KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore-641 021

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF CS, CA & IT

CLASS: III- B. Sc (CS) **SEMESTER : V**

17CSU504A **Oracle (SQL/PL-SQL)** **3H – 3C**

Instruction Hours / week: L: 3 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

SCOPE

The Objective of Relational Database Management System including relational, object-relational, and object-oriented systems, SQL standards, algebraic query languages, integrity constraints, triggers, functional dependencies, and normal forms. Other topics include tuning database transactions, security from the application perspective, and data warehousing.

OBJECTIVES

- Understand the role and nature of relational database management systems (RDBMS) in today's IT environment.
- Translate written business requirements into conceptual entity-relationship data models.
- Convert conceptual data models into relational database schemas using the SQL Data Definition Language (DDL).
- Query and manipulate databases using the SQL Data Manipulation Language (DML).

UNIT-I

Introduction to Oracle as RDBMS SQL Vs. SQL * Plus: SQL Commands and Data types, Operators and Expressions, Introduction to SQL * Plus.

UNIT-II

Managing Tables and Data: Creating and Altering Tables (Including constraints) ,Data Manipulation Command like Insert, update, delete, SELECT statement with WHERE, GROUP BY and HAVING, ORDER BY, DISTINCT, Special operator like IN, ANY, ALL BETWEEN, EXISTS, LIKE, Join, Built in functions

UNIT-III

Other Database Objects - View, Synonyms, Index

UNIT-IV

Transaction Control Statements - Commit, Rollback, Savepoint

UNIT-V

Introduction to PL/SQL SQL v/s PL/SQL, PL/SQL Block Structure, Language construct of PL/SQL (Variables, Basic and Composite Data type, Conditions looping etc.) TYPE and % ROWTYPE , Using Cursor (Implicit, Explicit)

Suggested Readings

1. Ivan Bayross. (2010). SQL, PL/SQL the Programming Language of Oracle. New Delhi: BPB Publications.
2. Steven Feuerstein., & Bill Pribyl. (2014). Oracle PL/SQL Programming (6th ed.). O'Reilly Media.
3. Rajeeb, C. Chatterjee. (2012). Learning Oracle SQL and PL/SQL: A simplified Guide. New Delhi: PHI.
4. Ron Hardman.,& Michael Mclaughlin. (2005). Expert Oracle PL/SQL. Oracle Press.
5. Michael Mclaughlin. (2008). Oracle Database 11g PL/SQL Programming. Oracle Press.
6. John Watson.,& Roopesh Ramklass. (2008). OCA Oracle Database11g SQL Fundamentals I Exam Guide. Oracle Press.

Websites

W1: https://www.tutorialspoint.com/plsql/
W2: http://plsql-tutorial.com/

ESE MARKS ALLOCATION

1.	**Section A** 20 X1 = 20 (Online Examination)	20
2.	**Section B** 5X2 = 10	10
3	**Section C** 5X6 = 30 (Either 'A' or 'B' Choice)	30
	Total	**60**

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
DEPARTMENT OF CS, CA & IT
LESSON PLAN

SUBJECT NAME: Oracle (SQL/PL-SQL)

SUBJECT CODE: 17CSU504A **SEMESTER: V**

STAFF: K. BANUROOPA **CLASS: III B.Sc. CS A& B**

S.No	Lecture Duration (Hr)	Topics Covered	Reference Materials
Unit I			
1.	1	**Introduction to Oracle as RDBMS** SQL Vs. SQL * Plus	S1:10-14
2.	1	SQL Commands	S1:23-25
3.	1	Data types	S1-457:451
4.	1	Operators and Expressions	S1-113:132
5.	1	Introduction to SQL * Plus.	S1:26-33
6.	1	Recapitulation and Possible Questions Discussion	
Total No of hours for Unit 1:6			
Unit II			
1.	1	**Managing Tables and Data:** Creating and Altering Tables (Including constraints)	S1:461-473
2.	1	Data Manipulation Command: Insert,	S1:403-405
3.	1	Data Manipulation Command: Update, delete,	S1:406-408
4.	1	SELECT statement with WHERE, GROUP BY	S1:103-113 S1:287-291
5.	1	SELECT statement with HAVING, ORDER BY, DISTINCT	S1:294-296 S1:136-141
6.	1	Special operators: IN, ANY, ALL	W1
7.	1	Special operators: BETWEEN, EXISTS, LIKE,	W2
8.	1	Join, Built in functions	S1:309-331
9.	1	Recapitulation and Possible Questions Discussion	
Total No of hours for Unit 2:9			
Unit –III			
1.	1	**Other Database Objects** - View -CREATE VIEW, ALTER VIEW, and DROP VIEW	S1: 487-495
2.	1	Retrieve Data from Views	S1: 496-497
3.	1	Synonyms-Create Private and Public Synonyms	S1: 498-500
4.	1	Index-Create and Maintain Indexes	S1: 509-510
5.	1	Types of Index	S1: 511-516
6.	1	Modifying and Dropping Indexes	S1: 517-518

7.	1	Recapitulation and Possible Questions Discussion	
Total No of hours for Unit 3:7			
Unit – IV			
1.	1	**Transaction Control Statements -** Database Transactions	S1:427-428
2.	1	Commit	S1:431-439
3.	1	Rollback	W3
4.	1	Savepoint	W3
5.	1	Recapitulation and Possible Questions Discussion	
Total No of hours for Unit 4:5			
Unit – V			
1.	1	**Introduction to PL/SQL** SQL v/s PL/SQL	S2:3-7,W4
2.	1	PL/SQL Block Structure	S2:53-62
3.	1	Language construct of PL/SQL (Variables, Basic and Composite Data type, etc.)	S2:63-74 S2:83-94
4.	1	Conditions, looping	S2:105-116
5.	1	TYPE and % ROWTYPE	S1:323-330,W5
6.	1	Using Cursor (Implicit, Explicit)	S2:485-512
7.	1	Recapitulation and Possible Questions Discussion	
8.	1	Previous year end-semester question paper discussion	
Total No of hours for Unit 5:09			
Total No. Of Hours Allocated: 36			

SUGGESTED READINGS:

Suggested Readings

1. John Watson.,& Roopesh Ramklass. (2008). OCA Oracle Database11g SQL Fundamentals I Exam Guide. Oracle Press.
2. Ivan Bayross. (2010). SQL, PL/SQL the Programming Language of Oracle. New Delhi: BPB Publications.
3. Steven Feuerstein., & Bill Pribyl. (2014). Oracle PL/SQL Programming (6th ed.). O'Reilly Media.
4. Rajeeb, C. Chatterjee. (2012). Learning Oracle SQL and PL/SQL: A simplified Guide. New Delhi: PHI.
5. Ron Hardman.,& Michael Mclaughlin. (2005). Expert Oracle PL/SQL. Oracle Press.
6. Michael Mclaughlin. (2008). Oracle Database 11g PL/SQL Programming. Oracle Press.

Web Sites:

W1. https://www.geeksforgeeks.org/sql-all-and-any/
W2. https://www.oracletutorial.com/oracle-basics/oracle-like/
W3. https://www.oracle-dba-online.com/sql/commit_rollback_savepoint.htm
W4. https://www.tutorialspoint.com/plsql/
W5. http://plsql-tutorial.com/

UNIT I

SYLLABUS

Introduction to Oracle as RDBMS SQL Vs. SQL * Plus: SQL Commands and Data types, Operators and Expressions, Introduction to SQL * Plus.

Introduction to Oracle as RDBMS

A database is a collection of data treated as a unit. The purpose of a database is to store and retrieve related information. A database server is the key to information management. In general, a server reliably manages a large amount of data in a multiuser environment so that many users can concurrently access the same data. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.

ORACLE is a fourth generation relational database management system. In general, a database management system (DBMS) must be able to reliably manage a large amount of data in a multi-user environment so that many users can concurrently access the same data. All this must be accomplished while delivering high performance to the users of the database. A DBMS must also be secure from unauthorized access and provide efficient solutions for failure recovery. The ORACLE Server provides efficient and effective solutions for the major database features.

ORACLE consists of many tools that allow you to create an application with ease and flexibility. You must determine how to implement your requirements using the features available in ORACLE, along with its tools. The features and tools that you choose to use to implement your application can significantly affect the performance of your application.

