EXPERIMENT-1

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

Oracle Built in Functions

There are two types of functions in Oracle:

1) Single Row Functions: Single row or Scalar functions return a value for every row that is processed in a query.

2) Group Functions: These functions group the rows of data based on the values returned by the query. This is discussed in SQL GROUP Functions. The group functions are used to calculate aggregate values like total or average, which return just one total or one average value after processing a group of rows.

There are four types of single row functions. They are:

1) Numeric Functions: These are functions that accept numeric input and return numeric values.
2) Character or Text Functions: These are functions that accept character input and can return both character and number values.

3) Date Functions: These are functions that take values that are of datatype DATE as input and return values of datatype DATE, except for the MONTHS_BETWEEN function, which returns a number.
4) Conversion Functions: These are functions that help us to convert a value in one form to another form. For Example: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE etc.

You can combine more than one function together in an expression. This is known as nesting of functions.

What is a DUAL Table in Oracle?

This is a single row and single column dummy table provided by oracle. This is used to perform mathematical calculations without using a table.

Select * from DUAL

Output:

```
DUMMY
-----
X
```

Select 777 * 888 from Dual

Output:

```
777 * 888
------
689976
```

1) Numeric Functions:

Numeric functions are used to perform operations on numbers. They accept numeric values as input and return numeric values as output. Few of the Numeric functions are:

Function Name	Return Value
ABS (x)	Absolute value of the number '*x*'
CEIL (x)	Integer value that is Greater than or equal to the number '*x*'

FLOOR (x)	Integer value that is Less than or equal to the number '*x*'
TRUNC (x, y)	Truncates value of number '*x*' up to '*y*' decimal places
ROUND (x, y)	Rounded off value of the number '*x*' up to the number '*y*' decimal places

The following examples explains the usage of the above numeric functions

Function Name	**Examples**	**Return Value**
ABS (x)	ABS (1) ABS (-1)	1 -1
CEIL (x)	CEIL (2.83) CEIL (2.49) CEIL (-1.6)	3 3 -1
FLOOR (x)	FLOOR (2.83) FLOOR (2.49) FLOOR (-1.6)	2 2 -2
TRUNC (x, y)	ROUND (125.456, 1) ROUND (125.456, 0) ROUND (124.456, -1)	125.4 125 120
ROUND (x, y)	TRUNC (140.234, 2) TRUNC (-54, 1) TRUNC (5.7) TRUNC (142, -1)	140.23 54 5 140

These functions can be used on database columns.

For Example: Let's consider the product table used in sql joins. We can use ROUND to round off the unit_price to the nearest integer, if any product has prices in fraction.

SELECT ROUND (unit_price) FROM product;

2) Character or Text Functions:

Character or text functions are used to manipulate text strings. They accept strings or characters as input and can return both character and number values as output.

Few of the character or text functions are as given below:

Function Name	**Return Value**
LOWER (string_value)	All the letters in ***'string_value'*** is converted to lowercase.
UPPER (string_value)	All the letters in ***'string_value'*** is converted to uppercase.
INITCAP (string_value)	All the letters in ***'string_value'*** is converted to mixed case.
LTRIM (string_value, trim_text)	All occurrences of ***'trim_text'*** is removed from the left of ***'string_value'***.
RTRIM (string_value, trim_text)	All occurrences of ***'trim_text'*** is removed from the right of ***'string_value'*** .
TRIM (trim_text FROM string_value)	All occurrences of ***'trim_text'*** from the left and right of ***'string_value'*** , ***'trim_text'*** can also be only one character long .
SUBSTR (string_value, m, n)	Returns ***'n'*** number of characters from ***'string_value'*** starting from the ***'m'*** position.
LENGTH (string_value)	Number of characters in *'string_value'* in returned.
LPAD (string_value, n, pad_value)	Returns *'string_value'* left-padded with *'pad_value'* . The length of the whole string will be of *'n'* characters.
RPAD (string_value, n, pad_value)	Returns *'string_value'* right-padded with *'pad_value'* . The

	length of the whole string will be of *'n'* characters.

For Example, we can use the above UPPER() text function with the column value as follows.

SELECT UPPER (product_name) FROM product;

The following examples explains the usage of the above character or text functions

Function Name	Examples	Return Value
LOWER(string_value)	LOWER('Good Morning')	good morning
UPPER(string_value)	UPPER('Good Morning')	GOOD MORNING
INITCAP(string_value)	INITCAP('GOOD MORNING')	Good Morning
LTRIM(string_value, trim_text)	LTRIM ('Good Morning', 'Good)	Morning
RTRIM (string_value, trim_text)	RTRIM ('Good Morning', ' Morning')	Good
TRIM (trim_text FROM string_value)	TRIM ('o' FROM 'Good Morning')	Gd Mrning
SUBSTR (string_value, m, n)	SUBSTR ('Good Morning', 6, 7)	Morning
LENGTH (string_value)	LENGTH ('Good Morning')	12
LPAD (string_value, n, pad_value)	LPAD ('Good', 6, '*')	**Good
RPAD (string_value, n, pad_value)	RPAD ('Good', 6, '*')	Good**

3) Date Functions:

These are functions that take values that are of datatype DATE as input and return values of datatypes DATE, except for the MONTHS_BETWEEN function, which returns a number as output.

Few date functions are as given below.

Function Name	Return Value
ADD_MONTHS (date, n)	Returns a date value after adding *'n'* months to the date *'x'*.
MONTHS_BETWEEN (x1, x2)	Returns the number of months between dates x1 and x2.
ROUND (x, date_format)	Returns the date *'x'* rounded off to the nearest century, year, month, date, hour, minute, or second as specified by the *'date_format'*.
TRUNC (x, date_format)	Returns the date *'x'* lesser than or equal to the nearest century, year, month, date, hour, minute, or second as specified by the 'date_format'.
NEXT_DAY (x, week_day)	Returns the next date of the *'week_day'* on or after the date *'x'* occurs.
LAST_DAY (x)	It is used to determine the number of days remaining in a month from the date *'x'* specified.
SYSDATE	Returns the systems current date and time.
NEW_TIME (x, zone1, zone2)	Returns the date and time in zone2 if date 'x' represents the time in zone1.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
ADD_MONTHS ( )	ADD_MONTHS ('16-Sep-81', 3)	16-Dec-81
MONTHS_BETWEEN( )	MONTHS_BETWEEN ('16-Sep-81', '16-Dec-81')	3
NEXT_DAY( )	NEXT_DAY ('01-Jun-08', 'Wednesday')	04-JUN-08

LAST_DAY( )	LAST_DAY ('01-Jun-08')	30-Jun-08
NEW_TIME( )	NEW_TIME ('01-Jun-08', 'IST', 'EST')	31-May-08

4) Conversion Functions:

These are functions that help us to convert a value in one form to another form. For Ex: a null value into an actual value, or a value from one datatype to another datatype like NVL, TO_CHAR, TO_NUMBER, TO_DATE.

Few of the conversion functions available in oracle are:

Function Name	Return Value
TO_CHAR (x [,y])	Converts Numeric and Date values to a character string value. It cannot be used for calculations since it is a string value.
TO_DATE (x [, date_format])	Converts a valid Numeric and Character values to a Date value. Date is formatted to the format specified by *'date_format'*.
NVL (x, y)	If *'x'* is NULL, replace it with *'y'*. *'x'* and *'y'* must be of the same datatype.
DECODE (a, b, c, d, e, default_value)	Checks the value of *'a'*, if *a = b*, then returns *'c'*. If *a = d*, then returns *'e'*. Else, returns *default_value*.

The below table provides the examples for the above functions

Function Name	Examples	Return Value
TO_CHAR ()	TO_CHAR (3000, '$9999') TO_CHAR (SYSDATE, 'Day, Month YYYY')	$3000 Monday, June 2008
TO_DATE ()	TO_DATE ('01-Jun-08')	01-Jun-08
NVL ()	NVL (null, 1)	1

EXPERIMENT-2

2. Queries using operators in SQL

SQL Operators Overview

An operator manipulates individual data items and returns a result. The data items are called operands or arguments. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL. There are two general classes of operators: unary and binary. Oracle Database Lite SQL also supports set operators.

Unary Operators

A unary operator uses only one operand. A unary operator typically appears with its operand in the following format.

operator operand

Binary Operators

A binary operator uses two operands. A binary operator appears with its operands in the following format.

operand1 operator operand2

Set Operators

Set operators combine sets of rows returned by queries, instead of individual data items. All set operators have equal precedence. Oracle Database Lite supports the following set operators.

- UNION
- UNION ALL
- INTERSECT
- MINUS

The levels of precedence among the Oracle Database Lite SQL operators from high to low are listed in Table. Operators listed on the same line have the same level of precedence.

Table 2-1 Levels of Precedence of the Oracle Database Lite SQL Operators

Precedence Level	SQL Operator
1	Unary + - arithmetic operators, PRIOR operator
2	* / arithmetic operators
3	Binary + - arithmetic operators, || character operators
4	All comparison operators
5	NOT logical operator
6	AND logical operator
7	OR logical operator

Other Operators:

Other operators with special formats accept more than two operands. If an operator receives a null operator, the result is always null. The only operator that does not follow this rule is CONCAT.

Arithmetic Operators

Arithmetic operators manipulate numeric operands. The '-' operator is also used in date arithmetic. Supported arithmetic operators are listed in Table.

