

LAB MANUAL

DATABASE MANAGEMENT SYSTEM LAB (ECS-356)

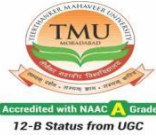
B.TECH. (CSE) III Semester

Faculty of Engineering



**TEERTHANKER MAHAVEER UNIVERSITY
MORADABAD**

2024



FACULTY OF ENGINEERING

(TEERTHANKER MAHAVEER UNIVERSITY MORADABAD)

LAB MANUAL

Submitted By

Student Name:.....

RollNo :.....

Branch:.....Section.....

Year Semester.....

Submitted To

FACULTY INCHARGE

COURSE OUTCOME:

CO 1	Understanding the concepts of DML operation to database table to complete different queries on database.
CO 2	Applying the concepts of different DDL operations.
CO 3	Applying the concepts of DCL operations like grant and revoke for administration purpose on a table.
CO 4	Applying the concepts of PL/SQL for creating different triggers to develop event driven action in database.
CO 5	Analyzing the concepts of PL/SQL for creating functions and procedure to apply DML on tables

Program Outcomes :

PO 1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2	Problem analysis& Solving: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO 3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
PO 6	Social Interaction & effective citizenship: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO 9	Attitude (Individual and team work): Function effectively as an individual, and as member or leader in diverse teams, and in multidisciplinary settings.
PO 10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a

	member and leader in a team, to manage projects and in multidisciplinary environments.
PO 12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Programme Specific Outcomes (PSOs)

PSO 1	Developing skills for creating computational solutions with emerging technologies, programming languages, mathematical foundations, algorithmic principles and open source platforms to solve complex engineering problems.
PSO 2	Ability to understand the evolutionary changes in computing by applying standard practices and skills acquired through computer engineering to provide solutions with innovative ideas and interdisciplinary research.
PSO 3	Ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur and pursue higher studies in their respective domains.

CO-PO Mapping

Course Code	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
ECS 356	CO1	2	1			1					1		1
	CO2	2	3	3	1	3				1	2		2
	CO3	3	3	3	2	3	1		1	2	2	1	3
	CO4	3	3	3	3	3	1		1	3	3	2	3
	CO5	3	3	3	3	3	1		1	3	3	3	3

CO-PSO Mapping

Course Code	CO	PSO1	PSO2	PSO3
ECS 356	CO1	3	2	3
	CO2	3	2	3
	CO3	3	2	3
	CO4	3	2	3
	CO5	3	2	3

Lab Equipment

Following hardware and software are necessary to perform the experiments in DBMS Lab.

Hardware:

1. A computer that can execute oracle RDBMS.

Software:

1. Windows Operating System.
2. Oracle

Practical Courses Internal Evaluation (50 marks)

Experiment (30 Marks)	Attendance (10 Marks)	Viva (10 Marks)	Total Internal (50 Marks)
--------------------------	--------------------------	--------------------	------------------------------

Practical Courses External Evaluation (50 marks)

The external evaluation would also be done by the External Examiner based on the experiment performed during the examination.

Experiment (30 Marks)	File Work (10 Marks)	Viva (10 Marks)	Total Internal (50 Marks)
--------------------------	-------------------------	--------------------	------------------------------

FORMAT OF THE LAB RECORD TO BE PREPARED BY THE STUDENTS

The front page of the lab record prepared by the students should have a cover page as displayed below.

PRACTICAL FILE

Of

ECS 356

Database Management System Lab.

B. TECH 2nd year, 3rd Semester



Faculty of Engineering

TMU, MORADABAD

2024-25 Odd Sem

Submitted To:

Faculty Name

Submitted By:

(Student Name)

(University Roll Number)

LAB ASSIGNMENT NAME –Basic SQL Queries.