Several of the more useful features available to ORACLE application developers are integrity constraints, stored procedures and packages, database triggers, cost-based optimizer, shared SQL, locking and sequences.

In Oracle database management, PL/SQL is a procedural language extension to Structured Query Language (SQL). The purpose of PL/SQL is to combine database language and procedural programming language. The basic unit in PL/SQL is called a block, which is made up of three parts: a declarative part, an executable part, and an exception-building part.

SQL Vs. SQL * Plus

The scope of SQL includes data insert, query, update and delete.

SQL*Plus is an interactive and batch query tool that is installed with every Oracle Database Server or Client installation. It has a command-line user interface, a Windows Graphical User Interface (GUI) and the iSQL* Plus web-based user interface.

SQL*Plus has its own commands and environment, and it provides access to the Oracle Database. It enables you to enter and execute SQL, PL/SQL

SQL is a language, while **SQL*Plus** is a tool.

SQL is the query language used for communication with **Oracle server** to access and modify the data.
SQL* Plus is a command line tool with which you can send SQL queries to the server. Also, it can help you format the query result.

SQL is a language which is invented by **IBM**.
SQL * Plus is a tool to use SQL language for a database from Oracle corporation.

SQL can be simply used to ask queries, i.e. it involves **DML**, **DDL** and **DCL**.
SQL * Plus is command line tool which doesn't involve DML, DDL and DCL.

In SQL, there is no continuation character..
Whereas, in SQL * Plus there is a continuation character.

Keywords cannot be abbreviated in SQL.
But keywords can be abbreviated in SQL*Plus.

SQL uses functions to manipulate the data.
SQL * plus uses commands to manipulate the data.

SQL Commands

SQL, **S**tructured **Q**uery **L**anguage, is a programming language designed to manage data stored in relational databases. SQL operates through simple, declarative statements. This keeps data accurate and secure, and it helps maintain the integrity of databases, regardless of size.

DML

DML is abbreviation of **Data Manipulation Language**. It is used to retrieve, store, modify, delete, insert and update data in database.

Examples: SELECT, UPDATE, INSERT statements

DDL

DDL is abbreviation of **Data Definition Language**. It is used to create and modify the structure of database objects in database.

Examples: CREATE, ALTER, DROP statements

DCL

DCL is abbreviation of **Data Control Language**. It is used to create roles, permissions, and referential integrity as well it is used to control access to database by securing it.

Examples: GRANT, REVOKE statements

TCL

TCL is abbreviation of **Transactional Control Language**. It is used to manage different transactions occurring within a database.

Examples: COMMIT, ROLLBACK statements

Data Types in Oracle

Each column in a database table is required to have a name and a data type.

A datatype associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one datatype differently from values of another datatype; for example, Oracle can add values of NUMBER datatype but not values of RAW datatype.

Oracle supplies the following built-in datatypes:

- character datatypes
  - CHAR
  - NCHAR
  - VARCHAR2 and VARCHAR
  - NVARCHAR2
  - CLOB
  - NCLOB
  - LONG
- NUMBER datatype
- DATE datatype
- binary datatypes
  - BLOB

- BFILE
- RAW
- LONG RAW

Another datatype, ROWID, is used for values in the ROWID pseudocolumn, which represents the unique address of each row in a table.

Datatype	Description	Column Length and Default
CHAR (*size*)	Fixed-length character data of length *size* bytes.	Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (one-byte or multibyte) before setting *size.*
VARCHAR2 (*size*)	Variable-length character data.	Variable for each row, up to 4000 bytes per row. Consider the character set (one-byte or multibyte) before setting *size.* A maximum *size* must be specified.
NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the national character set.	Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum*size* is determined by the number of bytes required to store one character, with an upper limit of 2000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
NVARCHAR2 (*size*)	Variable-length character data of length *size* characters or bytes, depending on national character set. A maximum *size* must be specified.	Variable for each row. Column *size* is the number of characters for a fixed-width national character set or the number of bytes for a varying-width national character set. Maximum *size* is determined by the number of bytes required to store one character, with an upper limit of 4000 bytes per row. Default is 1 character or 1 byte, depending on the character set.
CLOB	Single-byte character data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
NCLOB	Single-byte or fixed-length multibyte national character set (NCHAR) data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
LONG	Variable-length character	Variable for each row in the table, up to

	data.	2^31 - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
NUMBER (p, s)	Variable-length numeric data. Maximum precision *p* and/or scale *s*is 38.	Variable for each row. The maximum space required for a given column is 21 bytes per row.
DATE	Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E.	Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-YY) specified by NLS_DATE_FORMAT parameter.
BLOB	Unstructured binary data.	Up to 2^32 - 1 bytes, or 4 gigabytes.
BFILE	Binary data stored in an external file.	Up to 2^32 - 1 bytes, or 4 gigabytes.
RAW (size)	Variable-length raw binary data.	Variable for each row in the table, up to 2000 bytes per row. A maximum *size* must be specified. Provided for backward compatibility.
LONG RAW	Variable-length raw binary data.	Variable for each row in the table, up to 2^31 - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility.
ROWID	Binary data representing row addresses.	Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.

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.

2.1.1 Unary Operators

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

operator operand

COURSE CODE: 17CSU504A BATCH: 2017-2020
UNIT I: INTRODUCTION TO ORACLE AS RDBMS
KARPAGAM ACADEMY OF HIGHER EDUCATION
CLASS : III B.SC CS COURSE NAME: ORACLE (SQL/PL-SQL)
COURSE CODE: 17CSU504A BATCH: 2017-2020
UNIT I: INTRODUCTION TO ORACLE AS RDBMS

2.1.2 Binary Operators

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

operand1 operator operand2

2.1.3 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 2-1. 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

Precedence Level	SQL Operator
6	AND logical operator
7	OR logical operator

2.1.4 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.

2.2 Arithmetic Operators

Arithmetic operators manipulate numeric operands. The '-' operator is also used in date arithmetic. Supported arithmetic operators are listed in Table 2-2.

Table 2-2 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;

2.3 Character Operators

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

Table 2-3 Character Operators

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

2.3.1 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.

2.4 Comparison Operators

Comparison operators used in conditions that compare one expression with another are listed in Table 2-4. The result of a comparison can be TRUE, FALSE, or UNKNOWN.

Table 2-4 Comparison Operators

Operator	Description	Example
=	Equality test.	SELECT ENAME "Employee" FROM EMP

Operator	Description	Example
		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;

Operator	Description	Example
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;

2.5 Logical Operators

Logical operators which manipulate the results of conditions are listed in Table 2-5.

Table 2-5 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

2.6 Set Operators

Set operators which combine the results of two queries into a single result are listed in Table 2-6.

Table 2-6 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'

Operator	Description	Example
		MINUS SELECT SAL FROM EMP WHERE JOB = 'MANAGER');

Note: :

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

ALL

TRUE if all of the subquery values meet the condition

SELECT * FROM Products
WHERE Price > ALL (SELECT Price FROM Products WHERE Price > 50);

ProductID	ProductName	SupplierID	CategoryID	Price
1	Toy	1	1	18
2	Plastic	1	1	19
3	Steel	1	2	10

AND

SELECT * FROM Customers
WHERE City = "London" AND Country = "UK";

CustomerID	CustomerName	City	Country
4	**ContactName**	London	UK
11	Thomas Hardy	London	UK
16	Victoria Ashworth	London	U

ANY

TRUE if any of the subquery values meet the condition

SELECT * FROM Products
WHERE Price > ANY (SELECT Price FROM Products WHERE Price > 50);

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
9	A	4	6	18 - 500 g pkgs.	97
18	B	7	8	16 kg pkg.	62.5
20	C	8	3	30 gift boxes	81

BETWEEN

TRUE if the operand is within the range of comparisons

SELECT * FROM Products
WHERE Price BETWEEN 50 AND 60;

ProductID	ProductName	SupplierID	CategoryID	Price
51	A	24	7	53
59	B	28	4	55

IN

TRUE if the operand is equal to one of a list of expressions

SELECT * FROM Customers
WHERE City IN ('Paris','London');

CustomerID	CustomerName	ContactName	Address	City
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London
16	Consolidated Holdings	Elizabeth Brown	Berkeley Gardens 12 Brewery	London

NOT

Displays a record if the condition(s) is NOT TRUE

SELECT * FROM Customers
WHERE City NOT LIKE 's%';

CustomerID	CustomerName	Address	City
1	Alfreds Futterkiste	Obere Str. 57	Berlin
2	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.
3	Antonio Moreno Taquería	Mataderos 2312	México D.F.