Table Arithmetic Operators

Operator	Description	Example
+ (unary)	Makes operand positive	SELECT +3 FROM DUAL;

- (unary)	Negates operand	SELECT -4 FROM DUAL;
/	Division (numbers and dates)	SELECT SAL / 10 FROM EMP;
*	Multiplication	SELECT SAL * 5 FROM EMP;
+	Addition (numbers and dates)	SELECT SAL + 200 FROM EMP;
-	Subtraction (numbers and dates)	SELECT SAL - 100 FROM EMP;

Character Operators

Character operators used in expressions to manipulate character strings are listed in Table 2-3.

Table Character Operators

Operator	Description	Example
||	Concatenates character strings	SELECT 'The Name of the employee is: ' || ENAME FROM EMP;

Concatenating Character Strings

With Oracle Database Lite, you can concatenate character strings with the following results.

- Concatenating two character strings results in another character string.
- Oracle Database Lite preserves trailing blanks in character strings by concatenation, regardless of the strings' datatypes.
- Oracle Database Lite provides the CONCAT character function as an alternative to the vertical bar operator. For example,

  SELECT CONCAT (CONCAT (ENAME, ' is a '),job) FROM EMP WHERE SAL > 2000;
  This returns the following output.

  ```
  CONCAT(CONCAT(ENAME
  ----------------
  KING      is a PRESIDENT
  BLAKE     is a MANAGER
  CLARK     is a MANAGER
  JONES     is a MANAGER
  FORD      is a ANALYST
  SCOTT     is a ANALYST

  6 rows selected.
  ```
- Oracle Database Lite treats zero-length character strings as nulls. When you concatenate a zero-length character string with another operand the result is always the other operand. A null value can only result from the concatenation of two null strings.

Comparison Operators

Comparison operators used in conditions that compare one expression with another are listed in Table. The result of a comparison can be TRUE, FALSE, or UNKNOWN.

Table Comparison Operators

Operator	Description	Example
=	Equality test.	SELECT ENAME "Employee" FROM EMP WHERE SAL = 1500;

!=, ^=, <>	Inequality test.	SELECT ENAME FROM EMP WHERE SAL ^= 5000;
>	Greater than test.	SELECT ENAME "Employee", JOB "Title" FROM EMP WHERE SAL > 3000;
<	Less than test.	SELECT * FROM PRICE WHERE MINPRICE < 30;
>=	Greater than or equal to test.	SELECT * FROM PRICE WHERE MINPRICE >= 20;
<=	Less than or equal to test.	SELECT ENAME FROM EMP WHERE SAL <= 1500;
IN	"Equivalent to any member of" test. Equivalent to "=ANY".	SELECT * FROM EMP WHERE ENAME IN ('SMITH', 'WARD');
ANY/ SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <= or >=. Evaluates to FASLE if the query returns no rows.	SELECT * FROM DEPT WHERE LOC = SOME ('NEW YORK','DALLAS');
NOT IN	Equivalent to "!=ANY". Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM DEPT WHERE LOC NOT IN ('NEW YORK', 'DALLAS');
ALL	Compares a value with every value in a list or returned by a query. Must be preceded by =, !=, >, <, <= or >=. Evaluates to TRUE if the query returns no rows.	SELECT * FROM emp WHERE sal >= ALL (1400, 3000);
[NOT] BETWEEN x and y	[Not] greater than or equal to x and less than or equal to y.	SELECT ENAME, JOB FROM EMP WHERE SAL BETWEEN 3000 AND 5000;
EXISTS	TRUE if a sub-query returns at least one row.	SELECT * FROM EMP WHERE EXISTS (SELECT ENAME FROM EMP WHERE MGR IS NULL);
x [NOT] LIKE y [ESCAPE z]	TRUE if x does [not] match the pattern y. Within y, the character "%" matches any string of zero or more characters except null. The character "_" matches any single character. Any character following ESCAPE is interpreted literally, useful when y contains a percent (%) or underscore (_).	SELECT * FROM EMP WHERE ENAME LIKE '%E%';
IS [NOT] NULL	Tests for nulls. This is the only operator that should be used to test for nulls.	SELECT * FROM EMP WHERE COMM IS NOT

NULL AND SAL > 1500;

Logical Operators

Logical operators which manipulate the results of conditions are listed in Table

Table Logical Operators

Operator	Description	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	SELECT * FROM EMP WHERE NOT (job IS NULL) SELECT * FROM EMP WHERE NOT (sal BETWEEN 1000 AND 2000)
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise returns UNKNOWN.	SELECT * FROM EMP WHERE job='CLERK' AND deptno=10
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise, returns UNKNOWN.	SELECT * FROM emp WHERE job='CLERK' OR deptno=10

Set Operators

Set operators which combine the results of two queries into a single result are listed in Table.

Table Set Operators

Operator	Description	Example
UNION	Returns all distinct rows selected by either query.	SELECT * FROM (SELECT ENAME FROM EMP WHERE JOB = 'CLERK' UNION SELECT ENAME FROM EMP WHERE JOB = 'ANALYST');
UNION ALL	Returns all rows selected by either query, including all duplicates.	SELECT * FROM (SELECT SAL FROM EMP WHERE JOB = 'CLERK' UNION SELECT SAL FROM EMP WHERE JOB = 'ANALYST');
INTERSECT and INTERSECT ALL	Returns all distinct rows selected by both queries.	SELECT * FROM orders_list1

		INTERSECT SELECT * FROM orders_list2
MINUS	Returns all distinct rows selected by the first query but not the second.	SELECT * FROM (SELECT SAL FROM EMP WHERE JOB = 'PRESIDENT' MINUS SELECT SAL FROM EMP WHERE JOB = 'MANAGER');

Note: : The syntax for INTERSECT ALL is supported, but it returns the same results as INTERSECT.

Other Operators

Other operators used by Oracle Database Lite are listed in Table.

Table Other Operators

Operator	Description	Example
(+)	Indicates that the preceding column is the outer join column in a join.	SELECT ENAME, DNAME FROM EMP, DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO (+);
PRIOR	Evaluates the following expression for the parent row of the current row in a hierarchical, or tree-structured query. In such a query, you must use this operator in the CONNECT BY clause to define the relationship between the parent and child rows.	SELECT EMPNO, ENAME, MGR FROM EMP CONNECT BY PRIOR EMPNO = MGR;

EXPERIMENT-3

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

Inserting data

The INSERT statement is used to insert data into tables. We will create a new table, where we will do our examples.

```
sql> CREATE TABLE Books(Id INTEGER PRIMARY KEY, Title VARCHAR(100),
-> Author VARCHAR(60));
```

We create a new table Books, with Id, Title and Author columns.

```
sql> INSERT INTO Books(Id, Title, Author) VALUES(1, 'War and Peace',
-> 'Leo Tolstoy');
```

This is the classic INSERT SQL statement. We have specified all column names after the table name and all values after the VALUES keyword. We add our first row into the table.

```
sql> SELECT * FROM Books;
+----+--------------+-------------+
| Id | Title        | Author      |
+----+--------------+-------------+
|  1 | War and Peace | Leo Tolstoy |
+----+--------------+-------------+
```

We have inserted our first row into the Books table.

```
sql> INSERT INTO Books(Title, Author) VALUES ('The Brothers Karamazov',
-> 'Fyodor Dostoyevsky');
```

We add a new title into the Books table. We have omitted the Id column. The Id column has AUTO_INCREMENT attribute. This means that MySQL will increase the Id column automatically. The value by which the AUTO_INCREMENT column is increased is controlled by auto_increment system variable. By default it is 1.

```
mysql> SELECT * FROM Books;
+----+-----------------------+--------------------+
| Id | Title                 | Author             |
+----+-----------------------+--------------------+
|  1 | War and Peace         | Leo Tolstoy        |
|  2 | The Brothers Karamazov | Fyodor Dostoyevsky |
+----+-----------------------+--------------------+
```

Here is what we have in the Books table.

```
sql> INSERT INTO Books VALUES(3, 'Crime and Punishment',->'Fyodor Dostoyevsky');
```

In this SQL statement, we did not specify any column names after the table name. In such a case, we have to supply all values.

```
sql> REPLACE INTO Books VALUES(3, 'Paradise Lost', 'John Milton');
Query OK, 2 rows affected (0.00 sec)
```

The REPLACE statement is a MySQL extension to the SQL standard. It inserts a new row or replaces the old row if it collides with an existing row. In our table, there is a row with Id=3. So our previous statement replaces it with a new row. There is a message that two rows were affected. One row was deleted and one was inserted.

```
sql> SELECT * FROM Books WHERE Id=3;
+----+--------------+-------------+
| Id | Title        | Author      |
```

```
+----+--------------+------------+
|  3 | Paradise Lost | John Milton |
+----+--------------+------------+
```

This is what we have now in the third column.

We can use the INSERT and SELECT statements together in one statement.

```
sql> CREATE TABLE Books2(Id INTEGER PRIMARY KEY AUTO_INCREMENT,
  -> Title VARCHAR(100), Author VARCHAR(60)) type=MEMORY;
```

First, we create a temporary table called Books2 in memory.

```
sql> INSERT INTO Books2 SELECT * FROM Books;
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Here we insert all data into the Books2 that we select from the Books table.

```
sql> SELECT * FROM Books2;
+----+-----------------------+--------------------+
| Id | Title                 | Author             |
+----+-----------------------+--------------------+
|  1 | War and Peace         | Leo Tolstoy        |
| 2 | The Brothers Karamazov | Fyodor Dostoyevsky |
|  3 | Paradise Lost         | John Milton        |
+----+-----------------------+--------------------+
```

We verify it. All OK.

Deleting data

In MySQL, we can delete data using the DELETE and TRUNCATE statements. The TRUNCATE statement is a MySQL extension to the SQL specification. First, we are going to delete one row from a table. We will use the Books2 table that we have created previously.

