20');

1 row created.

SQL> INSERT INTO STU123 VALUES(3,'C','',21);

1 row created.

SQL> SELECT * FROM STU123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19
2	B	CSE	20
3	C		21

2.UNIQUE

These attributes does not accept the duplicate values but it can accept null values.

SQL> CREATE TABLE EMP123(SNO NUMBER,NAME VARCHAR2(10) UNIQUE,AGE VARCHAR2(10));

Table created.

SQL> INSERT INTO EMP123 VALUES(1,'A',21);

1 row created.

SQL> INSERT INTO EMP123 VALUES(2,'A',21);

INSERT INTO EMP123 VALUES(2,'A',21)

ERROR at line 1:

ORA-00001: unique constraint (13FE1A0552.SYS_C0040097) violated

SQL> SELECT * FROM EMP123;

SNO NAME AGE

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

1 A 21

SQL> INSERT INTO EMP123 VALUES(2,'B',21);

1 row created.

SQL> SELECT * FROM EMP123;

SNO NAME AGE

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

1 A 21

2 B 21

3.PRIMARY KEY:(NOT NULL+UNIQUE)

It does not accept duplicate and null values.

SQL> CREATE TABLE EMP124(SNO NUMBER PRIMARY KEY,SALARY INT,JOB VARCHAR2(10));

Table created.

SQL> INSERT INTO EMP124 VALUES(",1000,'A');

INSERT INTO EMP124 VALUES(",1000,'A')

ERROR at line 1:

ORA-01400: cannot insert NULL into ("13FE1A0552"."EMP124"."SNO")

SQL> INSERT INTO EMP124 VALUES(2,1000,'A');

1 row created.

SQL> SELECT * FROM EMP124

SNO SALARY JOB

2 1000 A

NOTE: If already table is in DB

SQL> ALTER TABLE STU123 ADD PRIMARY KEY(SNO);

Table altered.

SQL> INSERT INTO STU123 VALUES(4,'D','CSE',22);

1 row created.

SQL> INSERT INTO STU123 VALUES(4,'E','CSE',22);

INSERT INTO STU123 VALUES(4,'E','CSE’,22);

ERROR at line 1:

ORA-00001: unique constraint (13FE1A0552.SYS_C0040199) violated

4. CHECK:

We can limit the data values to be present in particular attribute.

SQL> select * from stu123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19
2	B	CSE	20
3	C		21
4	D	CSE	22

SQL> alter table stu123 add check(age>=19 and age<=22);

Table altered.

SQL> select * from stu123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19
2	B	CSE	20
3	C		21
4	D	CSE	22

SQL> INSERT INTO STU123 VALUES(5,'E','ECE',25);

INSERT INTO STU123 VALUES(5,'E','ECE',25)

ERROR at line 1:

ORA-02290: check constraint (13FE1A0552.SYS_C0040207) violated

SQL> INSERT INTO STU123 VALUES(5,'E','ECE',22);

1 row created.

SQL> SELECT * FROM STU123;

SNO	NAME	BRANCH	AGE
1	A	CSE	19
2	B	CSE	20
3	C		21
4	D	CSE	22
5	E	ECE	22

8. Queries on Joins and Correlated Sub-Queries

SQL> SELECT * FROM persons;

P_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
1	Hansen	Ola	Timoteivn 10	sandnes
2	Svendson	Tove	Borgn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

SQL> SELECT * FROM orders;

O_ID	ORDERNO	P_ID
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

LEFT JOIN EXAMPLE

SQL> SELECT persons.lastname,persons.firstname,orders.orderno
FROM persons
LEFT JOIN orders ON persons.p_Id = orders.p_Id ORDER BY persons.lastname;

LASTNAME	FIRSTNAME	ORDERNO
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	

FULL OUTER JOIN EXAMPLE

SQL> SELECT persons.lastname,persons.firstname,orders.orderno
FROM persons
FULL OUTER JOIN orders
ON persons.p_Id = orders.p_Id

ORDER BY persons.lastname;

RIGHT OUTTER JOIN EXAMPLE

```
SQL> SELECT persons.lastname,persons.firstname,orders.orderno
FROM persons
RIGHT OUTER JOIN orders
ON persons.p_Id = orders.p_Id
ORDER BY persons.lastname;
```

INNTER JOIN EXAMPLE

```
SQL> SELECT persons.lastname,persons.firstname,orders.orderno
2 FROM persons
3 INNER JOIN orders
4 ON persons.p_Id = orders.p_Id
5 ORDER BY persons.lastname;
```

LASTNAME	FIRSTNAME	ORDERNO
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678

Correlated Sub Queries:

SQL> select * from book;

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20
124	MAHABARATH	500	EPIC	100	25

PUBLISHER TABLE:

SQL> select * from publisher;

PID	PNAME	ADDRESS
12	CHETAN	BANGLORE
15	BHAGAT	BANGLORE
18	MAHI	HYDERABAD
20	TEJA	DELHI

ANY OERATOR: ANY operator compares a value with any of values written by subquery.this ooperator returns a false value if the sub query returns a tuple.

Q:Retrive the details of book with price equal to any of the books belonging to the novel category

SQL> SELECT * FROM BOOK WHERE PRICE=ANY(SELECT PRICE FROM BOOK WHERE CATEGORY='NOVEL');

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17

ALL OPERATOR:ALL operator compares a value to every value in a list returned by the sub query

Q:Retrive the details of books with price greater than the price of all books belonging to epic category

SQL> SELECT * FROM BOOK WHERE PRICE > ALL(SELECT PRICE FROM BOOK WHERE CATEGORY=

'EPIC');

no rows selected

IN OPERATOR: The IN operator is used to specify the list of values.the IN operator selects values that match any value in the given list of values.

Q:Retrive the book details belonging to the category novel

SQL> SELECT * FROM BOOK WHERE CATEGORY IN('NOVEL');

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17

Q:Retrive the details of books belonging to the category with the p_count>50

SQL> SELECT * FROM BOOK WHERE CATEGORY IN(SELECT CATEGORY FROM BOOK WHERE P_COUNT>50);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
124	MAHABARATH	500	EPIC	100	25
123	NIGHRT	280	EPIC	50	20

EXIST OPERATOR: It evaluates true if a sub query returns atleast one tuple as a result otherwise it returns a false value.

Q:Retrive the details of publishers having atleast one book publish

SQL> SELECT * FROM PUBLISHER WHERE EXISTS(SELECT * FROM BOOK WHERE PUBLISHER.PID

=BOOK.PID);

O/P:

PID	PNAME	ADDRESS
12	CHETAN	BANGLORE
20	TEJA	DELHI

NOT EXIST OPERATOR: It evaluates true if a subquery returns no tuple as a result.

Q:Retrive the details of publishers having not published any book

SQL> SELECT * FROM PUBLISHER WHERE NOT EXISTS(SELECT * FROM BOOK WHERE PUBLISHER

.PID=BOOK.PID);

O/P:

PID	PNAME	ADDRESS
18	MAHI	HYDERABAD
15	BHAGAT	BANGLORE

UNION OPERATOR: It is used to retrieve tuple from more than one relation and it also eliminate duplicate tuples

Q: Find the union of all tuples with price greater than 200 and all the tuples wuth price less than 450 from book

SQL> (SELECT * FROM BOOK WHERE PRICE>250) UNION (SELECT * FROM BOOK WHERE PRICE< 450);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
121	MISTAKES	200	NOVEL	30	12
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20

124	MAHABARATH	500	EPIC	100	25

INTERSECT OPERATOR:IT Is used to retrieve the common tuples from more than one relation.

Q:Find the intersection of all the tuples with price>20 and all the tuples with price<450 from book relation.