OR

TRUE if any of the conditions separated by OR is TRUE

SELECT * FROM Customers
WHERE City = "London" OR Country = "UK";

CustomerID	CustomerName	Address	City	Country
4	A	120 Hanover Sq.	London	UK
11	B	Fauntleroy Circus	London	UK
38	C	Garden House Crowther Way	Cowes	UK

EXISTS

TRUE if the subquery returns one or more records

SELECT * FROM Products
WHERE EXISTS (SELECT Price FROM Products WHERE Price > 50);

ProductID	ProductName	SupplierID	Unit	Price
1	A	1	10 boxes x 20 bags	18
2	B	1	24 - 12 oz bottles	19
3	C	1	12 - 550 ml bottles	10

LIKE

TRUE if the operand matches a pattern.

SELECT * FROM Customers
WHERE City LIKE 's%';

CustomerID	Address	City	Country
7	24, place Kléber	Strasbourg	France
15	Av. dos Lusíadas, 23	São Paulo	Brazil
21	Rua Orós, 92	São Paulo	Brazil

SQL Expressions

An **expression** is a combination of one or more values, operators, and SQL functions that evaluates to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to CHAR datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the SELECT statement
- A condition of the WHERE clause and HAVING clause
- The CONNECT BY, START WITH, and ORDER BY clauses
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

For example, you could use an expression in place of the quoted string 'Smith' in this UPDATE statement SETclause:

```
SET last_name = 'Smith';
```

This SET clause has the expression INITCAP(last_name) instead of the quoted string 'Smith':

```
SET last_name = INITCAP(last_name);
```

Syntax

Consider the basic syntax of the SELECT statement as follows −

```
SELECT column1, column2, columnN
FROM table_name
WHERE [CONDITION|EXPRESSION];
```

There are different types of SQL expressions, which are mentioned below −

- Boolean
- Numeric
- Date

Boolean Expressions

SQL Boolean Expressions fetch the data based on matching a single value. Following is the syntax −

```
SELECT column1, column2, columnN
FROM table_name
WHERE SINGLE VALUE MATCHING EXPRESSION;
```

The following table is a simple example showing the usage of various SQL Boolean Expressions −

```
SQL> SELECT * FROM CUSTOMERS WHERE SALARY = 10000;
```

Numeric Expression

These expressions are used to perform any mathematical operation in any query. Following is the syntax −

```
SELECT numerical_expression as  OPERATION_NAME
[FROM table_name
WHERE CONDITION] ;
```

Here, the numerical_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions −

```
SQL> SELECT (15 + 6) AS ADDITION
```

There are several built-in functions like avg(), sum(), count(), etc., to perform what is known as the aggregate data calculations against a table or a specific table column.

Date Expressions

Date Expressions return current system date and time values −

```
SQL>  SELECT CURRENT_TIMESTAMP from dual;
+--------------------+
| Current_Timestamp   |
+--------------------+
| 2009-11-12 06:40:23 |
+--------------------+
```

Another date expression is as shown below −

```
SQL>  SELECT  GETDATE();;
+-------------------------+
| GETDATE                 |
+-------------------------+
| 2009-10-22 12:07:18.140 |
+-------------------------+
```

Introduction to SQL * Plus

SQL*Plus is essentially an interactive query tool with some scripting capabilities. We can enter a SQL statement, such as a SELECT query, and view the results. We can execute *data definition language* (DDL) statements to create tables and other objects. DBAs can use SQL*Plus to start up, shut down, and otherwise administer a database.

In spite of all, GUI-based SQL generators contained in products such as PowerBuilder, Clear Access, and Crystal Reports, it is quicker and easier to build up and test a complex query in SQL*Plus before transferring it to whatever development tool.

Uses for SQL*Plus

Originally developed simply as a way to enter queries and see results, SQL*Plus has been enhanced with scripting and formatting capabilities and can be used for many different purposes. The basic functionality is simple. With SQL*Plus, you can do the following:

- Issue a SELECT query and view the results.
- Insert, update, and delete data from database tables.
- Submit PL/SQL blocks to the Oracle server for execution.

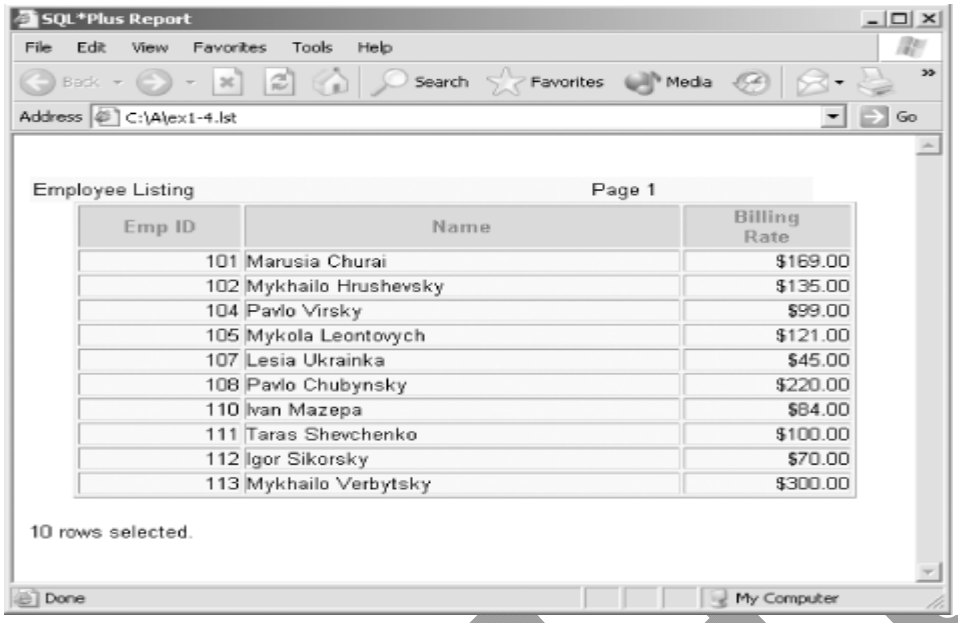
- Issue DDL statements, such as those used to create, alter, or drop database objects (e.g., tables, indexes, and users), as well as any other types of SQL statements that Oracle supports.
- Execute SQL*Plus script files.
- Write output to a file.
- Execute procedures and functions that are stored in a database.

Beginning with SQL*Plus in Oracle8*i* Database, you can use the SET MARKUP HTML command to generate HMTL output, such as that shown below.

QL*Plus report formatted in HTML

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=US-ASCII">
<meta name="generator" content="SQL*Plus 10.1.0">
<style type='text/css'> body {font:10pt Arial,Helvetica,
sans-serif; color:black; background:White;}
...
<tr>
<td align="right">
      101
</td>
<td>
Marusia Churai
</td>
<td align="right">
 $169.00
</td>
</tr>
<tr>
<td align="right">
      102
</td>
<td>
Mykhailo Hrushevsky
</td>
...
```

By writing such HTML output to a file, you can easily generate ad hoc reports for users to view from a corporate intranet. One DBA whom I spoke with regularly refreshes the phone list on his departmental intranet using this mechanism. The output is rendered in a browser.



Of course, it's rare that you would issue such a simple statement, or just one statement, when you add a new user. Usually, you also want to assign a default tablespace and often a quota on that tablespace. You may also want to grant the privilege needed to connect to the database. Whenever you have a task that requires a sequence of statements to be executed, you can simplify things by taking advantage of SQL*Plus's scripting capabilities. The statements in Example 1-5, when placed in a script file, allow you to add a new user with just one command.

Example 1-5. Script to create a new database user

```
CREATE USER &&1 IDENTIFIED BY &&2
  DEFAULT TABLESPACE users
  TEMPORARY TABLESPACE temp
  QUOTA &&3.M ON users;

GRANT CONNECT TO &&1;
```

The &&1, &&2, and &&3 in Example 1-5 are SQL*Plus user variables marking the locations at which to insert parameters that you pass to the script. Assuming that you give the name *create_user.s ql* to the file shown in Example 1-5, and assuming that you are the DBA, you can issue the following command from SQL*Plus whenever you need to add a user to your database:

Example 1-6 shows how this works, by creating a user named sql_dude with a password of yooper and a quota of 10 megabytes.

Example 1-6. Running a script to create a new database user