LAB ASSIGNMENT NO. –1

Assignment 1 is based on the basic DDL and DML queries:

S.N o	PROGRAM NAME	PAGE NO	DATE	SIGN	REMARK
1	Introduction of DBMS with database languages i.e. DDL, DML, and DCL.				
2	Introduction of various Integrity and Key Constraints of DBMS.				
3	Create table Student_1 having columns Roll _no (NUMBER), Name VARCHAR(20), Age (NUMBER), Address (VARCHAR(30)), DateOfBirth (DATE).				
4	Create table Student_2 having columns Roll _no (NUMBER), Name (VARCHAR(20) not null), Age (NUMBER), Address (VARCHAR(30)), DateOfBirth (DATE), Contact (NUMBER) unique where Roll_No is Primary Key.				
5	Insert at least 5 records of Student in both student_1 and student_2. And check all constraints with their parameters.				
6	Display the structure of table student_1 and student_2.				
7	Display the records present in tables student_1 and student_2.				
8	Display the records of students belonging to Moradabad in student_2.				
9	Display the all records of student having Roll_No 3.				
10	Insert Null value for address field in Student_2.				
11	Add columns in student_2 as Year_of_admission date. Check whether it contains null values or not.				
12	Add a column in student_2 as 'Course char(10)'. Course will contains 'BTECH(CS)' as default.				
13	Modify an existing column Course i.e. type varchar2 and size 20.				
14	Drop a newly added column 'Year of admission' from Student_2.				
15	Update record of a student whose roll Number is 1 in Student_2 table.				
16	Update values of Name, Address and course for the student having Roll_no 4 in table student_2.				
17	Delete all record of student having Roll_No as 5.				
18	Delete all tuples from student_1. And check whether its schema is deleted or not.				
19	Delete the record of student having Roll_No 7 and Address as Moradabad.				
20	Display the list of Roll_Number and Name available in Student_2 table.				
21	Display the records of students having age>22.				
22	Display the records of Student_2 in decreasing order of age.				

23	Select the Roll_Number, Name and age in student_2 in increasing order of age by using order by clause.				
24	Drop the full structure of table Student_1 from database.				

LAB ASSIGNMENT NAME – SQL Queries Based on specific table and perform various selection operations.

LAB ASSIGNMENT NO. –2

Assignment 2 is based on the following Table:

<u>FIRST</u>	<u>LAST</u>	<u>ID</u>	<u>AGE</u>	<u>CITY</u>	<u>STATE</u>
John	Jones	99980	45	Payson	Arizona
Mary	Jones	99982	25	Payson	Arizona
Eric	Edwards	88232	32	San Diego	California
Mary Ann	Edwards	88233	32	Phonix	Arizona
Ginger	Howell	98002	42	Cottonwood	Arizona
Sebastian	Smith	92001	23	Gila Bend	Arizona
Gus	Gray	22322	35	Bagdad	Arizona
Mary Ann	May	32326	52	Tucson	Arizona
Erica	William	32327	60	Show	Low Arizona
Leoroy	Brown	32380	22	Pinetop	Arizona
Elroy	Cleaver	32382	22	Globe	Arizona

S.NO	PROGRAM NAME	PAGE NO	DATE	SIGN	REMARK
1	Create the table EMPINFO with ID as a Primary key constraint.				
2	Insert all the above mentioned values in table.				
3	Select all the first name ,last name & city from the table whose names starts with 'Er'				
4	Select first name & last name from the table whose last names ends with 's'				
5	Select all the details from the table whose first name equals 'Eric' exactly				
6	Select all the details of the table EMPINFO.				
7	Display the first name & age for everyone that is in the table				
8	Display the first name, last name & city for everyone that is not from 'Payson'				

9	Display all columns for everyone that is over 40 years old				
10	Display the first & last names for everyone whose last name ends in an 'ay'				
11	Display all columns for everyone whose first name equals 'Mary'				
12	Display all the columns for everyone whose first name contains 'Mar'				
13	Select first name , last name & age from the table where last name contains 'illia'				
14	Select the first name , last name , city & state whose first name starts with 'J'				
15	Display the last name and city of employees who belongs to Arizona and whose age is greater than 25				
16	Display the details of employees with id 32382, 32326 and 99980.				
17	Count the number of employees who work in Tucson and Payson				
18	Display the first name and last name in a single column				
19	Display the record like : “Hello John welcome to Payson City” where name and city has to be fetch from table for all the employees				
20	Display the details of employees whose first name and last name starts with the same character				