SQL> (SELECT * FROM BOOK WHERE PRICE>200) INTERSECT(SELECT * FROM BOOK WHERE PRICE<450);

O/P:

ISBN	TITLE	PRICE	CATEGORY	P_COUNT	PID
122	REVOLUTION	250	NOVEL	39	17
123	NIGHRT	280	EPIC	50	20

9. Queries on Working with Index, Sequence, Synonym, Controlling Access, and Locking Rows for Update, Creating Password and Security features

For creating sequences and using the **NEXT VALUE FOR** function to generate sequence numbers, see Sequence Numbers.

Most of the following examples create sequence objects in a schema named Test.

To create the Test schema, execute the following statement.

```
CREATE SCHEMA Test ;
GO
```

A. Creating a sequence that increases by 1

In the following example, Thierry creates a sequence named CountBy1 that increases by one every time that it is used.

```
CREATE SEQUENCE Test.CountBy1
   START WITH 1
   INCREMENT BY 1 ;
GO
```

B. Creating a sequence that decreases by 1

The following example starts at 0 and counts into negative numbers by one every time it is used.

```
CREATE SEQUENCE Test.CountByNeg1
   START WITH 0
```

```
   INCREMENT BY -1 ;
GO
```

C. Creating a sequence that increases by 5

The following example creates a sequence that increases by 5 every time it is used.
Copy

```
CREATE SEQUENCE Test.CountBy1
   START WITH 5
   INCREMENT BY 5 ;
GO
```

D. Creating a sequence that starts with a designated number

After importing a table, Thierry notices that the highest ID number used is 24,328. Thierry needs a sequence that will generate numbers starting at 24,329. The following code creates a sequence that starts with 24,329 and increments by 1.

```
CREATE SEQUENCE Test.ID_Seq
   START WITH 24329
   INCREMENT BY 1 ;
GO
```

To learn commands related to Table Locking:

LOCK TABLE Statement Manually lock one or more tables.
Syntax:
LOCK TABLE [*schema*.] *table* [*options*] IN *lockmode* MODE [NOWAIT]
LOCK TABLE [*schema*.] *view* [*options*] IN *lockmode* MODE [NOWAIT]
Options:
PARTITION (*partition*)
SUBPARTITION (*subpartition*)
@dblink
lockmodes:
EXCLUSIVE
SHARE
ROW EXCLUSIVE
SHARE ROW EXCLUSIVE
ROW SHARE* | SHARE UPDATE*
If NOWAIT is omitted Oracle will wait until the table is available.
Several tables can be locked with a single command - separate with commas
e.g. LOCK TABLE table1,table2,table3 IN ROW EXCLUSIVE MODE;

Default Locking Behaviour :
A pure SELECT will not lock any rows.

INSERT, UPDATE or DELETE's - will place a ROW EXCLUSIVE lock.
SELECT...FROM...FOR UPDATE NOWAIT - will place a ROW EXCLUSIVE lock.

Multiple Locks on the same rows with LOCK TABLE

Even when a row is locked you can always perform a SELECT (because SELECT does not lock any rows) in addition to this, each type of lock will allow additional locks to be grantedas follows.
ROW SHARE = Allow ROW EXCLUSIVE or ROW SHARE or SHARE locks to be granted to the locked rows.
ROW EXCLUSIVE = Allow ROW EXCLUSIVE or ROW SHARE locks to be granted to the locked rows.

SHARE ROW EXCLUSIVE = Allow ROW SHARE locks to be granted to the locked rows.
SHARE = Allow ROW SHARE or SHARE locks to be granted to the locked rows.
EXCLUSIVE = Allow SELECT queries only

Although it is valid to place more than one lock on a row, UPDATES and DELETE's may still cause a *wait* if a conflicting row lock is held by another transaction.

Grant/Revoke Privileges:

Learn how to **grant and revoke privileges** in SQL Server (Transact-SQL) with syntax and examples.

Description

You can GRANT and REVOKE privileges on various database objects in SQL Server. We'll look at how to grant and revoke privileges on tables in SQL Server.

Grant Privileges on Table

You can grant users various privileges to tables. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

Syntax

The syntax for granting privileges on a table in SQL Server is:

```
GRANT privileges ON object TO user;
```

privileges

The privileges to assign. It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
ALL	ALL does not grant all permissions for the table. Rather, it grants the ANSI-92 permissions which are SELECT, INSERT, UPDATE, DELETE, and REFERENCES.

object

The name of the database object that you are granting permissions for. In the case of granting privileges on a table, this would be the table name.

user

The name of the user that will be granted these privileges.

Example

Let's look at some examples of how to grant privileges on tables in SQL Server.

For example, if you wanted to grant SELECT, INSERT, UPDATE, and DELETE privileges on a table called *employees* to a user name *smithj*, you would run the following GRANT statement:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO smithj;
```

You can also use the ALL keyword to indicate that you wish to grant the ANSI-92 permissions (ie: SELECT, INSERT, UPDATE, DELETE, and REFERENCES) to a user named *smithj*. For example:

```
GRANT ALL ON employees TO smithj;
```

If you wanted to grant only SELECT access on the *employees* table to all users, you could grant the privileges to the public role. For example:

```
GRANT SELECT ON employees TO public;
```

Revoke Privileges on Table

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can run a revoke command. You can revoke any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

Syntax

The syntax for revoking privileges on a table in SQL Server is:

```
REVOKE privileges ON object FROM user;
```

privileges

It is the privileges to assign. It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
ALL	ALL does not revoke all permissions for the table. Rather, it revokes the ANSI-92 permissions which are SELECT, INSERT, UPDATE, DELETE, and REFERENCES.

object

The name of the database object that you are revoking privileges for. In the case of revoking privileges on a table, this would be the table name.

user

The name of the user that will have these privileges revoked.

Example

Let's look at some examples of how to revoke privileges on tables in SQL Server.

For example, if you wanted to revoke DELETE privileges on a table called *employees* from a user named *anderson*, you would run the following REVOKE statement:

```
REVOKE DELETE ON employees FROM anderson;
```

PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

S.N.	Sections & Description
1	**Declaration Section** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Section** This section is enclosed between the keywords **BEGIN** and **END**and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the

	program.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

The 'Hello World' Example:

```
DECLARE
  message  varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
Hello World


PL/SQL procedure successfully completed.
```

Following is a valid declaration:

```
DECLARE
  num1 INTEGER;
  num2 REAL;
  num3 DOUBLE PRECISION;
BEGIN
  null;
```

```
END;
/
```

10. Write a PL/SQL Code using Basic Variable, Anchored Declarations, and Usage of Assignment Operation.

Q) Student marks can be selected from the table and printed for those who secured first class and an exception can be raised if no records were found.

SQL>Select *from Stud;

Sno	marks	class
1	49	pass
2	65	first
3	55	second

PLSQL Program:

```
declare
no int;                          //declaration
m int;
c stud.class%type;               //anchored declaration
begin
select sno,marks,class into no,m,c  from stud
where sno=&no and class='first';                //assignment
dbms_output.put_line('----output----');
dbms_output.put_line('student no:'||no);
dbms_output.put_line('marks:'||m);
dbms_output.put_line('class:'||c);
exception
when no_data_found then
 dbms_output.put_line('no matching record exits');
end;
```

11. Write a PL/SQL Code Bind and Substitution Variables. Printing in PL/SQL.

1. Substitution Variables

The clue here is in the name... "substitution". It relates to values being substituted into the code before it is submitted to the database.
These substitutions are carried out by the interface being used. In this example we're going to use SQL*Plus as our interface...

So let's take a bit of code with substitution variables:

```
create or replace function myfn return varchar2 is
  v_dname varchar2(20);
begin
  select dname
  into   v_dname
  from   dept
  where  deptno = &p_deptno;
  return v_dname;
end;
```

Now when this code is submitted...