```
SQL> @ex1-5 sql_dude yooper 10
old   1: CREATE USER &&1 IDENTIFIED BY &&2
new   1: CREATE USER sql_dude IDENTIFIED BY yooper
old   4:    QUOTA &&3.M ON users
new   4:    QUOTA 10M ON users

User created.

old   1: GRANT CONNECT TO &&1
new   1: GRANT CONNECT TO sql_dude

Grant succeeded.
```

Example 1-7. "Hello World" written as a PL/SQL block and executed from SQL*Plus

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2    DBMS_OUTPUT.PUT_LINE('Hello World!');
  3  END;
  4  /
Hello World!
```

SQL*Plus's Relation to SQL, PL/SQL, and the Oracle Database

SQL*Plus is often used in conjunction with two other products, both of which have the letters "SQL" in their names. The first is SQL itself. Without a doubt, the most common use of SQL*Plus is to submit SQL statements to the database for execution. The second product is Oracle's PL/SQL procedural language. Table 1-1 provides a short summary of each of these three products.

Table 1-1. The three SQLs: SQL, PL/SQL, and SQL*Plus

Product	Description
SQL	SQL is an ANSI and ISO standard language used to insert, delete, update, and retrieve data from relational databases. SQL is also used to manage relational databases.
PL/SQL	PL/SQL is a proprietary procedural language developed by Oracle as an

Product	Description
	extension to SQL, for use in coding business rules and other procedural logic at the database level. Like SQL, PL/SQL executes inside the database engine.
SQL*Plus	SQL*Plus is an Oracle-developed tool that allows you to interactively enter and execute SQL commands and PL/SQL blocks.

Because these three products all have "SQL" as part of their names, people occasionally get confused about the relationships among them and about which statements get executed where. SQL*Plus does have its own set of commands that it recognizes and executes (for example, SET SERVEROUTPUT ON), but any SQL statements and PL/SQL blocks are sent to the database server for execution. Figure 1-2 illustrates this relationship.

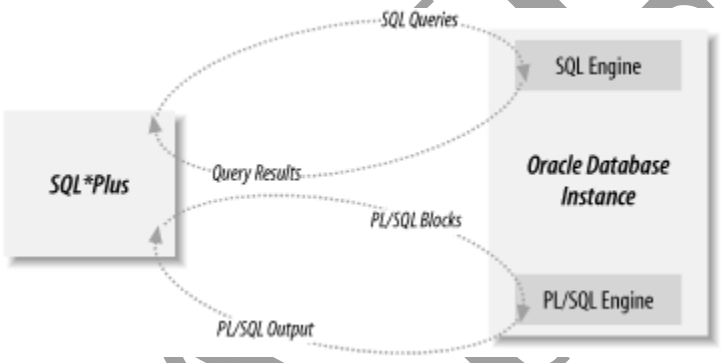


Figure 1-2. Relationships among SQL*Plus, SQL, and PL/SQL

Think of SQL*Plus as kind of a middleman, standing between you and Oracle and helping you to communicate with your database. You type in a SQL query, SQL*Plus takes it and sends it to the database, the database returns the results to SQL*Plus, and SQL*Plus displays those results in a format you can understand.

POSSIBLE QUESTIONS
UNIT I

2 marks Questions:

1. What is a SQL command?
2. Define TCL.
3. Define SQL*Plus.
4. What is meant by SQL expression?
5. What is meant by DML?
6. Define SQL operators.

6 marks Questions:

1. Differentiate SQL and SQL*Plus.
2. List and explain SQL commands.
3. Describe about SQL datatypes.
4. Explain in detail about SQL expressions.
5. Elaborate SQL*Plus.
6. Explain DDL commands with example.

UNIT II

SYLLABUS

Managing Tables and Data: Creating and Altering Tables (Including constraints) ,Data Manipulation Command like Insert, update, delete, SELECT statement with WHERE, GROUP BY and HAVING, ORDER BY, DISTINCT, Special operator like IN, ANY, ALL BETWEEN, EXISTS, LIKE, Join, Built in functions

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Persons
ADD UNIQUE (ID);

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)
VALUES (*value1*, *value2*, *value3*, ...);

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO *table_name*
VALUES (*value1*, *value2*, *value3*, ...);

The SQL GROUP BY Statement

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

GROUP BY Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *condition*
GROUP BY *column_name(s)*
ORDER BY *column_name(s);*

Example
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;

SQL - WHERE Clause

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc., which we would examine in the subsequent chapters.

Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

SELECT column1, column2, columnN
FROM table_name
WHERE [condition]

Example
SQL> SELECT ID, NAME, SALARY
FROM CUSTOMERS

WHERE SALARY > 2000;

HAVING

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

SELECT column_name, COUNT(*)
FROM table_name
GROUP BY column_name
HAVING COUNT(*) > value;

HAVING was added to SQL because the WHERE keyword could not be used with aggregate functions.

ORDER BY

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

SELECT column_name
FROM table_name
ORDER BY column_name ASC | DESC;

ORDER BY is a clause that indicates you want to sort the result set by a particular column either alphabetically or numerically.

SQL - Distinct Keyword

The SQL DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

Syntax

The basic syntax of DISTINCT keyword to eliminate the duplicate records is as follows −

```
SELECT DISTINCT column1, column2,.....columnN
FROM table_name
WHERE [condition]
```

IN

TRUE if the operand is equal to one of a list of expressions

SELECT * FROM Customers

WHERE City IN ('Paris','London');

CustomerID	CustomerName	ContactName	Address	City
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London
16	Consolidated Holdings	Elizabeth Brown	Berkeley Gardens 12 Brewery	London

NOT

Displays a record if the condition(s) is NOT TRUE

SELECT * FROM Customers

WHERE City NOT LIKE 's%';

CustomerID	CustomerName	Address	City
1	Alfreds Futterkiste	Obere Str. 57	Berlin
2	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.
3	Antonio Moreno Taquería	Mataderos 2312	México D.F.

OR

TRUE if any of the conditions separated by OR is TRUE

SELECT * FROM Customers

WHERE City = "London" OR Country = "UK";

CustomerID	CustomerName	Address	City	Country
4	A	120 Hanover Sq.	London	UK
11	B	Fauntleroy Circus	London	UK
38	C	Garden House Crowther Way	Cowes	UK

EXISTS

TRUE if the subquery returns one or more records

SELECT * FROM Products

WHERE EXISTS (SELECT Price FROM Products WHERE Price > 50);

ProductID	ProductName	SupplierID	Unit	Price
1	A	1	10 boxes x 20 bags	18
2	B	1	24 - 12 oz bottles	19
3	C	1	12 - 550 ml bottles	10

LIKE

TRUE if the operand matches a pattern.

SELECT * FROM Customers

WHERE City LIKE 's%';

CustomerID	Address	City	Country
7	24, place Kléber	Strasbourg	France
15	Av. dos Lusíadas, 23	São Paulo	Brazil
21	Rua Orós, 92	São Paulo	Brazil

SQL INNER JOIN (simple join)

SQL INNER JOINS return all rows from multiple tables where the join condition is met.

Syntax

The syntax for the INNER JOIN in SQL is:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example

Let's look at an example of how to use the INNER JOIN in a query.

In this example, we have a table called *customers* with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Sam	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

And a table called *orders* with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

Enter the following SQL statement:

```
SELECT customers.customer_id, orders.order_id, orders.order_date
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id
ORDER BY customers.customer_id;
```

Output

customer_id	order_id	order_date
4000	4	2016/04/20
5000	2	2016/04/18
7000	1	2016/04/18
8000	3	2016/04/19

SQL LEFT OUTER JOIN

Another type of join is called a LEFT OUTER JOIN. This type of join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

Syntax

The syntax for the LEFT OUTER JOIN in SQL is:

```
SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the OUTER keyword is omitted and written simply as LEFT JOIN.

The SQL LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

Using the same *customers* table as the previous example:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Sam	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

And the *orders* table with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

Enter the following SQL statement:

```
SELECT customers.customer_id, orders.order_id, orders.order_date
FROM customers
```

```
LEFT OUTER JOIN orders
ON customers.customer_id = orders.customer_id
ORDER BY customers.customer_id;
```

There will be 6 records selected. These are the results that you should see:

customer_id	order_id	order_date
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

SQL RIGHT OUTER JOIN

Another type of join is called a SQL RIGHT OUTER JOIN. This type of join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met).

Syntax

The syntax for the RIGHT OUTER JOIN in SQL is:

```
SELECT columns
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the OUTER keyword is omitted and written simply as RIGHT JOIN.