LAB ASSIGNMENT NAME – SQL Queries with Aggregate Functions and Clauses

LAB ASSIGNMENT NO. –3

Customer Table

Cus_ID	Cus_Name	Ref_ID	Prod_ID	Date_of_Purchase	Price	Discount
1359	Elina Taylor	1608	20	05-Oct-2006	1000	
1573	Mark Lyon	1175	40	07-Sep-2008	1550	110
1932	Stuart Twain	1608	60	23-Aug-2009	3000	230
1875	Allan Samuels	1610	80	15-Nov-2011	10000	970
1723	Peter Russel		40	17-July-2010	4000	250
1973	Andre Finch	1932	60	12-Mar-2014	6000	530
1205	Steffi Nicolus	1175	20	07-Feb-2012	1500	
1608	Adams Franklin		80	05-Jan-2010	8000	509
1175	Luke Martin	1932	60	14-Dec-2007	3500	300
1097	Joseph Milne	1610	40	07-Apr-2007	1200	80
1850	Mary Roberts	1875	20	12-Sep-2008	1170	
1610	Sheldon Moris	1175	80	08-May-2009	11050	915

Product Table

Prod_ID	Prod_Category
20	Cosmetics
40	Daily Needs
60	Garments
80	Equipment

S.NO.	PROGRAM NAME	PAGE NO	DATE	SIGN	REMARK
1	Create the Table Customer Cus_ID as Primary key				
2	Insert the given records into the table				
3	Select top 3 Customer ID from the sorted table				
4	Change Customer name into upercase.				
5	List customer name who purchahase the product ID 80.				
6	Alter Customer table to set the default value of customer name as ABC				
7	List First Customer ID whose price is less than 2000				
8	List the Customers belonging to the Product ID 20.				
9	List the ID and name of the customers whose Product Price is more than 4000				
10	List the Customer ID & ID of Reference Person.				
11	List the records in the Ascending Order of Customer Name				
12	List the Name of the Customers who purchase their products before 5-Jul-2009				
13	List the name of Customers who does not refer any person.				

14	List the different Products ID available in the Customer table.				
15	List the customer names who are not eligible for discount.				
16	List the ID of the Customers having "a" as second character of their name				
17	List the Customer name, Product Price & Product ID in ascending order of Product ID & then on descending order of Price				
18	Count the no. of different Products available in the Customer table				
19	List the total Price paid by the Customers				
20	List the maximum Price of products categorize as Equipment				
21	List the minimum price of products in Customer table				
22	List the average price and the no. of customers purchasing product having ID as 60				
23	List the average price from each job excluding Cosmetics				
24	List total price, maximum & minimum price and the average price for product Id 20 and 80.				
25	List the average price for each product within each product category.				
26	Update the discount of Mark Lyon from 110 to 100				
27	Delete the record of Luke Martin.				
28	Change the data type size of cus_name				
29	Create a copy of Customer Table as NewCustomer				
30	Remove the NewCustomer Table while maintaining it's structure				
31	Remove the NewCustomer table completely from the disk.				

LAB Assignment Name-SQL Queries with Join Operations

LAB Assignment No.-4

S.NO.	PROGRAM NAME	PAGE NO.	DATE	SIGN	REMARK
1	Display product ID and no. of customers for each Product ID.				
2	Display Customer ID, Customer Name and their respective Product Category.				
3	Display Customer name, Product Category, Price and Discount for those whose Price lie in between 5000 and 10000				
4	Display the id and name of all those customers who give the reference of at least one Customer.				
5	List the Customer Name and their respective referencee name.				
6	Display Customer Name and Product Category even if customers don't purchase a particular product category.				

7	List all customers who purchase their product before their reference person.				
8	List all customers who purchase their product after their reference person.				
9	List the Customer Names whose Price is greater than the lowest price of Product ID 60.				
10	Display Customer Name and their product price whose Product Price is greater than highest average of Product Category.				
11	List all those Customers whose product price is greater than the Product Price of their Reference Person.				
12	Display the Product Category along with total discount given for each category.				
13	Display those customers whose product price is less than the average of minimum and maximum price out of all categories.				
14	Display 3 least product prices and their ID.				
15	Count total reference persons who give their reference to customers.				

Students are advised to use Developer 2000 Oracle 8+ version for above experiments. However, depending on the availability of Software's students may use power builder/SQL Server/DB2 for implementation.