```
SQL> /
```

SQL*Plus, parses the code itself, and sees the "&" indicating a substitution variable.
SQL*Plus, then prompts for a value for that variable, which we enter...

```
Enter value for p_deptno: 20
old   7:   where  deptno = &p_deptno;
new   7:   where  deptno = 20;
```

... and it reports back that it has substituted the &p_deptno variable for the value 20, actually showing us the whole line of code with it's value.

This code is then submitted to the database. So if we look at what code has been created on the database we see...

```
SQL> select dbms_metadata.get_ddl('FUNCTION', 'MYFN', USER) from dual;
DBMS_METADATA.GET_DDL('FUNCTION','MYFN',USER)
--------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION "SCOTT"."MYFN" return varchar2 is
  v_dname varchar2(20);
begin
  select dname
  into   v_dname
  from   dept
```

```
  where  deptno = 20;
  return v_dname;
end;
```

The database itself knows nothing about any substitution variable... it just has some code, fixed with the value we supplied to SQL*Plus when we compiled it.

The only way we can change that value is by recompiling the code again, and substituting a new value for it.

Also, with substitution variables we don't necessarily have to use them just for 'values' (though that it typically what they're used for)... we can use them to substitute any part of the code/text that we are supplying to be compiled.. e.g.

```
create or replace function myfn(x in number, y in number) return number is
begin
  return &what_do_you_want_to_return;
end;
/
Enter value for what_do_you_want_to_return: y*power(x,2)
old   3:   return &what_do_you_want_to_return;
new   3:   return y*power(x,2);

Function created.

SQL> select dbms_metadata.get_ddl('FUNCTION', 'MYFN', USER) from dual;
DBMS_METADATA.GET_DDL('FUNCTION','MYFN',USER)
--------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION "SCOTT"."MYFN" (x in number, y in number) return
number is
begin
  return y*power(x,2);
end;
```

It really does substitute the substitution variable, with whatever text you supply.

2. Bind Variables

Bind variables are a completely difference concept to substitution variables.

Bind variables typically relate to SQL queries (they can be used in dynamic PL/SQL code, but that's not good practice!), and are a placeholder for values within the query. Unlike substitution variables, these are not prompted for when you come to compile the code.

Now there are various ways of supplying bind variables, and I'll use a couple of examples, but there are more (such as binding when creating queries via the DBMS_SQL package etc.)

In the following example:

```
create or replace function myfn(p_deptno in number) return varchar2 is
  v_dname varchar2(20);
  v_sql   varchar2(32767);
begin
  v_sql := 'select dname from dept where deptno = :1';
  execute immediate v_sql into v_dname using p_deptno;
  return v_dname;
end;
/

Function created.
```

The ":1" is the bind variable in the query.

If you examine queries running in the database you will typically see bind variables represented as :1, :2, :3 and so on, though it could be anything preceded by a ":" such as :A, :B, :C, :X, :FRED, :SOMETHING etc.

When the query is passed to the SQL engine (in this case by the EXECUTE IMMEDIATE statement), the query is parsed and optimised and the best execution plan determined. It doesn't need to know what that value is yet to determine the best plan. Then when the query is actually executed, the value that has been bound in (in this case with the USING part of the execute immediate statement) is used within the execution of the query to fetch the required data.

The advantage of using bind variables is that, if the same query is executed multiple times with different values being bound in, then the same execution plan is used because the query itself hasn't actually changed (so no hard parsing and determining the best plan has to be performed, saving time and resources).

Another example of using bind variable is this:

```
create or replace function myfn(p_deptno in number) return varchar2 is
  v_dname varchar2(20);
begin
  select dname
  into   v_dname
  from   dept
  where deptno = p_deptno;
  return v_dname;
end;
/

Function created.
```

Now, this isn't immediately obvious, but what we have here is the ability of the PL language to seamlessly integrate SQL within it (giving us PL/SQL). It looks as though we just have an SQL

statement in our code, but in reality, the PL engine parses the query and supplies the query to the SQL engine with a bind variable placeholder for where the PL variable (parameter p_deptno in this case) is within it. So the SQL engine will get a query like...

```
select dname
from   dept
where  deptno = :1
```

and then the PL engine will handle the binding of the value (p_deptno) into that query when it executes it, as well as dealing with the returning value being put INTO the PL variable v_dname. Again, the SQL supplied to the SQL engine can be optimised and re-used by code because it isn't hard coded with values.

So, here, the binding of values is implicit because the PL engine is removing the need for us to have to code them explicitly.

The other advantage of using bind variables is that you don't have to worry about the data types.

12. Write a PL/SQL block using SQL and Control Structures in PL/SQL.

a) PL/ SQL Program for IF Condition:

Pl/ sql general syntax for if condition:

```
SQL> DECLARE
<VARIABLE DECLARATION>;
BEGIN
IF(CONDITION)THEN
<EXECUTABLE STATEMENT >;
END;
```

Coding for If Statement:

```
DECLARE
b number;
c number;
BEGIN
B:=10;
C:=20;
if(C>B) THEN
dbms_output.put_line('C is maximum');
end if;
end;
/
```

Output:

C is maximum

PL/SQL procedure successfully completed.

b) PL/ SQL GENERAL SYNTAX FOR IF AND ELSECONDITION:

```
SQL> DECLARE
<VARIABLE DECLARATION>;
BEGIN
IF (TEST CONDITION) THEN
<STATEMENTS>;
ELSE
<STATEMENTS>;
ENDIF;
END;
```

*******************Less then or Greater Using IF ELSE *************************

```
SQL> declare
n number;
begin
dbms_output. put_line('enter a number');
n:=&number;
if n<5 then
dbms_output.put_line('entered number is less than 5');
else
dbms_output.put_line('entered number is greater than 5');
end if;
end;
/
```

Test Case 1:

Input

Enter value for number: 2
old 5: n:=&number;
new 5: n:=2;

Output:

entered number is less than 5
PL/SQL procedure successfully completed.

Test Case 1:

Input

Enter value for number: 6
old 5: n:=&number;
new 5: n:=6;

Output:

entered number is greater than 5
PL/SQL procedure successfully completed.

C) Program using Loops.

While loop syntax: **For loop syntax:**

```
While <condition>
loop
<statements>;
end loop;
```

```
for <loop_counter> in <lowest_number> . .< highest_number>
 loop
<statements>;
end loop;
```

While loop program:

```
declare
i number:=1;
n number;
f number:=1;
begin
n:=&n;
while (i<=n)
loop
f:=f*i;
i:=i+1;
end loop;
dbms_output.put_line(n ||'!='||f);
end;
```

Test Case 1:

Input:
Enter value for n: 4
old 6: n:=&n;
new 6: n:=4;

Output:
4!=24
PL/SQL procedure successfully completed.

Test Case 2:

Input:
Enter value for n: 0
old 6: n:=&n;

new 6: n:=0;

Output:
0!=1
PL/SQL procedure successfully completed.

For loop program:

```
declare

i number;

n number;

f number:=1;

begin

n:=&n;

for i in 1..n

loop

f:=f*i;

end loop;

dbms_output.put_line(n ||'!='||f);

end;
```

Test Case 1:

Input:
Enter value for n: 1
old 6: n:=&n;
new 6: n:=1;
Output:
1!=1
PL/SQL procedure successfully completed.

Test Case 2:

Input:
Enter value for n: 0
old 6: n:=&n;
new 6: n:=0;
Output:
0!=1
PL/SQL procedure successfully completed.

Nested Loops:

```
BEGIN
   FOR v_outerloopcounter IN 1..2 LOOP
         FOR v_innerloopcounter IN 1..4 LOOP
   DBMS_OUTPUT.PUT_LINE('Outer Loop counter is ' || v_outerloopcounter ||' Inner Loop counter is'||
v_innerloopcounter);
         END LOOP;
    END LOOP;
   END;
```

13. Write a PL/SQL Code using Cursors, Exceptions and Composite Data Types

A **cursor** is a pointer to the context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

SQL>SELECT *FROM EMPLOYE2;

ENAME	DEPTNO	SALARY
RAM	1	20000
SAM	2	3000

PLSQL Program for Cursor:

```
declare

cursor c_emp is select ename,deptno,salary from employe2 where

length(salary)=4 for update;

ename1 employe2.ename%type;

dept employe2.deptno%type;

salary1 employe2.salary%type;

BEGIN

open c_emp;

dbms_output.put_line('ename,deptno,salary');

loop

fetch c_emp into ename1,dept,salary1;
```

```
exit when c_emp%notfound;
update employe2 set salary=salary1*2 where current of c_emp;
end loop;
close c_emp;
end;
```

After running the above code the table is:

SQL>SELECT *FROM EMPLOYE2;

ENAME	DEPTNO	SALARY
RAM	**1**	**20000**
SAM	**2**	**6000**

Exceptions:

In PL/SQL, an *error condition* is called an **exception**. Exceptions can be internally defined (by the runtime system) or user defined. Examples of internally defined exceptions include *division by zero* and *out of memory*. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR.

PL/SQL built in exceptions

Following are some built in type exception,

Exception	Error Code	Description
CASE_NOT_FOUND	ORA-06592	Exception raised when no any choice case found in CASE statement as well as no ELSE clause in CASE statement.
DUP_VAL_ON_INDEX	ORA-00001	Exception raised when you store duplicate value in unique constraint column.
INVALID_CURSOR	ORA-01001	Exception raised when you perform operation on cursor and cursor is not really opened.

User Defined exceptions:

Apart from system exceptions we can explicitly define exceptions known as user-defined exceptions. Steps to be followed to use user-defined exceptions:

They should be explicitly declared in the declaration section.
They should be explicitly raised in the Execution Section.
They should be handled by referencing the user-defined exception name in the exception section.

Syntax

```
        DECLARE
        user_define_exception_name     EXCEPTION;
BEGIN
        statement(s);
        IF condition THEN
                RAISE user_define_exception_name;
        END IF;
EXCEPTION
        WHEN user_define_exception_name THEN
                User defined statement (action) will be taken;
END;
```

Example

```
DECLARE
        myex EXCEPTION;
        i NUMBER;
BEGIN
        FOR i IN (SELECT * FROM emp1) LOOP
                IF i.empno = 40 THEN
                        RAISE myex;
                END IF;
        END LOOP;
EXCEPTION
        WHEN myex THEN
        dbms_output.put_line('Employe number already exist in emp1 table.');
END;
/

SQL>SELECT *FROM EMP1;
```

ENAME	EMPNO	SALARY
RAM	**40**	**20000**
SAM	**2**	**6000**

Output:

'Employe number already exist in emp1 table.

PL/SQL procedure successfully completed.

14. Write a PL/SQL Code using Procedures, Functions and Packages.

Procedures are code fragments that don't normally return a value, but may have some outside effects (like updating tables). A procedure is created with the CREATE OR REPLACE PROCEDURE statement.

The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE]

PROCEDURE procedure_name [parameters] IS

BEGIN

procedure_body

END;

Where procedure_name can be any valid SQL name, parameters is a list of parameters to this procedure and procedure_body is various PL/SQL statements that make up the logic of the procedure.

Eg:

Create or replace

Procedure helloworld is

Begin

Dbms_output.put_line('hello world!');

End;

Sql>exec helloworld

helloworld

Parameters are optional. What's the use of a procedure that doesn't take any parameters and doesn't return anything?

Parameter Modes in PL/SQL

S.N.	Parameter Mode & Description

1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value.. **It is the default mode of parameter passing.**
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
2	**IN OUT** An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. **Actual parameter is passed by value.**

Example program

```
create or replace procedure inoutproc

(n1 in int,n2 in int,tot out int)

is

begin

tot:=n1+n2;

end;

/

variable t number

exec inoutproc(2,3,:t);

print t;
```

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

PL/SQL procedure successfully completed.

```
         T
-----------------
         5
```

Example 2

```
Create or replace
Procedure doublen (n in out int) is
Begin
n := n * 2;
End;
```

To run it, we also create a small code fragment:

```
Declare
R int;
Begin
R := 7;
Dbms_output.put_line('before call r is: ' || r);
Doublen(r);
Dbms_output.put_line('after call r is: ' || r);
End;
```

Which when ran displays:

```
BEFORE CALL R IS: 7
AFTER CALL R IS: 14
```

A PL/SQL **function** is same as a procedure except that it returns a value. The general format of a function is very similar to the general format of a procedure:

```
Create or replace
Function    function_name (function_params) return return_type    is
Begin
Function_body
Return something_of_return_type;
End;
```

Example

For example, to write a function that computes the sum of two numbers:

```
Create or replace
Function add_two (a int,b int) return int is
Begin
Return (a + b);
End;
```

To run it, we'll write a small piece of code that calls this:

```
Begin
Dbms_output.put_line('result is: ' || add_two(12,34));
End;
```

Which procudes the output:

Result is: 4

A **package** is a collection of related procedures, functions, variables and data types. A package typically contains two parts – specification and body.

Package **specification** contains declarations for items, procedure and functions that are to be made public. All public objects of package are visible outside the package.

Private items of the package can be used only in the package and not outside the package. The following is the syntax to create package specification.

CREATE PACKAGE package_name is
/* declare public objects of package */
End;

Body of the package defines all the objects of the package. It includes public objects that are declared in package specification and objects that are to be used only within the package – private members.

Syntax:

Create package body package_name is
/* define objects of package */
END;

Package specification:

```
SQL>create or replace package alloperation is
        procedure forinsert(rno number,sname varchar,crc varchar,gen varchar);
        procedure forretrive(rno number);
        procedure forupdate(rno number,sname varchar);
        procedure fordelete(rno number);
        end;

        /

        Package created.
```

Package body:

```
create or replace package body alloperation
 is
 procedure forinsert(rno number,sname varchar,crc varchar,gen varchar)
 is
 begin
 insert into student values(rno,sname,crc,gen);
 end forinsert;
 procedure forretrive(rno number)
 is
 sname student.student_name%type;
 crc student.course%type;
 gen student.gender%type;
 begin
 select student_name,course,gender into sname,crc,gen
 from student where roll_no=rno;
 dbms_output.put_line(sname||' '||crc||' '||gen);
```

```
end forretrive;
procedure forupdate(rno number,sname varchar)
is
begin
update student set student_name=sname where roll_no=rno;
end forupdate;
procedure fordelete(rno number)
is
begin
delete student where roll_no=rno;
end ;
end ;

/

Package body created.
```

Calling procedure of package:

```
Synatx  :    packagename.objectname.

SQL> begin
 alloperation.forinsert(4,'vivek',’ec','male');
 alloperation.forretrive(4);
 alloperation.forupdate(1,'swamy');
 end;

SQL> /
PL/SQL procedure successfully completed.

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

SQL> begin
 alloperation.fordelete(4);
 end;

SQL> /
PL/SQL procedure successfully completed.