Using the same *customers* table as the previous example:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Sam	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

And the *orders* table with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

Enter the following SQL statement:

```
SELECT customers.customer_id, orders.order_id, orders.order_date
FROM customers
```

RIGHT OUTER JOIN orders

ON customers.customer_id = orders.customer_id

ORDER BY customers.customer_id;

There will be 5 records selected. These are the results that you should see:

customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
7000	1	2016/04/18
8000	3	2016/04/19

SQL FULL OUTER JOIN

Another type of join is called a SQL FULL OUTER JOIN. This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with NULL values in place where the join condition is not met.

Syntax

The syntax for the SQL FULL OUTER JOIN is:

SELECT columns

FROM table1

FULL [OUTER] JOIN table2

ON table1.column = table2.column;

In some databases, the OUTER keyword is omitted and written simply as FULL JOIN.

Using the same *customers* table as the previous example:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Sam	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

And the *orders* table with the following data:

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20
5	NULL	2016/05/01

Enter the following SQL statement:

```
SELECT customers.customer_id, orders.order_id, orders.order_date
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id
ORDER BY customers.customer_id;
```

There will be 7 records selected. These are the results that you should see:

customer_id	order_id	order_date
NULL	5	2016/05/01
4000	4	2016/04/20
5000	2	2016/04/18
6000	NULL	NULL
7000	1	2016/04/18
8000	3	2016/04/19
9000	NULL	NULL

Built in functions

Numeric Functions

Function	Description
ABS	Returns the absolute value of a number
AVG	Returns the average value of an expression
CEILING	Returns the smallest integer value that is greater than or equal to a number
COUNT	Returns the count of an expression
FLOOR	Returns the largest integer value that is equal to or less than a number
MAX	Returns the maximum value of an expression
MIN	Returns the minimum value of an expression

RAND	Returns a random number or a random number within a range
ROUND	Returns a number rounded to a certain number of decimal places
SIGN	Returns a value indicating the sign of a number
SUM	Returns the summed value of an expression

Conversion Functions

Function	Description
CAST	Converts an expression from one data type to another
CONVERT	Converts an expression from one data type to another

POSSIBLE QUESTIONS
UNIT-II

2 marks questions

1. What is known as constraints?
2. Define primary key.
3. Define foreign key.
4. What is the use of WHERE clause?
5. Define FULL OUTER JOIN.

6 marks questions

1. Elaborate Constraints with suitable queries.
2. Explain about Data Manipulation Language.
3. Explain in detail about special operators.
4. Discuss the concept of join.
5. Explain about Built in functions in SQL.

UNIT III
SYLLABUS

Other Database Objects - View, Synonyms, Index

VIEW

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records −

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

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.

- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
  SET AGE =35
  WHERE name ='Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
  WHERE age =22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

Force VIEW Creation

FORCE keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATEor REPLACE FORCEVIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition;
```

Update a VIEW

UPDATE command for view is same as for tables.

Syntax to Update a View is,

```
UPDATEview-name SETVALUE
WHERE condition;
```

NOTE: If we update a view it also updates base table data automatically.

Read-Only VIEW

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

```
CREATEor REPLACE FORCEVIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition WITHread-only;
```

The above syntax will create view for **read-only** purpose, we cannot Update or Insert data into read-only view. It will throw an **error**.

Types of View

There are two types of view,

- Simple View
- Complex View

Simple View	Complex View
Created from one table	Created from one or more table
Does not contain functions	Contain functions
Does not contain groups of data	Contains groups of data

Synonyms

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

Create Synonym (or Replace)

You may wish to create a synonym so that users do not have to prefix the table name with the schema name when using the table in a query.

Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name
  FOR [schema .] object_name [@ dblink];
```

OR REPLACE

Allows you to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC

It means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema

The appropriate schema. If this phrase is omitted, Oracle assumes that you are referring to your own schema.

object_name

The name of the object for which you are creating the synonym. It can be one of the following:

- table
- view
- sequence

- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object
- synonym

Example
Let's look at an example of how to create a synonym in Oracle.
For example:

```
CREATE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

This first CREATE SYNONYM example demonstrates how to create a synonym called *suppliers*. Now, users of other schemas can reference the table called *suppliers* without having to prefix the table name with the schema named *app*. For example:

```
SELECT *
FROM suppliers;
```

If this synonym already existed and you wanted to redefine it, you could always use the *OR REPLACE* phrase as follows:

```
CREATE OR REPLACE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

Drop synonym
Once a synonym has been created in Oracle, you might at some point need to drop the synonym.
Syntax
The syntax to drop a synonym in Oracle is:

```
DROP [PUBLIC] SYNONYM [schema .] synonym_name [force];
```

PUBLIC
Allows you to drop a public synonym. If you have specified *PUBLIC*, then you don't specify a *schema*.
force
It will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use *force* as it can cause invalidation of Oracle objects.
Example
Let's look at an example of how to drop a synonym in Oracle.
For example:

```
DROP PUBLIC SYNONYM suppliers;
```

This DROP statement would drop the synonym called *suppliers* that we defined earlier.

SQL Index

Index in sql is created on existing tables to retrieve the rows quickly.

When there are thousands of records in a table, retrieving information will take a long time. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly. Indexes can be created on a single column or a group of columns. When a index is created, it first sorts the data and then it assigns a ROWID for each row.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and are then referred to one or more specific page numbers.

An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.

Create an Index

Syntax

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

Let's look at an example of how to create an index in Oracle/PLSQL.

For 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;
```

Create a Function-Based Index

In Oracle, you are not restricted to creating indexes on only columns. You can create function-based indexes.

Syntax

The syntax for creating a function-based index in Oracle/PLSQL is:

```
CREATE [UNIQUE] INDEX index_name
 ON table_name (function1, function2, ... function_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.
- **function1, function2, ... function_n**
  - The functions 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

Let's look at an example of how to create a function-based index in Oracle/PLSQL.

For example:

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

In this example, we've created an index based on the uppercase evaluation of the *supplier_name* field.

However, to be sure that the Oracle optimizer uses this index when executing your SQL statements, be sure that UPPER(supplier_name) does not evaluate to a NULL value. To ensure this, add **UPPER(supplier_name) IS NOT NULL** to your WHERE clause as follows:

```
SELECT supplier_id, supplier_name, UPPER(supplier_name)
```

```
FROM supplier
WHERE UPPER(supplier_name) IS NOT NULL
ORDER BY UPPER(supplier_name);
```

Rename an Index

Syntax

The syntax for renaming an index in Oracle/PLSQL is:

```
ALTER INDEX index_name
  RENAME TO new_index_name;
```

index_name

The name of the index that you wish to rename.

new_index_name

The new name to assign to the index.

Example

Let's look at an example of how to rename an index in Oracle/PLSQL.

For example:

```
ALTER INDEX supplier_idx
  RENAME TO supplier_index_name;
```

In this example, we're renaming the index called *supplier_idx* to *supplier_index_name*.

Collect Statistics on an Index

If you forgot to collect statistics on the index when you first created it or you want to update the statistics, you can always use the ALTER INDEX command to collect statistics at a later date.

Syntax

The syntax for collecting statistics on an index in Oracle/PLSQL is:

```
ALTER INDEX index_name
  REBUILD COMPUTE STATISTICS;
```

index_name

The index in which to collect statistics.

Example

Let's look at an example of how to collect statistics for an index in Oracle/PLSQL.

For example:

```
ALTER INDEX supplier_idx
  REBUILD COMPUTE STATISTICS;
```

In this example, we're collecting statistics for the index called supplier_idx.

Drop an Index

Syntax

The syntax for dropping an index in Oracle/PLSQL is:

```
DROP INDEX index_name;
```

Where **index_name -**The name of the index to drop.
Example
Let's look at an example of how to drop an index in Oracle/PLSQL.
For example:

```
DROP INDEX supplier_idx;
```

In this example, we're dropping an index called supplier_idx.