Introduction:

The purpose of the DBMS course is to introduce students to the advanced concepts of database systems, focusing on the relational algebra and data model, query optimization and transactions. Topics related to the engineering and design of database systems, including: data models; database and schema design; schema normalization and integrity constraints; query processing; query optimization and cost estimation; transactions; recovery; concurrency control; isolation and consistency; distributed and parallel databases; trigger systems; pub-sub systems; semi structured data and XML querying.

DBMS lab aims at practicing and achieving this aim by using various software's such as Oracle, MySQL, PostgreSQL. All these require a thorough practice of various DDL, DCL and DML queries.

SQL (Structured Query Language)

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in relational database.

SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server use SQL as standard database language.

Also, they are using different dialects, such as:

- ☐ MS SQL Server using T-SQL,
- ☐ Oracle using PL/SQL,
- ☐ MS Access version of SQL is called JET SQL (native format) etc.

SQL allows users to access, describe the data, define the data in database and manipulate that data, embed within other languages using SQL modules, libraries & pre-compilers, create and drop databases and tables, create view, stored procedure, functions in a database, set permissions on tables, procedures, and views.

SQL Architecture:

SQL Commands:

DDL - Data Definition Language:

Command

Description

CREATE

Creates a new table, a view of a table, or other object in database

ALTER

Modifies an existing database object, such as a table.

DROP

Deletes an entire table, a view of a table or other object in the database.

DML - Data Manipulation Language:

Command

Description

SELECT

Retrieves certain records from one or more tables

INSERT

Creates a record

UPDATE

Modifies records

DELETE

Deletes records

DCL - Data Control Language:

Command

Description

GRANT

Gives a privilege to user

REVOKE

Takes back privileges granted from user

What is RDBMS?

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

SQL SELECT Statement:

SELECT column1, column2....columnN FROM table_name;

SQL DISTINCT Clause:

SELECT DISTINCT column1, column2....columnN FROM table_name;

SQL WHERE Clause:

SELECT column1, column2....columnN FROM table_name WHERE CONDITION;

SQL AND/OR Clause:

SELECT column1, column2....columnN FROM table_name WHERE CONDITION-1
{AND|OR} CONDITION-2;

SQL IN Clause:

SELECT column1, column2....columnN FROM table_name WHERE column_name IN (val-1, val-2,...val-N);

SQL BETWEEN Clause:

SELECT column1, column2....columnN FROM table_name WHERE column_name
BETWEEN val-1 AND val-2;

SQL Like Clause:

SELECT column1, column2....columnN FROM table_name WHERE column_name LIKE { PATTERN };

SQL ORDER BY Clause:

SELECT column1, column2....columnN FROM table_name WHERE CONDITION
ORDER BY column_name {ASC|DESC};

SQL GROUP BY Clause:

SELECT SUM(column_name) FROM table_name WHERE CONDITION GROUP BY
column_name;

SQL COUNT Clause:

SELECT COUNT(column_name) FROM table_name WHERE CONDITION;

SQL HAVING Clause:

SELECT SUM(column_name) FROM table_name WHERE CONDITION GROUP BY
column_name HAVING (arithmetic function condition);

SQL CREATE TABLE Statement:

CREATE TABLE table_name(column1 datatype, column2 datatype, column3 datatype,
..... columnN datatype, PRIMARY KEY(one or more columns));

SQL DROP TABLE Statement:

DROP TABLE table_name;

SQL CREATE INDEX Statement :

CREATE UNIQUE INDEX index_name ON table_name (column1, column2,...columnN);

SQL DROP INDEX Statement :

ALTER TABLE table_name DROP INDEX index_name;

SQL DESC Statement :

DESC table_name;

SQL TRUNCATE TABLE Statement:

TRUNCATE TABLE table_name;

SQL ALTER TABLE Statement:

ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_type};

SQL ALTER TABLE Statement (Rename) :

ALTER TABLE table_name RENAME TO new_table_name;

SQL INSERT INTO Statement:

INSERT INTO table_name(column1, column2....columnN) VALUES (value1,
value2....valueN);

SQL UPDATE Statement:

UPDATE table_name SET column1 = value1, column2 = value2....columnN=valueN
[WHERE CONDITION];

SQL DELETE Statement:

DELETE FROM table_name WHERE {CONDITION};

SQL CREATE DATABASE Statement:

CREATE DATABASE database_name;

SQL DROP DATABASE Statement:

DROP DATABASE database_name;

SQL USE Statement:

USE DATABASE database_name;

SQL COMMIT Statement:

COMMIT;

SQL ROLLBACK Statement:

ROLLBACK;

SQL - Operators:

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator a + b will give 30	
-	Subtraction - Subtracts right hand operand from left hand operand a - b will give -10	
*	Multiplication - Multiplies values on either side of the operator a * b will give 200	
/	Division - Divides left hand operand by right hand operand b / a will give 2	
%	Modulus - Divides left hand operand by right hand operand and returns remainder b % a will give 0	

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true. (a = b) is not true.	
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. (a != b) is true.	
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. (a <> b) is true.	
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	
<	Checks if the value of left operand is less	

than the value of right operand, if yes then condition becomes true.

(a < b) is true.

>=

Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.

(a >= b) is not true.

<=

Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

(a <= b) is true.

!<

Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.

(a !< b) is false.

!>

Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.

(a !> b) is true.

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Operator

Description

ALL

The ALL operator is used to compare a value to all values in another value set.

AND

The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

ANY

The ANY operator is used to compare a value to any applicable value in the list according to the condition.

BETWEEN

The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.

EXISTS

The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.

IN

The IN operator is used to compare a value to a list of literal values that have been specified.

LIKE

The LIKE operator is used to compare a value to similar values using wildcard operators.

NOT

The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**

OR

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

IS NULL

The NULL operator is used to compare a value with a NULL value.

UNIQUE

The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

SQL - Useful Functions:

SQL has many built-in functions for performing processing on string or numeric data. Following is the list of all useful SQL built-in functions:

- ☐ COUNT function - The SQL COUNT aggregate function is used to count the number of rows in a database table.
- ☐ MAX Function - The SQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.
- ☐ MIN Function - The SQL MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
- ☐ AVG Function - The SQL AVG aggregate function selects the average value for certain table column.
- ☐ SUM Function - The SQL SUM aggregate function allows selecting the total for a numeric column.
- ☐ SQRT Functions - This is used to generate a square root of a given number.
- ☐ RAND Function - This is used to generate a random number using SQL command.
- ☐ CONCAT Function - This is used to concatenate any string inside any SQL command.
- ☐ Numeric Functions - Complete list of SQL functions required to manipulate numbers in SQL.
- ☐ String Functions - Complete list of SQL functions required to manipulate strings in SQL.

SQL - Constraints

Constraints are the rules enforced on data columns on table. These 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 database. Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL.

- ☐ NOT NULL Constraint: Ensures that a column cannot have NULL value.
- ☐ DEFAULT Constraint: Provides a default value for a column when none is specified.
- ☐ UNIQUE Constraint: Ensures that all values in a column are different.
- ☐ PRIMARY Key: Uniquely identified each rows/records in a database table.
- ☐ FOREIGN Key: Uniquely identified a rows/records in any another database table.
- ☐ CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.
- ☐ INDEX: Use to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

Dropping Constraints:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the

following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

Integrity Constraints:

Integrity constraints are used to ensure accuracy and consistency of data in a relational database.

Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in referential integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints.

SQL – Joins

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables, (a) CUSTOMERS table is as follows:

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

(b) Another table is ORDERS as follows:

```
+---+-----+-----+-----+
|OID | DATE | CUSTOMER_ID | AMOUNT |
+---+-----+-----+-----+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+---+-----+-----+-----+
```

Now, let us join these two tables in our SELECT statement as follows:

```
SQL> SELECT ID, NAME, AGE, AMOUNT FROM CUSTOMERS, ORDERS WHERE CUSTOMERS.ID
= ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+---+-----+---+-----+
| ID | NAME | AGE | AMOUNT |
+---+-----+---+-----+
| 3 | kaushik | 23 | 3000 |
| 3 | kaushik | 23 | 1500 |
| 2 | Khilan | 25 | 1560 |
| 4 | Chaitali | 25 | 2060 |
+---+-----+---+-----+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

SQL Join Types:

There are different types of joins available in SQL:

- ❑ INNER JOIN: returns rows when there is a match in both tables.
- ❑ LEFT JOIN: returns all rows from the left table, even if there are no matches in the right

table.

- ❑ **RIGHT JOIN:** returns all rows from the right table, even if there are no matches in the left table.
- ❑ **FULL JOIN:** returns rows when there is a match in one of the tables.
- ❑ **SELF JOIN:** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- ❑ **CARTESIAN JOIN:** returns the Cartesian product of the sets of records from the two or more joined tables.

SQL - UNIONS

The SQL **UNION** clause/operator is used to combine the results of two or more **SELECT** statements without returning any duplicate rows.

To use **UNION**, each **SELECT** must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

The basic syntax of **UNION** is as follows:

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
UNION
```

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

Here given condition could be any given expression based on your requirement.