```
sql> DELETE FROM Books2 WHERE Id=1;
```

We delete a row with Id=1.

```
sql> SELECT * FROM Books2;
+----+-----------------------+--------------------+
| Id | Title                 | Author             |
+----+-----------------------+--------------------+
| 2 | The Brothers Karamazov | Fyodor Dostoyevsky |
|  3 | Paradise Lost         | John Milton        |
+----+-----------------------+--------------------+
```

We verify the data.

```
sql> DELETE FROM Books2;
sql> TRUNCATE Books2;
```

These two SQL statements delete all data in the table.

Updating data

The UPDATE statement is used to change the value of columns in selected rows of a table.

```
sql> SELECT * FROM Books;
+----+---------------------------+--------------------+
| Id | Title                     | Author             |
+----+---------------------------+--------------------+
|  1 | War and Peace             | Leo Tolstoy        |
```

```
|  2 | The Brothers Karamazov      | Fyodor Dostoyevsky |
|  3 | Paradise Lost               | John Milton        |
| 4 | The Insulted and Humiliated | Fyodor Dostoyevsky |
|  5 | Cousin Bette                | Honore de Balzac   |
+----+-----------------------------+--------------------+
```

We recreate the table Books. These are the rows.

Say we wanted to change 'Leo Tolstoy' to 'Lev Nikolayevich Tolstoy' table. The following statement shows, how to accomplish this.

```
sql> UPDATE Books SET Author='Lev Nikolayevich Tolstoy' -> WHERE Id=1;
```

The SQL statement sets the author column to 'Lev Nikolayevich Tolstoy' for the column with Id=1.

```
sql> SELECT * FROM Books WHERE Id=1;
+----+--------------+-------------------------+
| Id | Title        | Author                  |
+----+--------------+-------------------------+
|  1 | War and Peace | Lev Nikolayevich Tolstoy |
+----+--------------+-------------------------+
```

The row is correctly updated.

EXPERIMENT-4

4. Queries using Group By, Order By, and Having Clauses

The SQL HAVING syntax

The general syntax is:

```
1. SELECT column-names
2.  FROM table-name
3. WHERE condition
4. GROUP BY column-names
5. HAVING condition
```

The general syntax with ORDER BY is:

```
1. SELECT column-names
2.  FROM table-name
3. WHERE condition
4. GROUP BY column-names
5. HAVING condition
6. ORDER BY column-names
```

SQL GROUP BY Examples

Problem: List the number of customers in each country. Only include countries with more than 10 customers.

```
1. SELECT COUNT(Id), Country
2.  FROM Customer
3. GROUP BY Country
4. HAVING COUNT(Id) > 10
```

Results: 3 records

Count	Country
11	France
11	Germany
13	USA

Problem: List the number of customers in each country, except the USA, sorted high to low. Only include countries with 9 or more customers.

```
1. SELECT COUNT(Id), Country
2.  FROM Customer
3. WHERE Country <> 'USA'
4. GROUP BY Country
5. HAVING COUNT(Id) >= 9
6. ORDER BY COUNT(Id) DESC
```

Results: 3 records

Count	Country
11	France
11	Germany
9	Brazil

Problem: List all customer with average orders between $1000 and $1200.

```
SELECT AVG(TotalAmount), FirstName, LastName
1.  FROM [Order] O JOIN Customer C ON O.CustomerId = C.Id
2. GROUP BY FirstName, LastName
3. HAVING AVG(TotalAmount) BETWEEN 1000 AND 1200
```

Results: 10 records

Average	FirstName	LastName

1081.215000	Miguel	Angel Paolino
1063.420000	Isabel	de Castro
1008.440000	Alexander	Feuer
1062.038461	Thomas	Hardy
1107.806666	Pirkko	Koskitalo
1174.945454	Janete	Limeira
1073.621428	Antonio	Moreno
1065.385000	Rita	Müller
1183.010000	José	Pedro Freyre
1057.386666	Carine	Schmitt

EXPERIMENT-

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

SQL Transaction Control (TC) commands are used to manage database transaction. SQL transaction command use with DML statement for INSERT, UPDATE and DELETE.

DML statement are store into SQL buffer until you execute Transaction commands. Once you execute transaction commands its store permanent to a database.

SQL COMMIT

SQL COMMIT command save new changes store into database.

Syntax: COMMIT;

Example

```
SQL> COMMIT;
Commit complete.
```

SQL SAVEPOINT

SQL SAVEPOINT command create new save point. SAVEPOINT command save the current point with the unique name in the processing of a transaction.

Syntax

```
SAVEPOINT savepoint_name;
```

Example

```
SQL> CREATE TABLE emp_data (
   no NUMBER(3),
   name VARCHAR(50),
   code VARCHAR(12)     );
Table created.
SQL> SAVEPOINT table_create;
Savepoint created.
SQL> insert into emp_data VALUES(1,'Opal', 'e1401');
1 row created.
SQL> SAVEPOINT insert_1;
Savepoint created.
SQL> insert into emp_data VALUES(2,'Becca', 'e1402');
1 row created.
SQL> SAVEPOINT insert_2;
Savepoint created.
SQL> SELECT * FROM emp_data;
      NO NAME                           CODE
---------- ---------------------------------- --------
       1 Opal                           e1401
       2 Becca                          e1402
```

SQL ROLLBACK

SQL ROLLBACK command execute at the end of current transaction and undo/undone any changes made since the begin transaction.

Syntax

```
ROLLBACK [To SAVEPOINT_NAME];
```

Example: Above example we are create 3 SAVEPOINT table_create, insert_1 and insert_2. Now we are rollback to insert_1 SAVEPOINT.

```
SQL> ROLLBACK TO insert_1;
Rollback complete.
SQL> SELECT * FROM emp_data;
      NO NAME                           CODE
---------- ---------------------------------- --------
       1 Opal                           e1401
```

EXPERIMENT-6

6. Queries to Build Report in SQL *PLUS

Generating Reports from SQL*Plus:-

Creating Reports using Command-line SQL*Plus:

In addition to plain text output, the SQL*Plus command-line interface enables we to generate either a complete web page, HTML output which can be embedded in a web page, or data in CSV format. We 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. We 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 we 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.

We 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. We can specify the delimiter character by using the DELIMITER option. We can also output text without quotes by using QUOTE OFF. Creating HTML Reports

During a SQL*Plus session, use the SET MARKUP command interactively to write HTML to a spool file. We 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. We 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 we to easily identify the type of our output files, and also enables web browsers to identify and correctly display our HTML files. If no extension is specified, the default SQL*Plus file extension is used.

We use SPOOL OFF or EXIT to append final HTML tags to the spool file and then close it. If we 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.

We 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 we

invoke a script, we 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 our 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.

EXPERIMENT-7

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

WHAT IS THE ALTER COMMAND?

The alter command is used to modify an existing database, table, view or other database objects that might need to change during the life cycle of a database.

Alter- syntax

The basic syntax used to add a column to an already existing table is shown below

ALTER TABLE `table_name` ADD COLUMN `column_name` `data_type`;

HERE

- **"ALTER TABLE `table_name`"** is the command that tells MySQL server to modify the table named `table_name`.
- **"ADD COLUMN `column_name` `data_type`"** is the command that tells MySQL server to add a new column named `column_name` with data type `data_type'.
- We can use the script shown below to add a new field to the members table.
- ALTER TABLE `members` ADD COLUMN `credit_card_number` VARCHAR(25);

WHAT IS THE DROP COMMAND?

The DROP command is used to

1. Delete a database from MySQL server
2. Delete an object (like Table , Column)from a database.

Let's now look at practical examples that make use of the DROP command.

Suppose the online billing functionality will take some time and we want to DROP the credit card column

We can use the following script

ALTER TABLE `members` DROP COLUMN `credit_card_number`;

Executing the above script drops the column credit_card_number from the members table

DROP TABLE

The syntax to DROP a table from Database is as follow -

DROP TABLE `sample_table`;

Let'look at an example

DROP TABLE `categories_archive`;

Executing the above script deletes the table named ` categories_archive ` from our database.

WHAT IS THE RENAME COMMAND?

The rename command is used to **change the name of an existing database object(like Table,Column) to a new name**.

Renaming a table does not make it to lose any data is contained within it.

Syntax:-

The rename command has the following basic syntax.

RENAME TABLE `current_table_name` TO `new_table_name`;

Let's suppose that we want to rename the movierentals table to movie_rentals, we can use the script shown below to achieve that.

RENAME TABLE `movierentals` TO `movie_rentals`;

Executing the above script renames the table `movierentals` to `movie_rentals`.

We will now rename the movie_rentals table back to its original name.

RENAME TABLE `movie_rentals` TO `movierentals`;

CHANGE KEYWORD

Change Keywords allows you to

1. Change Name of Column
2. Change Column Data Type
3. Change Column Constraints

Let's look at an example. The full names field in the members table is of varchar data type and has a width of 150.

```
SHOW COLUMNS FROM `members`;
```

Suppose we want to

1. Change the field name from "full_names" to "fullname"
2. Change it to char data type with a width of 250
3. Add a NOT NULL constraint

We can accomplish this using the change command as follows -

```
ALTER TABLE `members` CHANGE COLUMN `full_names` `fullname` char(250) NOT NULL;
```

MODIFY KEYWORD

The MODIFY Keyword allows you to

1. Modify Column Data Type
2. Modify Column Constraints

In the CHANGE example above, we had to change the field name as well other details. **Omitting the field name from the CHANGE statement will generate an error.** Suppose we are only interested in changing the data type and constraints on the field without affecting the field name, we can use the MODIFY keyword to accomplish that.

The script below changes the width of "fullname" field from 250 to 50.