SQL> select * from student;

No rows selected.
```

15. Write a PL/SQL Code Creation of forms for any Information System such as Student Information System, Employee Information System etc.

BANK PROJECT AND PL/SQL

KCB_ACC_TAB

```
 1  create table kcb_acc_tab
 2  (
 3  accno number primary key,
 4  name varchar2(20) constraint name_nn not null,
 5  actype char check(actype in('s','c','fd')),
 6  doo date default sysdate,
 7  bal number(8,2) not null
 8* )
QL> /
Table created.
QL> insert into kcb_acc_tab values(37002167543,'srinivas','s',sysdate,15000)
 2  /
 row created.
QL> commit
 2  /
commit complete.
```

KCB_TRAN_TAB

```
create table kcb_tran_tab
(
tid number,
accno number(20) references kcb_acc_tab(accno),
trtype char(10) check(trtype in('d','w')),
dot date default sysdate,
amt number(7,2) check(amt>100)
)
```

SEQUENCE

```
create sequence s1
start with 1
increment by 1
maxvalue 1000
minvalue 0
nocache
nocycle
```

1) Write a PL/SQL program to modify the balance after deposite the amt and insert the transaction details also.

```
declare
i kcb_acc_tab%rowtype;
k kcb_tran_tab%rowtype;
begin
```

```
i.accno:=&accno;
k.trtype:='&trtype';
k.amt:=&amount;
select bal into i.bal from kcb_acc_tab
where accno=i.accno;
if k.trtype='D' then
i.bal:=i.bal+k.amt;
end if;
update kcb_acc_tab set bal=i.bal where accno=i.accno;
insert into kcb_tran_tab values(s1.nextval,i.accno,k.trtype,sysdate,k.amt);
commit;
end;
```

2) write a PL/SQL program for enter the transaction details perform the validation
i)if it is deposite update the bal and insert the transaction details
ii) if it is withdraw before withdraw
check the current bal if validationcontrol satisfy then only
perform the withdraw

```
declare
i kcb_acc_tab%rowtype;
k kcb_tran_tab%rowtype;
begin
i.accno:=&accno;
k.trtype:='&trtype';
k.amt:=&amt;
select actype,bal into i.atype,i.balance from kcb_acc_tab where accno=i.accno;
if k.trtype='D' then
i.bal:=i.bal+k.amt;
else
i.bal:=i.bal-k.amt;
if i.actype='s' and i.bal<5000 then
Raise_application_error(-20456,'the bal is too low to perform transaction');
endif;
update kcb_acc_tab set bal=i.bal
where accno=i.accno;
insert into kcb_tran_tab values(s1.nextval,i.accno,k.trtype,sysdate,k.amt);
commit;
end;
```

PROCEDURE

```
create or replace procedure upd_bal
(paccno kcb_acc_tab.accno%type,
pamt kcb_tran_tab.amt%type)
is
cbal kcb_acc_tab.bal%type;
begin
```

```
select bal into cbal from kcb_acc_tab where accno=paccno;
cbal:=cbal+pamt;
update kcb_acc_tab set bal=cbal where accno=paccno;
insert into kcb_tran_tab values(1001,paccno,'d',sysdate,pamt);
commit;
exception
when no_data_found then
display(paccno||'is not exists');
end upd_bal;


create or replace procedure upd_bal
(paccno kcb_acc_tab.accno%type,
pamt kcb_tran_tab.amt%type)
is
cbal kcb_acc_tab.bal%type;
vatype kcb_acc_tab.atype%type;
begin
select acctype,bal into vatype,cbal from kcb_acc_tab where accno=paccno;
if upper(pttype)='d' then
cbal:=cbal+pamt;
elsif upper(pttype)='w' then
cbal:=cbal-pamt;
if value='s' and cbal<5000 then
Raise_application_error(-20456,'there is insufficient balance so we cannot do the transaction:');
end if;
end if;
update kcb_acc_tab set bal =cbal
where accno=paccno;
insert into kcb_tran_tab
values(101,paccno,ptrtype,sysdate,pamt);
commit;
exception
when no_data_found then
display(paccno||'is not exist');
end upd_bal;
```

FUNCTIONS
---
write a function the account holder is eligible for the withdraw or not

```
create or replace function chk_bal
(paccno kcb_acc_tab.accno%type,
pamt kcb_tran_tab.amt%type)
return boolean
is
cbal kcb_acc_tab.bal%type;
vatype cb_acc_tab.acctype%type;
begin
```

```
select acctype,bal into vacctype,cbal from kcb_acc_tab where
accno=paccno;
cbal:=cbal-pamt;
if vacctype='s' and cbal<5000 then
return(false);
elsif vatype='c'and cbal<10000 then
return(false);
else
return(true);
end if;
end chk_bal;
```

call this function with another pl/sql pgm with appropriate msg.

```
begin
if chk_bal(&accno,&amt)then
display('it is validate');
else
display('it is not validate');
end if;
end;
```

call this function in a procedure for the validation

```
create or replace procedure upd_bal
(paccno kcb_acc_tab.accno%type,
ptrtype kcb_tran_tab.trtype%type,
pamt kcb_acc_tab.amt%type)
is
cbal kcb_acc_tab.bal%type;
begin
select bal into cbal
from kcb_acc_Tab
where accno=paccno;
if upper(ptrtype)='D' then
cbal:=cbal+pamt;
elsif upper(ptrtype)='w' then
if chk_bal(paccno,pamt)then
cbal:=cbal-pamt;
else
Raise_application_error(-20456,'There  IB  so we cannot do the transaction:');
end if;
end if;
update kcb_acc_tab set bal=cbalwhere accno=paccno;
insert into kcb_tran_tab values(101,paccno,ptrtype,sysdate,pamt);
commit;
exception
when no_data_found then
display(paccno||'is not exist');
```

```
end upd_bal;
```

PACKAGES

PACKAGE SPECIFICATION

```
create or replace package pack_updbal
as
cbal bankmaster.curr_bal%type;
procedure upd_bal(vaccno kc b_acc_tab.accno%type,
                  vtype kcb_tran_tab.ttype%type,
                  vamt kcb_tran_tab.amt%type);
function chk_bal(vaccno kcb_acc_tab.accno%type,
                 vamt kcb_tran_tab.amt%type)
return boolean;
cbal.kcb_acc_tab.bal%type;
end pack_updbal;
```

PACKAGE BODY

```
create or replace package body pack_updbal
as
procedure upd_bal(vaccno kcb_acc_tab.accno%type,
                  vtrtype kcb_tran_tab.trtype%type,
vamt kcb_tran_tab.amt%type)
is
begin
select bal into cbal
from kcb_acc_tab
where accno=vaccno;
if upper(vtype)='w' then
cbal:=cbal_vamt;
end if;
update kcb_acc_tab set sal=cbal where accno=vaccno;
commit;
end upd_bal;
function chk_bal(vaccno kcb_acc_tab.accno%type,
                 vamt kcb_tran_tab.amt%type)
return boolean
is
vatype kcb_acc_tab.acctype%type;
begin
select acctype,bal into vatype,cbal from kcb_acc_tab where accno=vaccno;
cbal:=cbal-vamt; (global variable)
if vatype='s' and cbal<5000 then
return(false);
elsif vatype='c' and cbal<10000 then
```

```
return(false);
else
return(true);
end if;
end chk_bal;
end pack_updbal;
```

Triggers

```
create or replace trigger trg_bal
before insert
on kcb_tran_tab
for each row
begin
if :new.trtype='d' then
pack_updbal.upd_bal(:new.accno,:new.trtype,:new.amt);
elsif   :new.trtype='w' then
if pack_updbal.chk_bal(:new.accno,:new.amt)then
pack_updbal.upd_bal(:new.accno,:new.trtype,:new.amt);
else
Raise_application_error(-20451,'the bal is too low so no transaction:');
end if;
end if;
exception
when no_data_found then
display(:new.accno||'is not exists');
end;
```

PL/SQL:

It is a programming language which is developed by oracle company.
It is a procedural language it is used to process only a row at a time where as non procedural laguage process a set of rows at a time.

It support to execute a bloc of stmts at once.
Block: collection of executable statements.

struture of block:

```
Declare
[variable Declaration];
Begin
<executable statements>;
[exception
executable statements];
End;
```

There are two types of blocks
I) Anonoums block

II) named block

Anonmous Blcok:
The Block which is having no name called as anonmous Block
This block cannot call any other programs.
used in D2K forms.

Named Block:
The Block which is having a named called as named block.
This block can call in other PL/SQL programs.
eg: procedure
function
Trigger
package

PL/SQL supports the variables&constraints
SQL will supports the Bind variables only.
eg\; var a number
exec a:=1000
print :a

PL/SQL will support bind variables &list variables.

It support the Error handlings.

In SQL we can see the errors on the program Or select stmt,
But we cannot handle&provide the solution.

Where as in PL/SQL we can handle that errors and provides the
Appropriate actions.

It supports conditional constructs.
It supports the Iteration controls
i)simple loop
ii)while loop
iii)for loop
It supports the sub programs
There are Two types of sub programs:
i)function
ii)proedure

EG:
```
declare
Begin
null;
end;
```

Data types in PL/SQL:

Scalar
Composite
Eg: Table
Record
varray
Reference:
Ref cursor
Ref object_type
LOB

Variable:

variables are used to store datavalues that are used by pl/sql
variables are represents memory locations used to store user or database data.
variables supports the simple data types and composite data types.

Host variables are supports the Boolean Datatypes where as
Bind variables are not supports the Boolean Datatypes.

Syntax: <variable name> datatype(size);

declaration part only u declare the variables.

eg: declare
v_empno number(4):=7902;
v_name varchar2(20) not null;
note: we should not assign the null values.

Assignment operators:
Into: This operator for internal values
:= This operator any external values.

Executable sub languages are:
DQL
DML
TCL

We cannot use DDL,DCL directely in PL/SQL by using dynamic SQL.

Syntax of Select statement:
Select <column list>into <variable list>
from <table name>
where <condition>;

Comments in PL/SQL:

There are Two types of comments:
i)- -single line comment
ii) /* multi line comment */

DBMS_OUT.PUT_LINE('Message'||Variable);

it is used to print the msg and variable value on the screen.

Set serveroutput on

It is environment command used to activates DBMS Statemens.

```
SQL> declare
  2  v_sal number(7,2);
  3  v_comm number(7,2);
  4  net number(7,2);
  5  begin
  6  v_sal:=&salary;
  7  v_comm:=&comm;
  8  net:=v_sal+nvl(v_comm,0);
  9  dbms_output.put_line('the net sal is:'||net);
 10  end;
```

```
  1  declare
  2  v_sal number(7,2);
  3  v_comm number(7,2);
  4  net number(7,2);
  5  begin
  6  dbms_output.put_line('the net sal is:'||(&sal+nvl(&comm,0)));
  7* end;
```

```
 1  declare
 2  vempno number(4):=&empno;
 3  vename varchar2(20);
 4  vsal number(7,2);
 5  vcomm number(7,2);
 6  netsal number(7,2);
 7  begin
 8  select ename,sal,comm into vename,vsal,vcomm from emp
 9  where empno=vempno;
10  netsal:=vsal+nvl(vcomm,0);
11  dbms_output.put_line('ename'||' '||'sal'||' '||'comm'||' '||'netsal');
12  dbms_output.put_line(rpad(vename,7)||' '||rpad(vsal,7)||' '||rpad(vcomm,7)||' '||n
13* end;
```

Nested Block:
PL/SQL block can be nested the block which is declarew in another
Block called as nested block or inner block or child block.

```
Declare
Begin
Declare
Begin
end;
end;
```

note: variable forward Reference is possible the backword reference may not possible.

```
1  declare
  2  m number:=100;
  3  begin
  4  m:=500;
  5  declare
  6  n number:=400;
  7  total number;
  8  begin
  9  m:=600;
 10  total:=m+n;
 11  dbms_output.put_line('the sum of m,n is:'||total);
 12  end;    --end the inner block
 13  dbms_output.put_line('the m value is:'||m);
 14* end;
```

Variable Attributes:
There are Two types of variable attributes.

By using this variable attributes we can Make the Datatype,size
independentely for a variable..

Column Type Attribute:
Syntax:
<variable name> <table name>.<column name>%type;
Percentile type(%):- used to declare column type variables.
eg: vename emp.ename%type;

```
declare
  2  vname emp.ename%type;
  3  begin
  4  select ename into vname from emp
  5  where empno=&eno;
  6  dbms_output.put_line('the ename:'||vname);
  7  end;
```