Example:

Consider the following two tables, (a) **CUSTOMERS** table is as follows:

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

(b) Another table is **ORDERS** as follows:

```
+-----+-----+-----+-----+
|OID | DATE | CUSTOMER_ID | AMOUNT |
+-----+-----+-----+-----+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+-----+-----+-----+-----+
```

Now, let us join these two tables in our **SELECT** statement as follows:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS ON
CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
```

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS ON
CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+-----+-----+-----+-----+
| ID | NAME | AMOUNT | DATE |
+-----+-----+-----+-----+
| 1 | Ramesh | NULL | NULL |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL | NULL |
```

6	Komal	NULL	NULL
7	Muffy	NULL	NULL

The UNION ALL Clause:

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

The same rules that apply to UNION apply to the UNION ALL operator.

Syntax:

The basic syntax of **UNION ALL** is as follows:

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

```
UNION ALL
```

```
SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]
```

Here given condition could be any given expression based on your requirement.

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

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

(b) Another table is ORDERS as follows:

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, let us join these two tables in our SELECT statement as follows:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
```

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

There are two other clauses (i.e., operators), which are very similar to UNION clause:

□ **INTERSECT Clause:** is used to combine two SELECT statements, but returns rows only

from the first SELECT statement that are identical to a row in the second SELECT statement.

□ EXCEPT Clause : combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

SQL - NULL

The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

Syntax:

The basic syntax of **NULL** while creating a table:

```
SQL> 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));
```

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.

You must use the **IS NULL** or **IS NOT NULL** operators in order to check for a NULL value.

Consider the following table, CUSTOMERS 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	
7	Muffy	24	Indore	

Now, following is the usage of **IS NOT NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS WHERE SALARY IS  
NOT NULL;
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00

Now, following is the usage of **IS NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS WHERE SALARY IS  
NULL;
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
6	Komal	22	MP	
7	Muffy	24	Indore	

SQL – Alias

You can rename a table or a column temporarily by giving another name known as alias.

The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

Syntax:

The basic syntax of **table** alias is as follows:

```
SELECT column1, column2,... FROM table_name AS alias_name WHERE [condition];
```

The basic syntax of **column** alias is as follows:

```
SELECT column_name AS alias_name FROM table_name WHERE [condition];
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

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

(b) Another table is ORDERS as follows:

```
+---+-----+-----+-----+
|OID | DATE | CUSTOMER_ID | AMOUNT |
+---+-----+-----+-----+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+---+-----+-----+-----+
```

Now, following is the usage of **table alias**:

```
SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT FROM CUSTOMERS AS C, ORDERS AS O
WHERE C.ID = O.CUSTOMER_ID;
```

This would produce the following result:

```
+---+-----+---+-----+
| ID | NAME | AGE | AMOUNT |
+---+-----+---+-----+
| 3 | kaushik | 23 | 3000 |
| 3 | kaushik | 23 | 1500 |
| 2 | Khilan | 25 | 1560 |
| 4 | Chaitali | 25 | 2060 |
+---+-----+---+-----+
```

Following is the usage of **column alias**:

```
SQL> SELECT ID AS CUSTOMER_ID, NAME AS CUSTOMER_NAME FROM CUSTOMERS
WHERE SALARY IS NOT NULL;
```

This would produce the following result:

```
+-----+-----+
| CUSTOMER_ID | CUSTOMER_NAME |
+-----+-----+
| 1 | Ramesh |
| 2 | Khilan |
| 3 | kaushik |
| 4 | Chaitali |
| 5 | Hardik |
| 6 | Komal |
| 7 | Muffy |
```

SQL – Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

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

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and 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 ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is as follows:

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX index_name ON table_name (column_name);
```

Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE UNIQUE INDEX index_name on table_name (column_name);
```

Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command:

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

SQL – Alter Table

The SQL **ALTER TABLE** command is used to add, delete or modify columns and also add and drop various constraints on a an existing table.