```
ALTER TABLE `members`MODIFY `fullname` char(50) NOT NULL;
```

AFTER KEYWORD

Suppose that we want to add a new column at a specific position in the table.

We can use the alter command together with the AFTER keyword.

The script below adds "date_of_registration" just after the date of birth in the members table.

```
ALTER TABLE `members` ADD `date_of_registration` date NULL AFTER `date_of_birth`;
```

EXPERIMENT-8

8. Queries on Joins and Correlated Sub-Queries

Noncorrelated and Correlated Subqueries

Subqueries can be categorized into two types:

- A *noncorrelated* (simple) subquery obtains its results independently of its containing (outer) statement.
- A *correlated* subquery requires values from its outer query in order to execute.

Noncorrelated Subqueries

A noncorrelated subquery executes independently of the outer query. The subquery executes first, and then passes its results to the outer query, For example:

=> SELECT name, street, city, state FROM addresses WHERE state IN **(SELECT state FROM states)**;

Vertica executes this query as follows:

1. Executes the subquery SELECT state FROM states (in bold).
2. Passes the subquery results to the outer query.

A query's WHERE and HAVING clauses can specify noncorrelated subqueries if the subquery resolves to a single row, as shown below:

In WHERE clause

=> SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);

In HAVING clause

=> SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a HAVING SubQ1.a = (SubQ1.a & (SELECT y from SubQ2)

Correlated Subqueries

A correlated subquery typically obtains values from its outer query before it executes. When the subquery returns, it passes its results to the outer query.

Note: You can use an outer join to obtain the same effect as a correlated subquery.

In the following example, the subquery needs values from the addresses.state column in the outer query:

```
=> SELECT name, street, city, state FROM addresses
   WHERE EXISTS (SELECT * FROM states WHERE states.state = addresses.state);
```

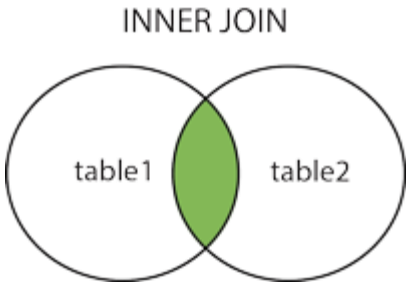
Vertica executes this query as follows:

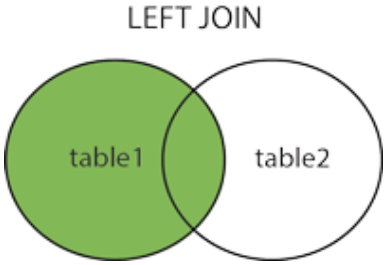
1. The subquery evaluates each addresses.state value in the outer block records.
2. It then passes its results to the outer query block.

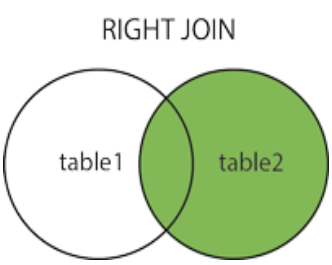
Different Types of SQL JOINs

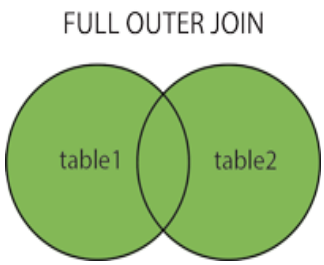
Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table







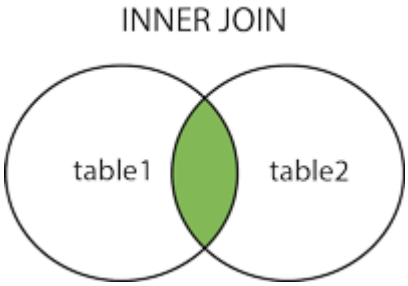


SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

INNER JOIN Syntax

SELECT *column_name(s)* FROM *table1* INNER JOIN *table2* ON *table1.column_name* = *table2.column_name*;



INNER JOIN:

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate FROM Orders INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

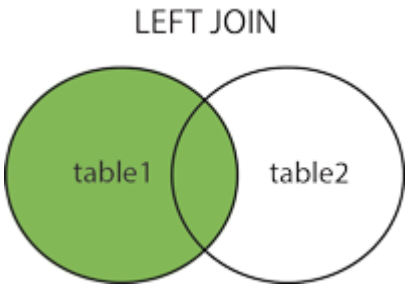
SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

LEFT JOIN Syntax

SELECT *column_name(s)* FROM *table1* LEFT JOIN *table2* ON *table1.column_name* = *table2.column_name*;

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.



Example

SELECT Customers.CustomerName, Orders.OrderID FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID ORDER BY Customers.CustomerName;

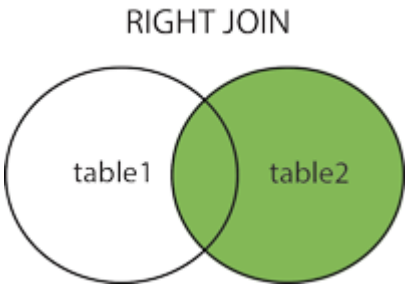
SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

RIGHT JOIN Syntax

SELECT *column_name(s)* FROM *table1* RIGHT JOIN *table2* ON *table1.column_name* = *table2.column_name*;

Note: In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



Example

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName FROM Orders RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID ORDER BY Orders.OrderID;

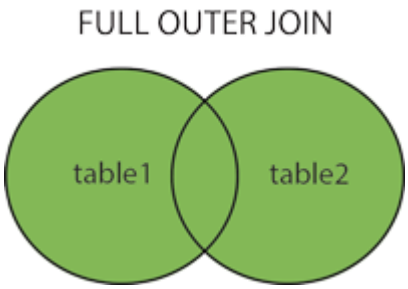
SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword return all records when there is a match in either left (table1) or right (table2) table records.

Note: FULL OUTER JOIN can potentially return very large result-sets!

FULL OUTER JOIN Syntax

SELECT *column_name(s)* FROM *table1* FULL OUTER JOIN *table2* ON *table1.column_name* = *table2.column_name*;



SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

SELECT Customers.CustomerName, Orders.OrderID FROM Customers FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID ORDER BY Customers.CustomerName;

SQL Self JOIN

A self JOIN is a regular join, but the table is joined with itself.

Self JOIN Syntax

SELECT *column_name(s)* FROM *table1 T1, table1 T2* WHERE *condition*;

Example

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City FROM Customers A, Customers B WHERE A.CustomerID <> B.CustomerID AND A.City = .City ORDER BY A.City;

EXPERIMENT-9

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

Index:

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Create an Index: The syntax for creating an index in Oracle/PLSQL is:

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (column1, column2, ... column_n)
  [ COMPUTE STATISTICS ];
```

UNIQUE:

It indicates that the combination of values in the indexed columns must be unique.

index_name: The name to assign to the index.

table_name: The name of the table in which to create the index.

column1, column2, ... column_n: The columns to use in the index.

COMPUTE STATISTICS: It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

Example

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name);
```

In this example, we've created an index on the supplier table called supplier_idx. It consists of only one field - the supplier_name field.

We could also create an index with more than one field as in the example below:

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name, city);
```

We could also choose to collect statistics upon creation of the index as follows:

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name, city)
  COMPUTE STATISTICS;
```

Sequence: A sequence is a special database object that generates integers according to specified rules at the time the sequence was created. Users select data from them using two special keywords to denote pseudo (or partial or virtual) columns in the database.

The pseudo columns in oracle data base are:

Pseudo Column	Meaning
Currval	Returns the current value of a sequence
Nextval	Returns the next value of a sequence
Rowid	Returns the rowid(address) of a row in the table it will be in format as block.row.file. It is used to quickly access the row and uniquely address them.
Rownum	Returns the number indicating in which order oracle select rows. First row selected will be rownum of 1 and second row rownum of 2 and so on.
User	Returns the current user name

- The first pseudo column is CURVAL which contains the current value that is generated by the sequence.
- The second pseudo column is NEXTVAL contains the next value that the sequence will generate according to the rules developed for it.

Selecting NEXTVAL on the sequence effectively eliminates whatever value is stored in CURVAL.
Note: Data may only drawn from sequence but never placed into it.

Create sequence <sequence name>
(INCREMENT BY <integer value> START WITH <integer value> MAXVALUE <integer value>/NOMAXVALUE MINVALUE <integer value>/NOMINVALUE CYCLE/NOCYCLE CACHE <integer value><integer value>/NOCACHE ORDER/NOORDER)

Start with – specify the starting value and after generated the value specified by start with the first time the sequence's NEXTVAL virtual column is referenced.
Increment by – specifies the number by which the sequence is incremented, every time the NEXTVAL column is referenced if the number is negative it decrements the sequence.
Minvalue – for specifying minimum value.
Maxvalue – for specifying maximum value.
Cycle – allows the sequence to recycle values produced when the maxvalue or minvalue is reached. If recycling is not desired the NOCYCLE keyword can be used.
Cache – allows the sequence to cache the specified number of values at any time in order to improve performance. If caching is not desired then the nocache is used.
Order – allows the sequence to assign sequence values in the order the requests are received by the sequence. If order is not desired then we can use NOORDER.

Example:

CREATE SEQUENCE count1 START WITH 20 INCREMENT BY -1 NOMAXVALUE CYCLE ORDER;
CREATE SEQUENCE rand MINVALUE 0 MAXVALUE 100 NOCYCLE;

Using sequences – once the sequence is created it is referenced using CURVAL and NEXTVAL pseudo columns.

Eg: select rand.CURVAL rand.NEXTVAL rand.CURVAL from dual;

Generally users do not select statements to draw data from sequences. While inserting or updating the value is directly incorporated into the field.

Eq: insert into expence(expno, amt) values(exp_seqno.NEXTVAL, 1234);

Sequence can be altered using alter command.

Synonym:

A Synonym is an alternative name for a table, view, sequence, procedure, stored function, package or another synonym.
Types of Synonyms:

1. Private – created for personal use.
2. Public – created for other people to use it.

Create synonym <synonym name> for <Actual Name> - creates private synonym.
Create public synonym <synonym name> for <Actial Name> - creates public synonym.

Controlling Access:

Transaction control commands: A transaction is one logical unit of work consisting of one or more logically related statements.
Commit: Ends the current transaction by saving database changes and starts a new transaction. All DML commands when executed will be modified in the buffer and the modification will be saved to database when we commit it explicitly.
Rollback: Ends the current transaction by discarding database changes and starts a new transaction.
Savepoint: Defines break points for the transaction to allow partial rollbacks.
Data Control Language

Privileges are the rights or permissions assigned to user to use some or all to Oracle's resources. Security can be provided using DCL commands.

The right or permission assigned to user(s) to use some or all of Oracle's resources on server, are known as privileges.

The granting of permission is known as granting of privileges. These are of two types:

- System Privileges
- Object Privileges

After creating the user we have to grant privileges.

GRANT:

Privileges can be granted by the grant command.

GRANT (object privileges) on <object name> to username [WITH GRANT OPTION];

WITH GRANT OPTION: allows the user to whom this privileges has been granted to in turn grant this privilege to other users.

Eg: grant all on scott.emp to ram;

Grant all on scott.emp to raju with grant option;

Grant select, update, delete on scott.emp to kumar;

Grant select on scott.emp to srinu; - This is given by raju to srinu as he has the privilege to grant given using with grant option.

Revoking Privilages

Privilages once granted can be revoked later if needed.

REVOKE (object privileges) on <object name> from <user name>;

Eg. REVOKE select on scott.emp from srinu;

REVOKE update, delete on scott.emp from kumar;

REVOKE all on scott.emp from ram;

There are 8 types of object privileges:

Object	**Description**
Select	Permits to access data in the table, sequence, view or snapshot.
Insert	Permits to insert data into a table or view.
Update	Permits to update data into a table or view.
Delete	Permits to delete data from a table or view.
Alter	Permits to alter the definition of a table or sequence only. The alter privilege on all other database objects are considered system privileges.
Index	Permits to create index on a table already defined.
References	Permits to create or alter a table in order to create a foreign key constraint against data in the referenced table.
Execute	Permits to run a stored procedure or function.

Locking Rows for Update:

Oracle Database provides data concurrency and consistency. The data a session is viewing or changing must not be changed by other sessions until the user is finished, and integrity. The data and structures must reflect all changes made to them in the correct sequence, among transactions through a locking mechanisms. The locks are performed automatically and requires no user action directly associated with a session. Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource. The resources can be either user objects, such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.

Oracle Database automatically obtains and manages necessary locks when executing SQL statements, so you need not be concerned with such details. However, the database also lets you lock data manually.

LOCK TABLE statement: The LOCK TABLE statement allows you to explicitly acquire a shared or exclusive table lock on the specified table. The table lock lasts until the end of the

current transaction. To lock a table, you must either be the database owner or the table owner. Explicitly locking a table is useful to:

- Avoid the overhead of multiple row locks on a table (in other words, user-initiated lock escalation)
- Avoid deadlocks

We cannot lock system tables with this statement.

Syntax:

LOCK TABLE tableName **IN { SHARE | EXCLUSIVE } MODE**

After a table is locked in either mode, a transaction does not acquire any subsequent row-level locks on a table. For example, if a transaction locks the entire Flights table in share mode in order to read data, a particular statement might need to lock a particular row in exclusive mode in order to update the row. However, the previous table-level lock on the Flights table forces the exclusive lock to be table-level as well. If the specified lock cannot be acquired because another connection already holds a lock on the table, a statement-level exception is raised (*SQLState* X0X02) after the deadlock timeout period.

Examples: To lock the entire Flights table in share mode to avoid a large number of row locks, use the following statement:

```
LOCK TABLE Flights IN SHARE MODE;
    SELECT * FROM Flights WHERE orig_airport > 'OOO';
```

We have a transaction with multiple UPDATE statements. Since each of the individual statements acquires only a few row-level locks, the transaction will not automatically upgrade the locks to a table-level lock. However, collectively the UPDATE statements acquire and release a large number of locks, which might result in deadlocks. For this type of transaction, you can acquire an exclusive table-level lock at the beginning of the transaction. For example:

```
LOCK TABLE FlightAvailability IN EXCLUSIVE MODE;
UPDATE FlightAvailability
SET economy_seats_taken = (economy_seats_taken + 2)
WHERE flight_id = 'AA1265' AND flight_date = DATE('2004-04-15');
.....
```

If a transaction needs to look at a table before updating the table, acquire an exclusive lock before selecting to avoid deadlocks. For example:

```
LOCK TABLE Maps IN EXCLUSIVE MODE;
SELECT MAX(map_id) + 1 FROM Maps;
-- INSERT INTO Maps . . .
```

Creating password and security feature:

Creating user in Oracle: First we have to login as administrator and then we have to create user.

```
conn sys/manager; or conn system/manger or conn /as sysdba
```

After connecting now we can create user as:

```
Create user <user name> identified by <password>;
```

Example:

```
create user abc identified by pqr;
```

EXPERIMENT-10

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

PL/SQL Variables

These are placeholders that store the values that can change through the PL/SQL Block.

General Syntax to declare a variable is

variable_name datatype [NOT NULL := value];

- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.
- *value* or DEFAULT *value*is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

DECLARE
salary number (6);

* "salary" is a variable of datatype number and of length 6.

When a variable is specified as NOT NULL, you must initialize the variable when it is declared

For example: The below example declares two variables, one of which is a not null.

DECLARE
salary number(4);
dept varchar2(10) NOT NULL := "HR Dept";

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

1) We can directly assign values to variables.
The General Syntax is:
variable_name:= value;

2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

SELECT column_name INTO variable_name FROM table_name [WHERE condition];

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE var_salary number(6);
var_emp_id number(6) = 1116;
BEGIN
 SELECT salary
 INTO var_salary
 FROM employee
 WHERE emp_id = var_emp_id;
 dbms_output.put_line(var_salary);
 dbms_output.put_line('The employee '
|| var_emp_id || ' has salary ' || var_salary);
END;
/
```

Scope of PS/SQL Variables

PL/SQL allows the nesting of Blocks within Blocks i.e., the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- *Local* variables - These are declared in a inner block and cannot be referenced by outside Blocks.
- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning thier product to the third variable created in the inner block. The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

```
DECLARE
var_num1 number;
var_num2 number;
BEGIN
  var_num1 := 100;
  var_num2 := 200;
  DECLARE
    var_mult number;
    BEGIN
            var_mult := var_num1 * var_num2;
     END;
END;
/
```

Anchored Declarations

I am deeply attached to the DRY principle: Don't Repeat Yourself. I also like to think of this more positively as SPOD: Single Point of Definition.

When you building code on top of your data structures, as you do with PL/SQL, pretty clearly your most important "point of definition" are those structures: tables and views.

So if you need to declare a variable or constant with the same type as a (and usually to hold a value from) column in a table, you should literally declare it that way with the %TYPE anchor:

```
DECLARE
  l_name employees.last_name%TYPE;
  c_hdate CONSTANT employees.hire_date%TYPE;
```

If you need to declare a record with the same structure as an entire row in a table or view, go with %ROWTYPE:

```
DECLARE
  l_employee employees%ROWTYPE;
```

Not only do you avoid copying and hard-coding the datatype (most critically the maximum length of your VARCHAR2 string), but whenever the object to which you anchored changes, the program unit containing the anchoring will be marked INVALID and recompiled automatically by the PL/SQL engine.

And after that recompilation, the datatype for your declarations will be updated to match the underlying structure. Check out my LiveSQL script for a demonstration of this wonderfulness.

Smart, tightly integrated database programming languages do a lot of fine work on our behalf!

Variables

You declare a variable when you need to manipulate it (set, change and use its value) in your block. A variable declaration always specifies the name and data type of the variable. For most data types, a variable declaration can also specify an initial value. If you include the NOT NULL constraint in the declaration, then you *must* provide an initial value (as with a constant, see below).