```
declare
  2  vname emp.ename%type;
  3  vdeptno emp.deptno%type;
  4  begin
  5  select ename,deptno into vname,vdeptno from emp
  6  where empno=&eno;
  7  dbms_output.put_line('the ename,deptno:'||vname||vdeptno);
  8* end;
```

16. Demonstration of database connectivity.

Step 1: Setup Project and Connect to Database

In general, it is not necessary to create a project to use the database tools of Workshop. The IDE can connect with any existing database that has a JDBC driver. From Workshop, you can simply create a database connection and proceed to use the database tools.

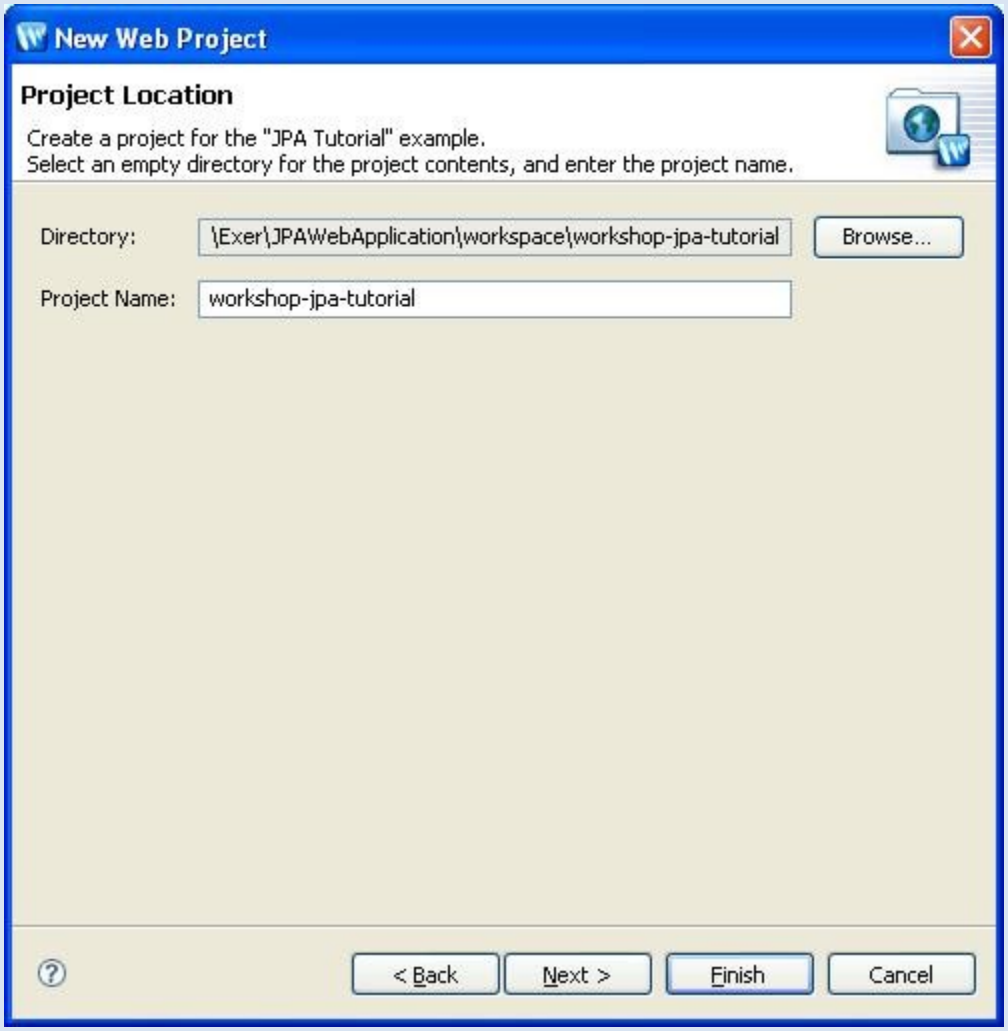
For this tutorial, we will open an existing demo application which contains a database, so that we can demonstrate the database tools.

Create a new project and load the sample application

In order to access the demo database, we must first create a project and install the files.

1. Choose **File > New > Example**.
2. In the **New Example** dialog, select **JPA > Workshop JPA Tutorial** from the list.
3. Click **Next**.

4. Verify that the value **workshop-jpa-tutorial** is in the project name field, click **Finish**.



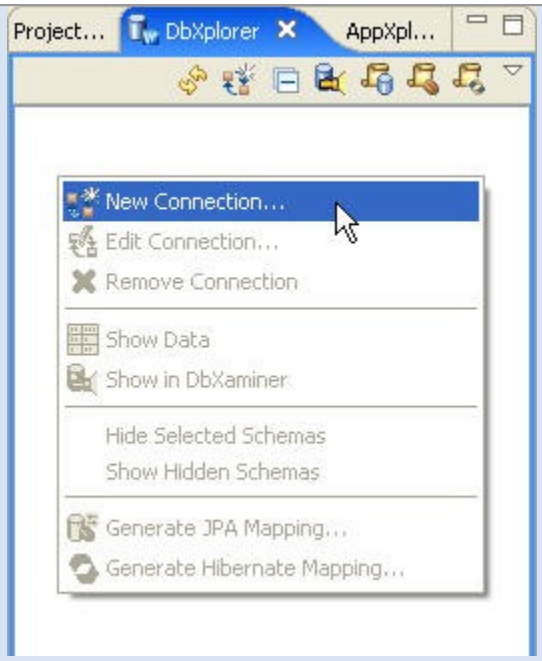
Connect and analyze database schema using DbXplorer

In this step, we will learn how to explore databases using the DbXplorer™, a view that provides an intuitive interface for database access through the ORM Workbench. Using the DbXplorer, you can setup a database connection, add and edit data, review the database artifacts, query the data in an existing table or column, and generate object relational mappings.