Syntax:

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

```
ALTER TABLE table_name ADD column_name datatype;
```

The basic syntax of **ALTER TABLE** to **DROP COLUMN** in an existing table is as follows:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The basic syntax of **ALTER TABLE** to change the **DATA TYPE** of a column in a table is as

follows:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

The basic syntax of ALTER TABLE to **ADD CHECK CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:

```
ALTER TABLE table_name ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

The basic syntax of ALTER TABLE to **DROP CONSTRAINT** from a table is as follows:

```
ALTER TABLE table_name DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name DROP INDEX MyUniqueConstraint;
```

The basic syntax of ALTER TABLE to **DROP PRIMARY KEY** constraint from a table is as follows:

```
ALTER TABLE table_name DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name DROP PRIMARY KEY;
```

Example:

Consider the CUSTOMERS table having the following records:

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

Following is the example to ADD a new column in an existing table:

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

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

Following is the example to DROP sex column from existing table:

```
ALTER TABLE CUSTOMERS DROP SEX;
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

```
+---+-----+---+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Ramesh | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
```

```
| 4 | kaushik | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
```

SQL – Truncate Table

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table.

You can also use **DROP TABLE** command to delete complete table but it would remove complete table structure from the database and you would need to re-create this table once again if you wish you store some data.

Syntax:

The basic syntax of **TRUNCATE TABLE** is as follows:

```
TRUNCATE TABLE table_name;
```

Example:

Consider the **CUSTOMERS** table having the following records:

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

Following is the example to truncate:

```
SQL > TRUNCATE TABLE CUSTOMERS;
```

Now, **CUSTOMERS** table is truncated and following would be the output from **SELECT** statement:

```
SQL> SELECT * FROM CUSTOMERS;
Empty set (0.00 sec)
```

SQL – Using Views

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 kind 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 such 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.

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 very similar way as you use them in 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 |
+-----+-----+-----+-----+
```

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

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

Now, you can query CUSTOMERS_VIEW in similar way as you query an actual table.

Following is the example:

```
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 is 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:

- ☐ 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 a view. Following is 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 same would reflect in the view

itself. Now, try to query base table, and 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 can not insert rows in CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in 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 same would reflect in the view itself. Now, try to query base table, and 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 as given below:

```
DROP VIEW view_name;
```

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

```
DROP VIEW CUSTOMERS_VIEW;
```

SQL – Having Clause

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

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

Syntax:

The following is the position of the HAVING clause in a query:

```
SELECT FROM WHERE GROUP BY HAVING ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

```
SELECT column1, column2 FROM table1, table2 WHERE [ conditions ] GROUP BY
column1, column2 HAVING [ conditions ] ORDER BY column1, column2
```

Example:

Consider the CUSTOMERS table having the following records:

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

Following is the example, which would display record for which similar age count would be more than or equal to 2:

```
SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY FROM CUSTOMERS GROUP BY age  
HAVING COUNT(age) >= 2;
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00

Transaction Control:

There are following commands used to control transactions:

□

COMMIT: to save the changes.

□

ROLLBACK: to rollback the changes.

□

SAVEPOINT: creates points within groups of transactions in which to ROLLBACK

□

SET TRANSACTION: Places a name on a transaction.

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

The COMMIT Command:

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

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

The syntax for COMMIT command is as follows:

```
COMMIT;
```

Example:

Consider the CUSTOMERS table having the following records:

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

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

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

SQL> COMMIT;

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

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

The ROLLBACK Command:

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

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

The syntax for ROLLBACK command is as follows:

ROLLBACK;

Example:

Consider the CUSTOMERS table having the following records:

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

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

SQL> DELETE FROM CUSTOMERS WHERE AGE = 25;

SQL> ROLLBACK;

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

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

The SAVEPOINT Command:

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

The syntax for SAVEPOINT command is as follows:

SAVEPOINT SAVEPOINT_NAME;

This command serves only in the creation of a SAVEPOINT among transactional statements.

The ROLLBACK command is used to undo a group of transactions.