The variable name must be a valid user-defined identifier . The data type can be any PL/SQL data type. The PL/SQL data types include the SQL data types. A data type is either scalar (without internal components) or composite (with internal components). Here are some examples:

```
DECLARE
  /* Initial value set to NULL by default */
  l_max_salary NUMBER;
  /* Assigning an initial static value */
  l_min_salary NUMBER := 10000;
  /* Assigning an initial value with a function call */
  l_hire_date  DATE := SYSDATE;
```

And here you see what happens when I declare a variable to be NOT NULL but do not provide an initial value:

```
DECLARE
  l_date DATE NOT NULL;
BEGIN
  l_date := DATE '2011-10-30';
END;
PLS-00218: a variable declared NOT NULL must have an initialization assignment
```

Tips for Variables

- Use consistent naming conventions for your variables and constants. For example, I generally use a "g_" prefix on global variables (declared at the package level), "l_" for local variables, an "c_" for constants.
- If you find yourself declaring a whole lot of variables with similar names, they probably belong "together" - in which case consider declaring a user-defined record type. Here's an example:

```
/* Instead of this... */
DECLARE
  l_name1          VARCHAR2 (100);
  l_total_sales1   NUMBER;
  l_deliver_pref1 VARCHAR2 (10);
  --
  l_name2          VARCHAR2 (100);
  l_total_sales2   NUMBER;
  l_deliver_pref2 VARCHAR2 (10);
BEGIN
/* Try something like this... */
DECLARE
  TYPE customer_info_rt IS RECORD (
    name          VARCHAR2 (100),
    total_sales   NUMBER,
    deliver_pref  VARCHAR2 (10)
  );
  l_customer1 customer_info_rt;
  l_customer2  customer_info_rt;
```

Constants

A constant is a variable whose value cannot be changed after it is declared. A constant declaration always specifies the name and data type of the constant. Differently from a variable, you *must* assign a value to that identifier right in the declaration itself.

This works:

```
DECLARE
  c_date CONSTANT DATE := DATE '2011-10-30';
BEGIN
```

This does not work:

```
DECLARE
```

```
  c_date CONSTANT DATE;
BEGIN
  c_date := DATE '2011-10-30';
END;
```

PLS-00322: declaration of a constant 'C_DATE' must contain an initialization assignment

The expression to the right of the assignment in a constant declaration does not have to be a literal. It can be any expression that evaluates, implicitly or explicitly, to the correct datatype.

EXPERIMENT-11

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 datatypes.

Going back to our dynamic SQL example, we often we see people creating code such as this (actually we see them posting it as it's not working for them, because they don't understand bind variables and datatypes)...

```
create or replace function myfn(p_hiredate in date) return number is v_empno number;
  v_sql varchar2(32767);
begin
  v_sql := 'select empno from emp where hiredate =
to_date('''||to_char(p_hiredate,'DD/MM/YYYY')||''','DD/MM/YYYY')';
execute immediate v_sql into v_empno;
  return v_empno;
end;
/
Function created.
```

The developer is trying to concatenate in a date or varchar variable with the appropriate single quotes and formatting required to make the SQL make sense. Not only does that prevent the SQL explain plan from being re-used with different values, but it makes the code hard to maintain and get right in the first place (as well as leaving things open to SQL injection in some cases)

But, with bind variable, that's not necessary... simply doing...

```
create or replace function myfn(p_hiredate in date) return number is v_empno number;
  v_sql varchar2(32767);
begin
  v_sql := 'select empno from emp where hiredate = :1';
  execute immediate v_sql into v_empno using p_hiredate;
  return v_empno;
end;
/
Function created.
```

EXPERIMENT-12

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

Using the IF-THEN Statement: The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF). The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example: Using a Simple IF-THEN Statement

```
DECLARE
        sales NUMBER(8,2) := 10100;
        quota NUMBER(8,2) := 10000;
        bonus NUMBER(6,2);
        emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
    bonus := (sales - quota)/4;
    UPDATE employees SET salary = salary + bonus WHERE employee_id =emp_id;
  END IF;
END;
/
```

Using CASE Statements: Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

Example: Using the CASE-WHEN Statement

```
DECLARE
grade CHAR(1);
BEGIN
 grade := 'B';
 CASE grade
  WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
  WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
  WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
  WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
  WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
 END CASE;
END;
/
```

Controlling Loop Iterations: LOOP and EXIT Statements

LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

Using the LOOP Statement

The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an EXIT statement to stop looping and prevent an infinite loop. You can

place one or more EXIT statements anywhere inside a loop, but not outside a loop. There are two forms of EXIT statements: EXIT and

EXIT-WHEN.

Using the EXIT Statement

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

Using the EXIT-WHEN Statement

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

Labeling a PL/SQL Loop

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

Using the WHILE-LOOP Statement

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

Example: Using a Simple FOR..LOOP Statement

```
DECLARE
p NUMBER := 0;
BEGIN
FOR k IN 1..500 LOOP -- calculate pi with 500 terms
  p := p + ( ( (-1) ** (k + 1) ) / ((2 * k) - 1) );
END LOOP;
p := 4 * p;
DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); -- print result
END;
/
```

Sequential Control: GOTO and NULL Statements

The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

Using the GOTO Statement

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements.

Example : Using a Simple GOTO Statement

```
DECLARE
```

```
  p VARCHAR2(30);
  n PLS_INTEGER := 37; -- test any integer > 2 for prime
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
   IF n MOD j = 0 THEN -- test for prime
     p := ' is not a prime number'; -- not a prime number
     GOTO print_now;
   END IF;
END LOOP;
p := ' is a prime number';<<print_now>>
DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

Using the NULL Statement

The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Example: Using the NULL Statement to Show No Action

```
DECLARE
v_job_id VARCHAR2(10);
v_emp_id NUMBER(6) := 110;
BEGIN
 SELECT job_id INTO v_job_id FROM employees WHERE employee_id = v_emp_id;
 IF v_job_id = 'SA_REP' THEN
  UPDATE employees SET commission_pct = commission_pct * 1.2;
 ELSE
  NULL; -- do nothing if not a sales representative
 END IF;
END;
/
```

EXPERIMENT-13

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

Composite types have internal components that can be manipulated individually, such as the elements of an array, record, or table.

Oracle Times Ten In-Memory Database supports the following composite data types:

Associative array (index-by table)
Nested table
Varray
Record

Associative arrays, nested tables, and varrays are also referred to as collections.

The following sections discuss the use of composite data types:

Using collections in PL/SQL
Using records in PL/SQL
Using associative arrays from applications

See "PL/SQL Collections and Records" in Oracle Database PL/SQL Language Reference for additional information.

Using collections in PL/SQL

You can declare collection data types similar to arrays, sets, and hash tables found in other languages. A collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection.

In PL/SQL, array types are known as varrays (variable size arrays), set types are known as nested tables, and hash table types are known as associative arrays or index-by tables. These are all collection types.

Example: Using a PL/SQL collection type

This example declares collection type staff_list as a table of employee_id, then uses the collection type in a loop and in the WHERE clause of the SELECT statement.

```
DECLARE
TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
staff staff_list;
lname  employees.last_name%TYPE;
fname employees.first_name%TYPE;
BEGIN
staff := staff_list(100, 114, 115, 120, 122);
FOR i IN staff.FIRST..staff.LAST LOOP
  SELECT last_name, first_name INTO lname, fname FROM employees WHERE
employees.employee_id = staff(i);
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(staff(i)) || ': ' || lname || ', ' || fname );
END LOOP;
END;
/
```

Output:

```
100: King, Steven
114: Raphaely, Den
115: Khoo, Alexander
120: Weiss, Matthew
```

122: Kaufling, Payam

PL/SQL procedure successfully completed.

Using records in PL/SQL

Records are composite data structures that have fields with different data types. You can pass records to subprograms with a single parameter. You can also use the %ROWTYPE attribute to declare a record that represents a row in a table or a row from a query result set, without specifying the names and types for the fields, as shown in Example 2-2.

Example Declaring a record type. Declare various record types.

```
DECLARE
TYPE timerec IS RECORD (hours SMALLINT, minutes SMALLINT);
TYPE meetin_typ IS RECORD (date_held DATE,
duration timerec, -- nested record
location VARCHAR2(20),
purpose VARCHAR2(50));
BEGIN
   ...
END;
 /
```

Conversion between PL/SQL data types

Times Ten supports implicit and explicit conversions between PL/SQL data types.

Consider this example: The variable v_sal_hike is of type VARCHAR2. When calculating the total salary, PL/SQL first converts v_sal_hike to NUMBER then performs the operation. The result is of type NUMBER. PL/SQL uses implicit conversion to obtain the correct result.

Command> DECLARE

```
v_salary NUMBER (6) := 6000;
v_sal_hike VARCHAR2(5) := '1000';
v_total_salary v_salary%TYPE;
BEGIN
v_total_salary := v_salary + v_sal_hike;
DBMS_OUTPUT.PUT_LINE (v_total_salary);
end;
/
```

Output:

7000

PL/SQL procedure successfully completed.

Understanding exceptions: This section provides an overview of exceptions in PL/SQL programming, covering the following topics:

An exception is a PL/SQL error that is raised during program execution, either implicitly by Times Ten or explicitly by your program. Handle an exception by trapping it with a handler or propagating it to the calling environment.

For example, if your SELECT statement returns multiple rows, Times Ten returns an error (exception) at runtime. As the following example shows, you would see Times Ten error 8507, then the associated ORA error message. (ORA messages, originally defined for Oracle Database, are similarly implemented by Times Ten.)

```
DECLARE
v_lname VARCHAR2 (15);
BEGIN
```

```
SELECT last_name INTO v_lname FROM employees WHERE first_name = 'John';
DBMS_OUTPUT.PUT_LINE ('Last name is :' || v_lname);
END;
/
```

Output:

```
8507: ORA-01422: exact fetch returns more than requested number of rows
8507: ORA-06512: at line 4
The command failed.
```

You can handle such exceptions in your PL/SQL block so that your program completes successfully. For example:

```
DECLARE
v_lname VARCHAR2 (15);
BEGIN
SELECT last_name INTO v_lname FROM employees WHERE first_name = 'John';
DBMS_OUTPUT.PUT_LINE ('Last name is :' || v_lname);
EXCEPTION
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE (' Your SELECT statement retrieved multiple
rows. Consider using a cursor.');
END;
/
PL/SQL procedure successfully completed.
```

Your SELECT statement retrieved multiple rows. Consider using a cursor.

Example 4-1 Using the ZERO_DIVIDE predefined exception

In this example, a PL/SQL program attempts to divide by 0. The ZERO_DIVIDE predefined exception is used to trap the error in an exception-handling routine.