UNIT-III

POSSIBLE QUESTIONS

2 marks questions

1. Define view.
2. How to create a view?
3. What is WITH CHECK OPTION used for?
4. What is the use of FORCE keyword?
5. Define synonyms.
6. What is an index?

6 marks questions

1. Discuss the different types of Indexes with examples.
2. What is the purpose of using synonym? Explain with example.
3. Illustrate creation, modification and deletion of views with appropriate examples.

UNIT IV

SYLLABUS

Transaction Control Statements - Commit, Rollback, Savepoint

Commit, Rollback and Savepoint SQL commands

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

COMMIT Statement

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Syntax

The syntax for the COMMIT statement in Oracle/PLSQL is:

```
COMMIT [ WORK ] [ COMMENT clause ] [ WRITE clause ] [ FORCE clause ];
```

Parameters or Arguments

WORK

Optional. It was added by Oracle to be SQL-compliant. Issuing the COMMIT with or without the WORK parameter will result in the same outcome.

COMMENT clause

Optional. It is used to specify a comment to be associated with the current transaction. The comment that can be up to 255 bytes of text enclosed in single quotes. It is stored in the system view called DBA_2PC_PENDING along with the transaction ID if there is a problem.

WRITE clause

Optional. It is used to specify the priority that the redo information for the committed transaction is to be written to the redo log. With this clause, you have two parameters to specify:

- *WAIT* or *NOWAIT* (*WAIT* is the default if omitted)
  - *WAIT* - means that the commit returns to the client only after the redo information is persistent in the redo log.
  - *NOWAIT* - means that the commit returns to the client right away regardless of the status of the redo log.
- *IMMEDIATE* or *BATCH* (*IMMEDIATE* is the default if omitted)
  - *IMMEDIATE* - forces a disk I/O causing the log writer to write the redo information to the redo log.
  - *BATCH* - forces a "group commit" and buffers the redo log to be written with other transactions.

FORCE clause

Optional. It is used to force the commit of a transaction that may be corrupt or in doubt. With this clause, you can specify the FORCE in 3 ways:

- *FORCE 'string', [integer]* or *FORCE CORRUPT_XID 'string'* or *FORCE CORRUPT_XID_ALL*
  - *FORCE 'string', [integer]* - allows you to commit a corrupt or in doubt transaction in a distributed database system by specifying the transaction ID in single quotes as *string*. You can find the transaction ID in the system view called DBA_2PC_PENDING. You can specify *integer* to assign the transaction a system change number if you do not wish to commit the transaction using the current system change number.
  - *FORCE CORRUPT_XID 'string'* - allows you to commit a corrupt or in doubt transaction by specifying the transaction ID in single quotes as *string*. You can find the transaction ID in the system view called V$CORRUPT_XID_LIST.
  - *FORCE CORRUPT_XID_ALL* - allows you to commit all corrupted transactions.

Note

- You must have DBA privileges to access the system views - DBA_2PC_PENDING and V$CORRUPT_XID_LIST.
- You must have DBA privileges to specify certain features of the COMMIT statement.

Example

Let's look at an example that shows how to issue a commit in Oracle using the COMMIT statement.

For example:

```
COMMIT;
```

This COMMIT example would perform the same as the following:

```
COMMIT WORK WRITE WAIT IMMEDIATE;
```

In this example, the WORK keyword is implied and the omission of the WRITE clause would default to WRITE WAIT IMMEDIATE so the first 2 COMMIT statements are equivalent.

Comment

Let's look at an example of a COMMIT that shows how to use the *COMMENT* clause:

For example, you can write the COMMIT with a comment in two ways:

```
COMMIT COMMENT 'This is the comment for the transaction';
```

OR

```
COMMIT WORK COMMENT 'This is the comment for the transaction';
```

Since the WORK keyword is always implied, both of these COMMIT examples are equivalent. The COMMIT would store the comment enclosed in quotes along with the transaction ID in the DBA_2PC_PENDING system view, if the transaction was in error or in doubt.

Force

Finally, look at an example of a COMMIT that shows how to use the FORCE clause.

For example, you can write the COMMIT of an in-doubt transaction in two ways:

```
COMMIT FORCE '22.14.67';
```

OR

```
COMMIT WORK FORCE '22.14.67';
```

Since the WORK keyword is always implied, both of these COMMIT examples would force the commit of the corrupted or in doubt transaction identified by the transaction ID '22.14.67'.

ROLLBACK

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.

ROLLBACKTOsavepoint_name;

Syntax

The syntax for the ROLLBACK statement is:

```
ROLLBACK [ WORK ] [ TO [SAVEPOINT] savepoint_name  | FORCE 'string' ];
```

Parameters or Arguments

WORK

Optional. It was added by Oracle to be SQL-compliant. Issuing the ROLLBACK with or without the WORK parameter will result in the same outcome.

TO SAVEPOINT savepoint_name

Optional. The ROLLBACK statement undoes all changes for the current session up to the savepoint specified by *savepoint_name*. If this clause is omitted, then all changes are undone.

FORCE 'string'

Optional. It is used to force the rollback of a transaction that may be corrupt or in doubt. With this clause, you specify the transaction ID in single quotes as *string*. You can find the transaction ID in the system view called DBA_2PC_PENDING.

Note

- You must have DBA privileges to access the system views - DBA_2PC_PENDING and V$CORRUPT_XID_LIST.
- You can not rollback a transaction that is in doubt to a savepoint.

Example

Let's look at an example that shows how to issue a rollback in Oracle using the ROLLBACK statement.

For example:

```
ROLLBACK;
```

This ROLLBACK example would perform the same as the following:

```
ROLLBACK WORK;
```

In this example, the WORK keyword is implied so the first 2 ROLLBACK statements are equivalent. These examples would rollback the current transaction.

Savepoint

Let's look at an example of a ROLLBACK that shows how to use the rollback to a specific savepoint.

For example, you can write the ROLLBACK to a savepoint in two ways:

```
ROLLBACK TO SAVEPOINT savepoint1;
```

OR

```
ROLLBACK WORK TO SAVEPOINT savepoint1;
```

Since the WORK keyword is always implied, both of these ROLLBACK examples would rollback the current transaction to the savepoint called savepoint1.

Force

Finally, look at an example of a ROLLBACK that shows how to force the rollback of a transaction that is in doubt.

For example, you can write the ROLLBACK of an in-doubt transaction in two ways:

```
ROLLBACK FORCE '22.14.67';
```

OR

```
ROLLBACK WORK FORCE '22.14.67';
```

Since the WORK keyword is always implied, both of these ROLLBACK examples would force the rollback of the corrupted or in doubt transaction identified by the transaction ID '22.14.67'.

SAVEPOINT

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINTsavepoint_name;

In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.

Using Savepoint and Rollback

Following is the table class,

id	name
1	Abhi
2	Adam
4	Alex

Lets use some SQL queries on the above table and see the results.

INSERTINTO class VALUES(5,'Rahul');

COMMIT;

```
UPDATE class SET name ='Abhijit'WHERE id ='5';

SAVEPOINT A;

INSERTINTO class VALUES(6,'Chris');

SAVEPOINT B;

INSERTINTO class VALUES(7,'Bravo');

SAVEPOINT C;

SELECT*FROM class;
```

NOTE: SELECT statement is used to show the data stored in the table.
The resultant table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

Now let's use the ROLLBACK command to roll back the state of data to the savepoint B.

```
ROLLBACKTO B;
```

SELECT*FROM class;

Now class table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

Now let's again use the ROLLBACK command to roll back the state of data to the savepoint A

ROLLBACKTO A;

SELECT*FROM class;

Now the table will look like,

id	name
1	Abhi
2	Adam

4	Alex
5	Abhijit

UNIT-IV

POSSIBLE QUESTIONS

2 marks questions

1. Define commit.
2. What is meant by Rollback?
3. Define Savepoint.
4. Give an example query for TO SAVEPOINT.

6 marks questions

1. Elaborate COMMIT with examples.
2. Explain about Rollback.
3. Elaborate Savepoint.

UNIT V
SYLLABUS

Introduction to PL/SQL SQL v/s PL/SQL, PL/SQL Block Structure, Language construct of PL/SQL (Variables, Basic and Composite Data type, Conditions looping etc.) TYPE and % ROWTYPE , Using Cursor (Implicit, Explicit)