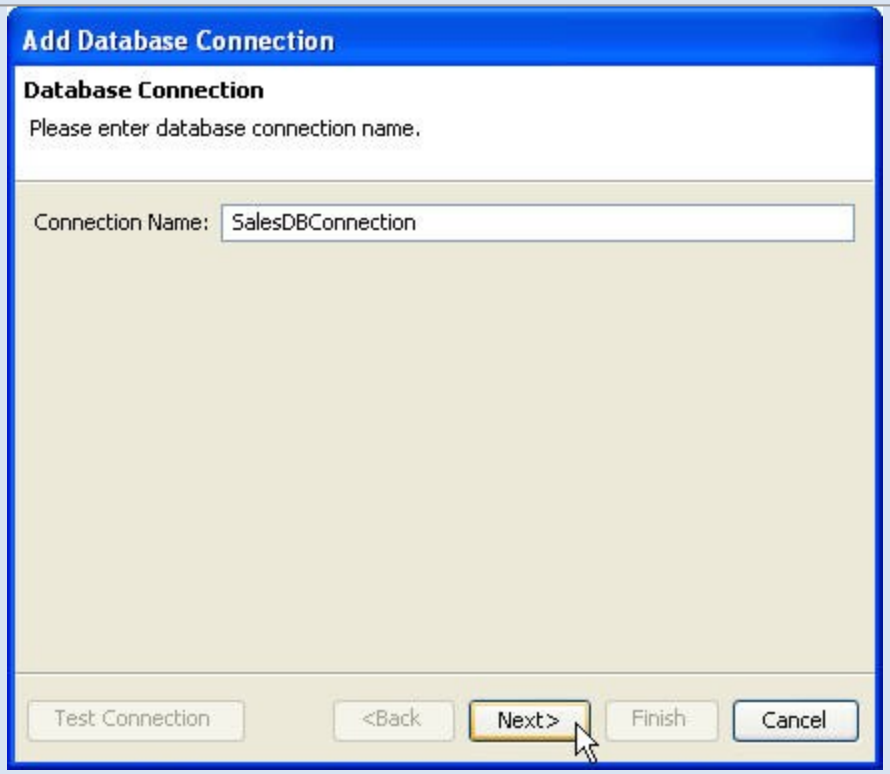
Create a New Database Connection

1. Click on the **DbXplorer** view tab, if it is visible. If not, open the **DbXplorer** view by clicking **Window** > **Show View > DbXplorer**.

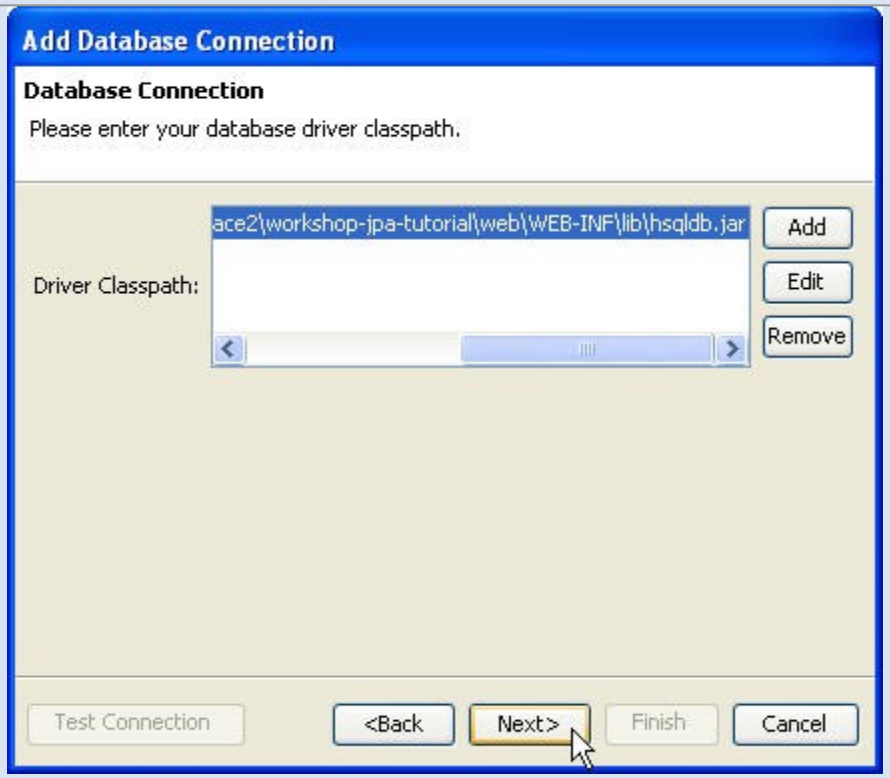
2. Right-click anywhere within the **DbXplorer** view and select **New Connection**.



3. In the **Add Database Connection** wizard, enter a database connection name. The database connection name can be arbitrary and does not have to match the actual name of the database server. Click **Next** to proceed.



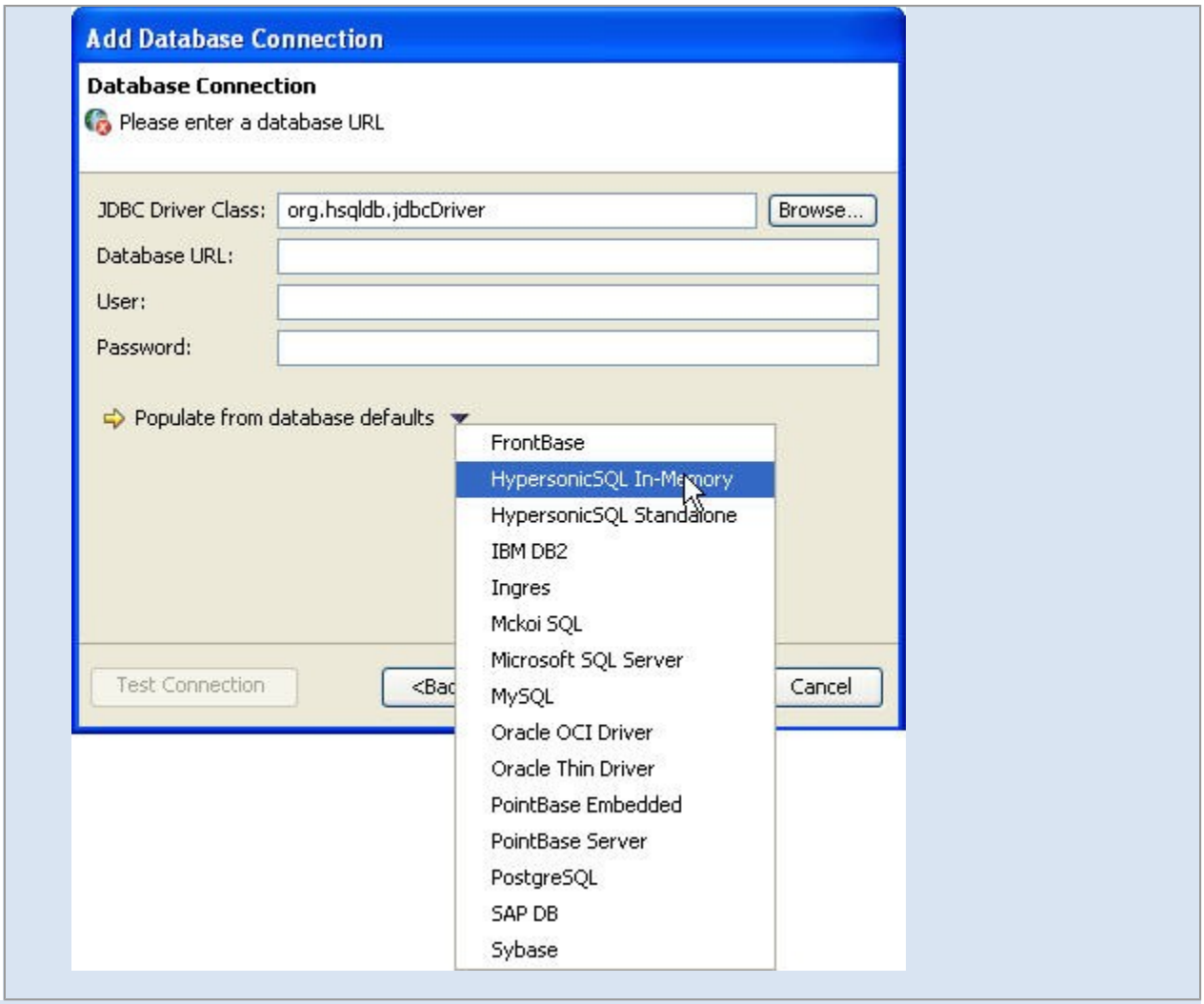
4. In the **Add Database Connection** dialog, click **Add** and select the Hypersonic JDBC driver file, **<path to workspace>\workshop-jpa-tutorial\web\WEB-INF\lib\hsqldb.jar**.



5. Click **Next**.

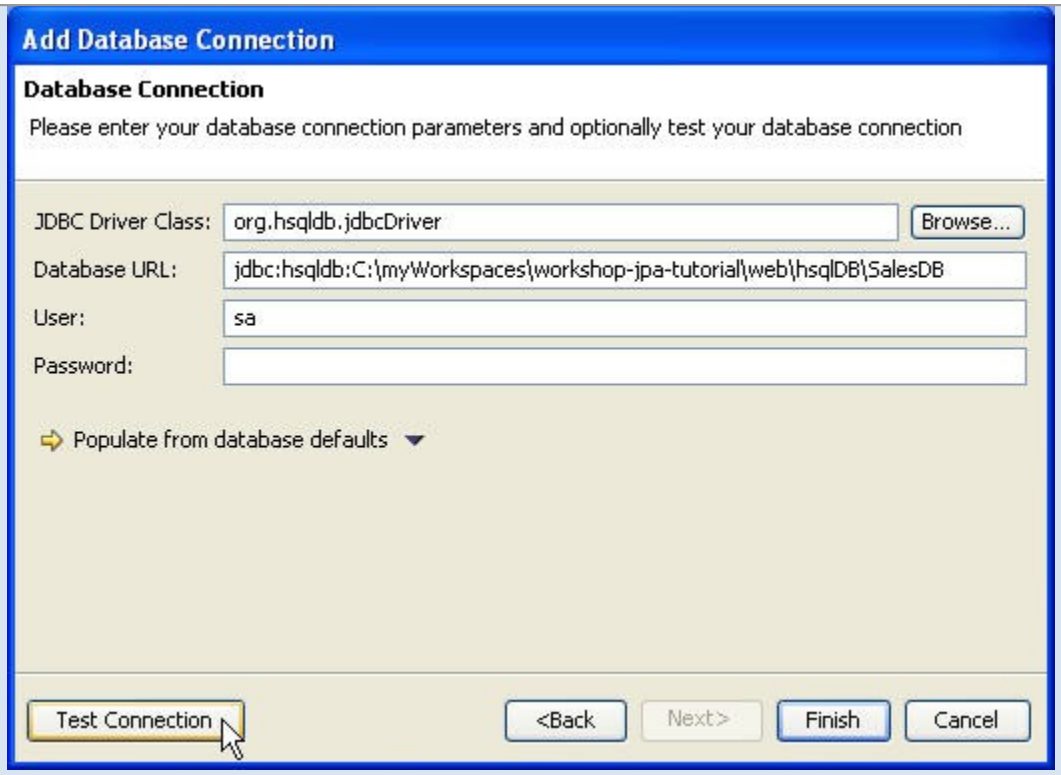
6. In the **JDBC Driver Class** field click **Browse** and select **org.hsqldb.jdbcDriver**.

7. Workshop provides sample Database URL's for some standard databases, which can be accessed from the **Populate from database defaults** pull down menu.
Select **HypersonicSQL In-Memory**.

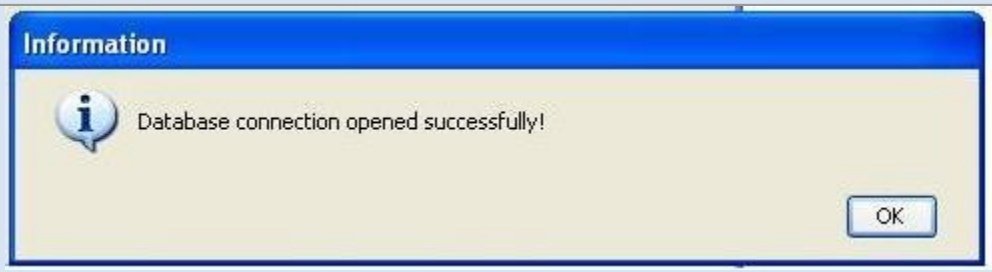


8. For database URL **jdbc:hsqldb:{db filename}**, specify the Hypersonic database script file location for {db filename}: <path to workspace>\workshop-jpa-tutorial\web\hsqlDB\SalesDB .

9. For User, enter **sa**.



10. Click the **Test Connection** button to verify the connection information.



11. Click **Finish**. The new database connection displays in the **DbXplorer** view.