```
Command> DECLARE v_invalid PLS_INTEGER;
BEGIN
v_invalid := 100/0;
EXCEPTION
WHEN ZERO_DIVIDE THEN
DBMS_OUTPUT.PUT_LINE ('Attempt to divide by 0');
END;
/
Attempt to divide by 0
PL/SQL procedure successfully completed.
Trapping user-defined exceptions
```

You can define your own exceptions in PL/SQL in Times Ten, and you can raise user-defined exceptions explicitly with either the PL/SQL RAISE statement or the RAISE_APPLICATION_ERROR procedure.

Using the RAISE statement:

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler. RAISE statements can raise predefined exceptions, or user-defined exceptions whose names you decide.

Example 4-2 Using RAISE statement to trap user-defined exception

In this example, the department number 500 does not exist, so no rows are updated in the departments table. The RAISE statement is used to explicitly raise an exception and display an error message, returned by the SQLERRM built-in function, and an error code, returned by the SQLCODE built-in function. Use the RAISE statement by itself within an exception handler to raise the same exception again and propagate it back to the calling environment.

```
DECLARE
v_deptno NUMBER := 500;
v_name VARCHAR2 (20) := 'Testing';
e_invalid_dept EXCEPTION;
BEGIN
UPDATE departments
SET department_name = v_name
WHERE department_id = v_deptno;
IF SQL%NOTFOUND THEN
  RAISE e_invalid_dept;
END IF;
ROLLBACK;
EXCEPTION
 WHEN e_invalid_dept THEN
DBMS_OUTPUT.PUT_LINE ('No such department');
DBMS_OUTPUT.PUT_LINE (SQLERRM);
DBMS_OUTPUT.PUT_LINE (SQLCODE);
END;
/
No such department
User-Defined Exception
1
PL/SQL procedure successfully completed.
The command succeeded.
```

Using the RAISE_APPLICATION_ERROR procedure:

Use the RAISE_APPLICATION_ERROR procedure in the executable section or exception section (or both) of your PL/SQL program. Times Ten reports errors to your application so you can avoid returning unhandled exceptions.

Use an error number between -20,000 and -20,999. Specify a character string up to 2,048 bytes for your message.

Example: Using the RAISE_APPLICATION_ERROR procedure

This example attempts to delete from the employees table where last_name=Patterson. The RAISE_APPLICATION_ERROR procedure raises the error, using error number -20201.

```
DECLARE
v_last_name employees.last_name%TYPE := 'Patterson';
BEGIN
DELETE FROM employees WHERE last_name = v_last_name;
IF SQL%NOTFOUND THEN
   RAISE_APPLICATION_ERROR (-20201, v_last_name || ' does not exist');
END IF;
```

```
END;
/
```

Output

```
8507: ORA-20201: Patterson does not exist
8507: ORA-06512: at line 6
The command failed.
```

EXPERIMENT-14

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

Procedure:

In PL/SQL, we can pass parameters to procedures and functions in three ways.

1) IN type parameter: These types of parameters are used to send values to stored procedures.

2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.

3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.

NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

IN Parameter: This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter. We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

General syntax to pass a IN parameter is:

CREATE [OR REPLACE] PROCEDURE procedure_name (
param_name1 IN datatype, param_name12 IN datatype ...)

- param_name1, param_name2... are unique parameter names.
- datatype - defines the datatype of the variable.
- IN - is optional, by default it is a IN type parameter.

2) OUT Parameter: The OUT parameters are used to send the OUTPUT from a procedure or a function. This is a write-only parameter i.e, we cannot pass values to OUT paramters while executing the stored procedure, but we can assign values to OUT parameter inside the stored procedure and the calling program can recieve this output value. The General syntax to create an OUT parameter is:

CREATE [OR REPLACE] PROCEDURE proc2 (param_name OUT datatype)

The parameter should be explicitly declared as OUT parameter.

3) INOUT Parameter: The INOUT parameter allows us to pass values into a procedure and get output values from the procedure. This parameter is used if the value of the IN parameter can be changed in the calling program. By using INOUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter. But this is possible only if the value passed to the procedure and output value have a same datatype. This parameter is used if the value of the parameter will be changed in the procedure. The General syntax to create an INOUT parameter is:

CREATE [OR REPLACE] PROCEDURE proc3 (param_name INOUT datatype)

Using INOUT parameter: Let's create a procedure which gets the name of the employee when the employee id is passed.

```
SQL> CREATE or replace PROCEDURE emp_name(id IN NUMBER,empname OUT
varchar2) IS
  BEGIN
    SELECT ename INTO empname FROM emp WHERE empno= id;
  END;
  /
Procedure created.
```

We can call the procedure 'emp_name' in this way from a PL/SQL Block.

```
SQL> DECLARE
  empName varchar(20);
```

```
CURSOR id_cur is SELECT empno FROM emp;
BEGIN
FOR emp in id_cur
 LOOP
  emp_name(emp.empno, empName);
  dbms_output.put_line('The employee ' || empName || ' has id ' ||emp.empno);
 END LOOP;
END;
/
```

Output:

```
The employee SMITH has id 7369
The employee ALLEN has id 7499
The employee WARD has id 7521
The employee JONES has id 7566
The employee MARTIN has id 7654
The employee BLAKE has id 7698
The employee CLARK has id 7782
The employee SCOTT has id 7788
The employee KING has id 7839
The employee TURNER has id 7844
The employee ADAMS has id 7876
The employee JAMES has id 7900
The employee FORD has id 7902
The employee MILLER has id 7934
PL/SQL procedure successfully completed.
```

Using IN OUT parameter in procedures:

```
SQL> CREATE OR REPLACE PROCEDURE emp_salary_increase(emp_id IN
        empnew.empno%type, salary_inc IN OUT empnew.sal%type) IS
         tmp_sal number;
  BEGIN
   SELECT sal INTO tmp_sal FROM empnew WHERE empno = emp_id;
   IF tmp_sal between 1000 and 2000 THEN
      salary_inc := tmp_sal * 1.2;
   ELSIF tmp_sal between 2000 and 3000 THEN
      salary_inc := tmp_sal * 1.3;
   ELSIF tmp_sal > 3000 THEN
     salary_inc := tmp_sal * 1.4;
   END IF;
  END;
  /
Procedure created.
```

The below PL/SQL block shows how to execute the above 'emp_salary_increase' procedure.

```
SQL> DECLARE CURSOR updated_sal IS
      SELECT empno,sal FROM empnew;
      pre_sal number;
     BEGIN
      FOR emp_rec IN updated_sal LOOP
       pre_sal := emp_rec.sal;
       emp_salary_increase(emp_rec.empno, emp_rec.sal);
       dbms_output.put_line('The salary of ' || emp_rec.empno || 'increased from ' || pre_sal || '
  to '||emp_rec.sal);
```

```
 END LOOP;
END;
/
```

Output:

```
The salary of 7369 increased from 800 to      800
The salary of 7499 increased from 1600 to     1920
The salary of 7521 increased from 1250 to     1500
The salary of 7566 increased from 2975 to     3867.5
The salary of 7654 increased from 1250 to     1500
The salary of 7698 increased from 2850 to     3705
The salary of 7782 increased from 2450 to     3185
The salary of 7788 increased from 3000 to     3900
The salary of 7839 increased from 5000 to     7000
The salary of 7844 increased from 1500 to     1800
The salary of 7876 increased from 1100 to     1320
The salary of 7900 increased from 950 to      950
The salary of 7902 increased from 3000 to     3900
The salary of 7934 increased from 1300 to     1560
PL/SQL procedure successfully completed.
```

Functions

Types of functions depending on the their definition:

- Built-in Functions
- User Defined Functions

Difference between Procedure and Function

Procedure	Function
A procedure never returns a value to the calling portion of code	A function returns exactly one value to the calling program
A procedure may have a RETURN statement or may not. Procedures with RETURN statement, simply the control of execution is transferred back to the portion of code that called the procedure	It is mandatory for a function to have at least one RETURN statement
Procedures are not allowed to be used in DML/DRL statements Eg. insert into abc values(sum(sal)); This is not valid if sum is a procedure.	Functions are allowed to be used in DML/DRL statements. User-defined functions cannot be used in CHECK or DEFAULT constraints. Eg. insert into abc values(sum(sal)); This is valid if sum is a function.
Procedures are generally designed specifically to implement DML operations.	The DML operations are restricted to use inside the function.

PL/SQL Package

A package is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification (spec) and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms.

The following is contained in a PL/SQL package:

- Get and Set methods for the package variables, if we want to avoid letting other procedures read and write them directly.

- Cursor declarations with the text of SQL queries. Reusing exactly the same query with slight differences. It is also easier to maintain.
- Declarations for exceptions.
- Declarations for procedures and functions that call each other.
- Declarations for overloaded procedures and functions.
- Variables that we want to remain available between procedure calls in the same session. We can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored procedures or functions, we must declare the type in a package so that both the calling and called subprogram can refer to it.

Advantages of PL/SQL Packages:

Modularity: Packages let we encapsulate logically related types, items, and subprograms in a named PL/SQL module.

Easier Application Design: When designing an application, all we need initially is the interface information in the package specs. We need not define the package bodies fully until we are ready to complete the application.

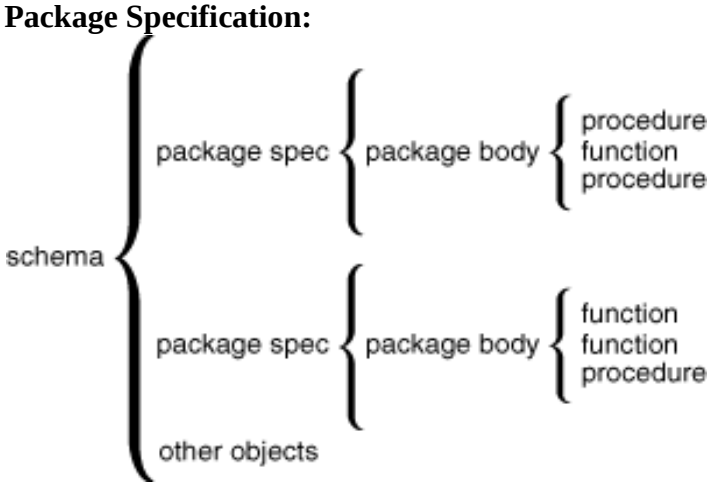
Information Hiding: With packages, we can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible).

Added Functionality: Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that execute in the environment.

Better Performance: When we call a packaged subprogram for the first time, the whole package is loaded into memory.

Packages stop cascading dependencies and avoid unnecessary recompiling. For example, if we change the body of a packaged function, Oracle does not recompile other subprograms that call the function; these subprograms only depend on the parameters and return value that are declared in the spec, so they are only recompiled if the spec changes.

Package Specification:



A Simple Package Specification Without a Body