SQL v/s PL/SQL

Difference between SQL and PL/SQL

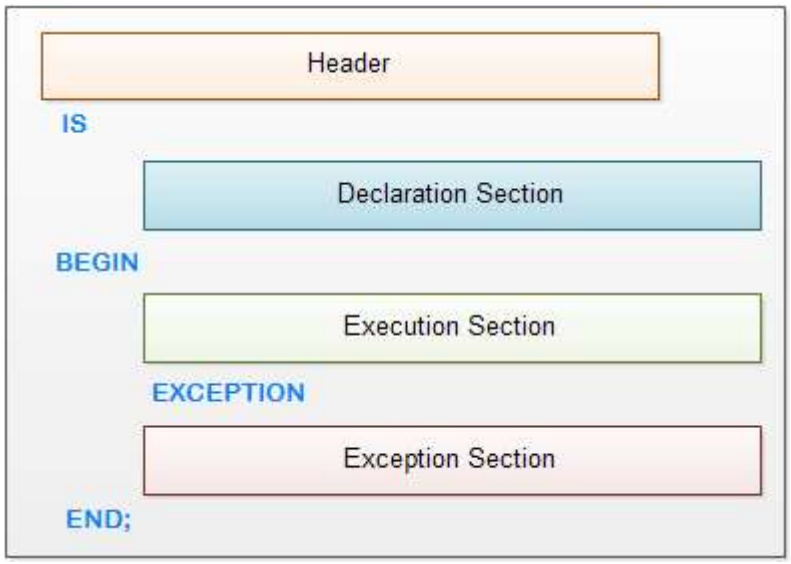
SQL	PL/SQL
• SQL is a single query that is used to perform DML and DDL operations.	• PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
• It is declarative, that defines what need to be done, rather than how things need to be done.	• PL/SQL is procedural that defines how the things needs to be done.
• Execute as a single statement.	• Execute as a whole block.
• Mainly used to manipulate data.	• Mainly used to create an application.
• Interaction with a Database server.	• No interaction with the database server.
• Cannot contain PL/SQL code in it.	• It is an extension of SQL, so that it can contain SQL inside it.

PL/SQL BLOCK STRUCTURE

PL/SQL is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts.

S.No	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

2	**Executable Commands** This section is enclosed between the keywords **BEGIN**and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.



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

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

The 'Hello World' Example

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

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

```
Hello World

PL/SQL procedure successfully completed.
```

The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL −

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator

||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE
  -- variable declaration
message  varchar2(20):= 'Hello, World!';
BEGIN
  /*
*  PL/SQL executable statement(s)
  */
dbms_output.put_line(message);
END;
/
```

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

```
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL Program Units

A PL/SQL unit is any one of the following −

- PL/SQL block
- Function
- Package
- Package body

- Procedure
- Trigger
- Type
- Type body

LANGUAGE CONSTRUCT OF PL/SQL

Variables

A variable is a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype*mustbe a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below −

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example −

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

- The **DEFAULT** keyword
- The **assignment** operator

For example −

```
counterbinary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL**value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables −

```
DECLARE
a integer :=10;
b integer :=20;
c integer;
f real;
BEGIN
c:= a + b;
dbms_output.put_line('Value of c: '|| c);
f:=70.0/3.0;
dbms_output.put_line('Value of f: '|| f);
END;
/
```

When the above code is executed, it produces the following result −

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global**variables in its simple form −

```
DECLARE
--Global variables
num1 number :=95;
num2 number :=85;
BEGIN
dbms_output.put_line('Outer Variable num1: '|| num1);
dbms_output.put_line('Outer Variable num2: '|| num2);
  DECLARE
--Local variables
```

```
num1 number :=195;
num2 number :=185;
BEGIN
dbms_output.put_line('Inner Variable num1: '|| num1);
dbms_output.put_line('Inner Variable num2: '|| num2);
END;
END;
/
```

When the above code is executed, it produces the following result −

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY   DECIMAL (18,2),
   PRIMARY KEY (ID)
);

TableCreated
```

Let us now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1,'Ramesh',32,'Ahmedabad',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2,'Khilan',25,'Delhi',1500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3,'kaushik',23,'Kota',2000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4,'Chaitali',25,'Mumbai',6500.00);
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5,'Hardik',27,'Bhopal',8500.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6,'Komal',22,'MP',4500.00);
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
c_idcustomers.id%type:=1;
c_namecustomers.name%type;
c_addrcustomers.address%type;
c_salcustomers.salary%type;
BEGIN
   SELECT name, address, salary INTO c_name,c_addr,c_sal
   FROM customers
   WHERE id =c_id;
dbms_output.put_line
('Customer '||c_name||' from '||c_addr||' earns '||c_sal);
END;
/
```

When the above code is executed, it produces the following result −

```
Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully
```

Data Types

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB**data types in this chapter. The other two data types will be covered in other chapters.

S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**.
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	**Composite** Data items that have internal components that can be accessed individually. For example, collections and records.

4	**Reference** Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories −

S.No	Date Type & Description
1	**Numeric** Numeric values on which arithmetic operations are performed.
2	**Character** Alphanumeric values that represent single characters or strings of characters.
3	**Boolean** Logical values on which logical operations are performed.
4	**Datetime** Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types −

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number

5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration −

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

```
/
```

When the above code is compiled and executed, it produces the following result −

```
PL/SQL procedure successfully completed
```

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types −

S.No	Data Type & Description
1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes
7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

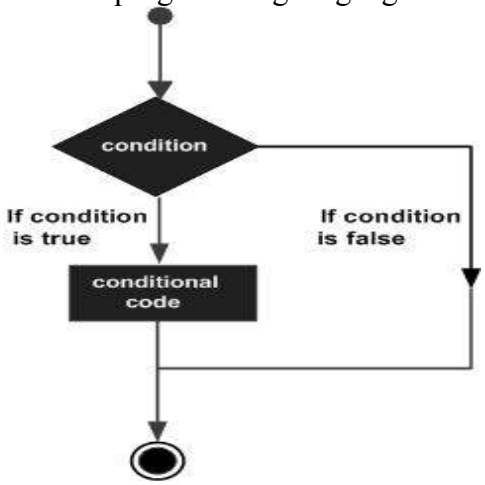
PL/SQL BOOLEAN DATA TYPES

The BOOLEAN data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.

Control Structures in PL/SQL

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

S.No	Statement & Description
1	**IF - THEN statement** The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
2	**IF-THEN-ELSE statement** **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
3	**IF-THEN-ELSIF statement** It allows you to choose between several alternatives.

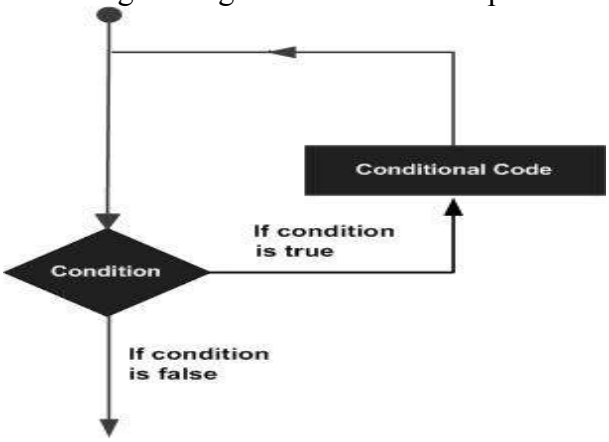
4	**Case statement** Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
5	**Searched CASE statement** The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
6	**nested IF-THEN-ELSE** You can use one **IF-THEN** or **IF-THEN-ELSIF**statement inside another **IF-THEN** or **IF-THEN-ELSIF**statement(s).

LOOPING

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

S.No	Loop Type & Description
1	**PL/SQL Basic LOOP** In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

2	**PL/SQL WHILE LOOP** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	**PL/SQL FOR LOOP** Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	**Nested loops in PL/SQL** You can use one or more loop inside any another basic loop, while, or for loop.

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept −

```
DECLARE
i number(1);
j number(1);
BEGIN
<<outer_loop>>
  FOR i IN 1..3 LOOP
<<inner_loop>>
    FOR j IN 1..3 LOOP
dbms_output.put_line('i is: '|| i ||' and j is: '|| j);
END loop inner_loop;
END loop outer_loop;
END;
/
```

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