```
CREATE PACKAGE trans_data AS -- bodiless package
  TYPE TimeRec IS RECORD (
    minutes SMALLINT,
    hours  SMALLINT);
  TYPE TransRec IS RECORD (
    category VARCHAR2(10),
    account  INT,
    amount REAL,
    time_of  TimeRec);
  minimum_balance    CONSTANT REAL := 10.00;
  number_processed INT;
```

```
  insufficient_funds EXCEPTION;
END trans_data;
/
```

Referencing Package Contents: To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation:

package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name

We can reference package contents from database triggers, stored subprograms. The following example calls the hire_employee procedure from an anonymous block in a Pro*C program. The actual parameters emp_id, emp_lname, and emp_fname are host variables.

```
EXEC SQL EXECUTE
 BEGIN
  emp_actions.hire_employee(:emp_id,:emp_lname,:emp_fname, ...);
```

Restrictions: We cannot reference remote packaged variables, either directly or indirectly. For example, we cannot call the a procedure through a database link if the procedure refers to a packaged variable.

Note: Inside a package, we cannot reference host variables.

Creating Forms

We can include forms that enable users to update just a single row in a table or multiple rows at once. Application Builder includes a number of wizards you can use to create forms automatically, or we can create forms manually.

We can also create a form manually by performing the following steps:

- Create an HTML region (to serve as a container for your page items)
- Create items to display in the region
- Create processes and branches

To create a form manually by creating and HTML region:

Navigate to the appropriate Page Definition. See "Accessing a Page Definition".

Create an HTML region:

Under Regions, click the Create icon.
Select the region type HTML.
Follow the on-screen instructions.

Start adding items to the page:

Under Items, click the Create icon.
Follow the on-screen instructions.

Assume also there are three buttons labeled Insert, Update, and Delete. Also assume you have a table T that contains the columns id, first_name, and last_name. The table has a trigger that automatically populates the ID column when there is no value supplied.

To process the insertion of a new row, you create a conditional process of type PL/SQL that executes when the user clicks the Insert button. For example:

```
BEGIN
  INSERT INTO T ( first_name, last_name )
    VALUES  (:P1_FIRST_NAME, :P1_LAST_NAME);
END;
```

To process the updating of a row, you create another conditional process of type PL/SQL.

For example:

```
BEGIN
UPDATE T
```

```
 SET first_name = :P1_FIRST_NAME,last_name = :P1_LAST_NAME WHERE ID =
:P1_ID;
END;
```

To process the deletion of a row, you create a conditional process that executes when the user clicks the Delete button. For example:

```
BEGIN
   DELETE FROM T
   WHERE ID = :P1_ID;
END;
```

EXPERIMENT-15

15. Write a PL/SQL Code for Employee Information System etc

This user exit concatenates form fields. To call the user exit from a Oracle Forms trigger, use the syntax

<user_exit>('CONCAT <field1>, <field2>, ..., <result_field>');

where user_exit is a packaged procedure supplied with Oracle Forms and CONCAT is the name of the user exit. A sample CONCAT form invokes the user exit. For more information about Oracle Forms user exits, see Chapter 11 of the Programmer's Guide to the Oracle Precompilers.

Note: The sample code listed is for a Oracle*Forms user exit and is not intended to be compiled in the same manner as the other programs.

```
      INTEGER FUNCTION CONCAT (CMD,CMDL,ERR,ERRL,INQRY)
        EXEC SQL BEGIN DECLARE SECTION
          LOGICAL*1   VALUE(81)
          LOGICAL*1   FINAL(241)
          LOGICAL*1 FIELD(81)
        EXEC SQL END DECLARE SECTION
        EXEC SQL INCLUDE SQLCA
        EXEC SQL WHENEVER SQLERROR GO TO 999
        LOGICAL*1 CMD(80)
        LOGICAL*1   ERR(80)
        INTEGER*2  CMDL, ERRL, INQRY
C       CERR IS A DYNAMICALLY BUILT ERROR MESSAGE RETURNED
C       TO SQL*FORMS.
        LOGICAL*1   CERR(80)
C       TEMPORARY VARIABLES TO DO STRING MANIPULATIONS.
        INTEGER*2 CMDCNT
        INTEGER*2 FLDCNT
        INTEGER*2   FNLCNT
C       INITIALIZE VARIABLES.
        DO 1 I = 1, 81
          FIELD(I) = ' '
1         VALUE(I) = ' '
        DO 2 I = 1, 241
2         FINAL(I) = ' '
        FNLCNT = 0
C       STRIP CONCAT FROM COMMAND LINE.
        CMDCNT = 7
        I = 1
C       LOOP UNTIL END OF COMMAND LINE.
        DO WHILE (CMDCNT .LE. CMDL)
C         PARSE EACH FIELD DELIMITED BY A COMMA.
          FLDCNT = 0
          DO WHILE ((CMD(CMDCNT) .NE. ',').AND.(CMDCNT .LE. CMDL))
            FLDCNT = FLDCNT + 1
            FIELD(FLDCNT) = CMD(CMDCNT)
            CMDCNT = CMDCNT + 1
          END DO
          IF (CMDCNT .LT. CMDL) THEN
C           WE HAVE FIELD1...FIELDN.  THESE ARE NAMES OF
```

```
//          SQL*FORMS FIELDS; GET THE VALUE.
            EXEC IAF GET :FIELD INTO :VALUE
//          REINITIALIZE FIELD NAME.
            DO 20 K = 1, FLDCNT
20              FIELD(K) = ' '
//          MOVE VALUE RETRIEVED FROM FIELD TO A CHARACTER
//          TO FIND LENGTH.
            DO WHILE (VALUE(I) .NE. ' ')
              FNLCNT = FNLCNT + 1
              FINAL(FNLCNT) = VALUE(I)
              I = I + 1
            END DO
            I = 1
            CMDCNT = CMDCNT + 1
          ELSE
//            WE HAVE RESULT_FIELD; STORE IN SQL*FORMS FIELD.
            EXEC IAF PUT :FIELD VALUES (:FINAL)
          END IF
        END DO
//      ALL OK.  RETURN SUCCESS CODE.
        CONCAT = IAPSUC
        RETURN
//      ERROR OCCURRED.  PREFIX NAME OF USER EXIT TO ORACLE
//      ERROR MESSAGE, SET FAILURE RETURN CODE, AND EXIT.
999     CERR(1) = 'C'
        CERR(2) = 'O'
        CERR(3) = 'N'
        CERR(4) = 'C'
        CERR(5) = 'A'
        CERR(6) = 'T'
        CERR(7) = ':'
        CERR(8) = ' '
        DO 1000 J = 1, 70
          CERR(J + 8) = SQLEMC(J)
1000    CONTINUE
        ERRL = 78
        CALL SQLIEM (CERR, ERRL)
        CONCAT = IAPFAI
        RETURN
        END
```

EXPERIMENT-16

16. Demonstration of database connectivity

The code below is to help you get started. Copy the code into your your JSP directory under /usr/local/etc/httpd/htdocs/html/LOGIN (where LOGIN is your login name). You will need to change the 'xxxxxx' of

"con=DriverManager.getConnection("jdbc:mysql://localhost/"+db,user,"xxxxxxx");"

to reflect your MySQL password. Copy the HTML form (right click and view source) to the same directory above or to your public_html directory and use it connect to your database via the JSP code. Replace the 'test418' with your database name and the URL with your JSP URL.

```
<%@ page import="java.sql.*"%>
<html>
<head>
<title>JDBC Connection example</title>
</head>
<body>
<h1>JDBC Connection example</h1>
<%
  String db = request.getParameter("db");
  String user = db; // assumes database name is the same as username
  try {
    java.sql.Connection con;
    Class.forName("org.gjt.mm.mysql.Driver");
    con = DriverManager.getConnection("jdbc:mysql://localhost/"+db, user, "xxxxxxx");
    out.println (db+ "database successfully opened.");
  }
  catch(SQLException e) {
    out.println("SQLException caught: " +e.getMessage());
  }
%>
</body>
</html>
```