```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3

PL/SQL procedure successfully completed.
```

TYPE and % ROWTYPE

The %TYPE attribute provides the datatype of a variable or table column. This is particularly useful when declaring variables that will hold values of a table column. For example, suppose you want to declare variables as the same datatype as the employee_id and last_name columns in employees table. To declare variables named empid and emplname that have the same datatype as the table columns, use dot notation and the %TYPE attribute.

Using %TYPE With Table Columns in PL/SQL

```
DECLARE -- declare variables using %TYPE attribute
empidemployees.employee_id%TYPE;  -- employee_iddatatype is NUMBER(6)
emplnameemployees.last_name%TYPE;  -- last_namedatatype is VARCHAR2(25)
BEGIN
empid    := 100301;  -- this is OK because it fits in NUMBER(6)
--   empid  := 3018907;  -- this is too large and will cause an overflow
emplname := 'Patel'; --  this is OK because it fits in VARCHAR2(25)
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || empid);  -- display data
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || emplname); -- display data
END;
/
```

Using the %ROWTYPE Attribute to Declare Variables

For easier maintenance of code that interacts with the database, you can use the %ROWTYPE attribute to declare a variable that represents a row in a table. A PL/SQL record is the datatype that stores the same information as a row in a table.

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.

```
DECLARE
customer_reccustomers%rowtype;
BEGIN
  SELECT * into customer_rec
  FROM customers
  WHERE id = 5;
dbms_output.put_line('Customer ID: ' || customer_rec.id);
dbms_output.put_line('Customer Name: ' || customer_rec.name);
dbms_output.put_line('Customer Address: ' || customer_rec.address);
dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

Cursors

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

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

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors

- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL**statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
```

```
| 2 | Khilan   | 25 | Delhi     | 1500.00 |
| 3 | kaushik  | 23 | Kota      | 2000.00 |
| 4 | Chaitali | 25 | Mumbai    | 6500.00 |
| 5 | Hardik   | 27 | Bhopal    | 8500.00 |
| 6 | Komal    | 22 | MP        | 4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT**attribute to determine the number of rows affected −

```
DECLARE
total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary +500;
   IF sql%notfound THEN
dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
total_rows:=sql%rowcount;
dbms_output.put_line(total_rows||' customers selected ');
END IF;
END;
/
```

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1 | Ramesh   | 32 | Ahmedabad | 2500.00 |
| 2 | Khilan   | 25 | Delhi     | 2000.00 |
| 3 | kaushik  | 23 | Kota      | 2500.00 |
| 4 | Chaitali | 25 | Mumbai    | 7000.00 |
| 5 | Hardik   | 27 | Bhopal    | 9000.00 |
| 6 | Komal    | 22 | MP        | 5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id,c_name,c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors

```
DECLARE
c_idcustomers.id%type;
c_namecustomerS.No.ame%type;
c_addrcustomers.address%type;
  CURSOR c_customersis
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customersintoc_id,c_name,c_addr;
    EXIT WHEN c_customers%notfound;
dbms_output.put_line(c_id||' '||c_name||' '||c_addr);
END LOOP;
  CLOSE c_customers;
```

```
END;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

POSSIBLE QUESTIONS
UNIT-V

2 marks questions

1. Define PL/SQL.
2. What is meant by datatype?
3. Define Large Object.
4. Define Looping.
5. Define cursor.
6. What are the types of cursor?

6 marks questions

1. Explain about PL/SQL Block Structure.
2. Explain various types of datatypes.
3. Describe Conditions looping.
4. Elaborate TYPE and % ROWTYPE.
5. Explain cursor with examples.

Register Number_____

[17CSU504A]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore-641021.

B.Sc COMPUTER SCIENCE

FIRST INTERNAL EXAMINATION - JULY 2019

Fifth Semester

Oracle (SQL/PL-SQL)

Date & Session: .7.2019 & N **Duration: 2 Hours**

Maximum : 50 Marks **Class : III-B. Sc(CS) A & B**

PART A – (20 X 1 = 20 Marks)

ANSWER ALL THE QUESTIONS

1. The relational model is based on the concept that data stored in tables called _____
   a. Fields b. Records c. Relations d. Keys
2. Which command is used for removing a table and all its data from the database?
   a. Create command c. Alter table command
   b. Drop table command d. All of the Mentioned
3. In SQL, which of the following is not a data Manipulation Language Commands?
   a. Delete b. Truncate c. Update d. Create
4. Which of the following is not a type of SQL statement?
   a. Data Manipulation Language (DML) c. Data Control Language (DCL)
   b. Data Definition Language (DDL) d. Data Communication Language (DCL)
5. In SQL, which command is used to add new rows to a table?
   a. Alter Table c. Insert
   b. Add row d. Append
6. In SQL, which command(s) is(are) used to change a table's storage characteristics?
   a. ALTER TABLE c. CHANGE TABLE
   b. MODIFY TABLE d. All of the Mentioned
7. _____ defines rules regarding the values allowed in columns and is the standard mechanism for enforcing database integrity.
   a.Column b. Constraint c. Index d. Trigger
8. Which command is used for removing a table and all its data from the database?
   a. Create command c. Alter table command
   b. Drop table command d. All of the Mentioned
9. In SQL, which command is used to SELECT only one copy of each set of duplicable rows
   a. SELECT DISTINCT c. SELECT DIFFERENT
   b. SELECT UNIQUE d. All of the Mentioned
10. Which of the SQL statements is correct?
    a. SELECT Username AND Password FROM Users
    b. SELECT Username, Password FROM Users
    c. SELECT Username, Password WHERE Username = 'user1'
    d. None of the Mentioned
11. Let the statement : SELECT column1 FROM myTable; return 10 rows. The statement: SELECT ALL column1 FROM myTable; will return___.
    a. less than 10 rows c. exactly 10 rows
    b. more than 10 rows d. none of the above
12. What is true about Unique and primary key?
    a. Unique can have multiple NULL values but Primary can't have.
    b. Unique can have single NULL value but Primary can't have even single.
    c. Both can have duplicate values
    d. None of the above
13. What will be the consequence of omitting 'Where' clause in Update Statement?
    a. No effect on the query as well as on table.
    b. All records present in the table will be updated
    c. Only one record will be updated
    d. None of the above
14. Which one is correct syntax for Insert Statement?
    a. Insert table_name Columns(Col1, Col2,Col3);
    b. Insert into table_name (Col1, Col2,Col3) VALUES (Val1,Val2,Val3);
    c. Insert Columns(Col1, Col2,Col3) VALUE (Val1, Val2,Val3) Into table_name;
    d. None of the above
15. Which one of the following sorts rows in SQL?
    a. SORT BY b. ALIGN BY c. ORDER BY d. GROUP BY
16. Which of the constraint cannot be defined at the table level?
    a. Check c. Not null
    b. Unique d. Primary key
17. Which of the following wild card character will select all columns in a table?
    a. * b. ? c. / d. +
18. What option is used in alter table statement to change the name of an existing column?
    a. RENAME b. MODIFY c. ADD d. DROP
19. What character is used to execute the previous SQL command again in SQL *plus?
    a. * b. ? c. / d. +
20. To obtain the structure of an Oracle table, the command to use in SQL* plus is ___
    a. STRUCTURE [TableName].
    b. DESCRIBE [TableName].
    c. DESCRIBE STRUCTURE [TableName].
    d. DESC TABLE [TableName].

PART – B (3 X 2 =6 Marks)

ANSWER ALL THE QUESTIONS

21. What is the SQL* Plus?
22. If an UNIQUE KEY constraint on DATE column is created, will it accept the rows that are inserted with SYSDATE?
23. Compare CHAR and VARCHAR2 data types in Oracle.

PART – C (3 X 8 =24 Marks)

ANSWER ALL THE QUESTIONS

24. (a) Differentiate SQL and SQL*Plus. [OR]

    (b) List and explain types of SQL commands.

25. (a) Describe about SQL data types. [OR]

    (b) Explain in detail about expressions in select statement with examples.

26. (a) Elaborate the syntax of alter table statement and explain with examples [OR]

    (b) Illustrate different type of constraints defined in a table with examples.