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

```
ROLLBACK TO SAVEPOINT_NAME;
```

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

Example:

Consider the CUSTOMERS table having the following records:

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

Now, here is the series of operations:

```
SQL> SAVEPOINT SP1;
```

Savepoint created.

```
SQL> DELETE FROM CUSTOMERS WHERE ID=1;
```

1 row deleted.

```
SQL> SAVEPOINT SP2;
```

Savepoint created.

```
SQL> DELETE FROM CUSTOMERS WHERE ID=2;
```

1 row deleted.

```
SQL> SAVEPOINT SP3;
```

Savepoint created.

```
SQL> DELETE FROM CUSTOMERS WHERE ID=3;
```

1 row deleted.

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
SQL> ROLLBACK TO SP2;
```

Rollback complete.

Notice that only the first deletion took place since you rolled back to SP2:

```
SQL> SELECT * FROM CUSTOMERS;
```

```
+---+-----+---+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 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 |
+---+-----+---+-----+-----+
```

6 rows selected.

The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for RELEASE SAVEPOINT is as follows:

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

The SET TRANSACTION Command:

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.

For example, you can specify a transaction to be read only, or read write.

The syntax for SET TRANSACTION is as follows:

SET TRANSACTION [READ WRITE | READ ONLY];

SQL – Wildcard Operators

We already have discussed SQL **LIKE** operator, which is used to compare a value to similar values using wildcard operators.

SQL supports following two wildcard operators in conjunction with the LIKE operator:

Wildcards

Description

The percent sign (%)

Matches one or more characters. Note that MS Access uses the asterisk (*) wildcard character instead of the percent sign (%) wildcard character.

The underscore (_)

Matches one character. Note that MS Access uses a question mark (?) instead of the underscore (_) to match any one character.

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Syntax:

The basic syntax of '%' and '_' is as follows:

SELECT FROM table_name WHERE column LIKE 'XXXX%'

or

SELECT FROM table_name WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name WHERE column LIKE 'XXXX_'

or

SELECT FROM table_name WHERE column LIKE '_XXXX'

or

SELECT FROM table_name WHERE column LIKE '_XXXX_'

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '_' operators:

Statement

Description

WHERE SALARY LIKE

'200%'

Finds any values that start with 200

WHERE SALARY LIKE

'%200%'

Finds any values that have 200 in any position

WHERE SALARY LIKE

'_00%'

Finds any values that have 00 in the second and third positions

WHERE SALARY LIKE

'2_%_ %'

Finds any values that start with 2 and are at least 3 characters in length

WHERE SALARY LIKE

'%2'

Finds any values that end with 2

WHERE SALARY LIKE

'_2%3'

Finds any values that have a 2 in the second position and end with a 3

WHERE SALARY LIKE

'2____3'

Finds any values in a five-digit number that start with 2 and end with 3

Let us take a real 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, which would display all the records from CUSTOMERS table where SALARY starts with 200:

SQL> SELECT * FROM CUSTOMERS WHERE SALARY LIKE '200%';

This would produce the following result:

```
+---+-----+---+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
+---+-----+---+-----+-----+
```

SQL – Date Functions

Following is a list of all important Date and Time related functions available through SQL. There are various other functions supported by your RDBMS. Given list is based on MySQL RDBMS.

Name

Description

ADDDATE()

Adds dates

ADDTIME()

Adds time

CONVERT_TZ()

Converts from one timezone to another

CURDATE()

Returns the current date

CURRENT_DATE(),

CURRENT_DATE

Synonyms for CURDATE()

CURRENT_TIME(),

CURRENT_TIME

Synonyms for CURTIME()

CURRENT_TIMESTAMP(),

CURRENT_TIMESTAMP

Synonyms for NOW()

CURTIME()

Returns the current time

DATE_ADD()
Adds two dates
DATE_FORMAT()
Formats date as specified
DATE_SUB()
Subtracts two dates
DATE()
Extracts the date part of a date or datetime expression
DATEDIFF()
Subtracts two dates
DAY()
Synonym for DAYOFMONTH()
DAYNAME()
Returns the name of the weekday
DAYOFMONTH()
Returns the day of the month (1-31)
DAYOFWEEK()
Returns the weekday index of the argument
DAYOFYEAR()
Returns the day of the year (1-366)
EXTRACT
Extracts part of a date
FROM_DAYS()
Converts a day number to a date
FROM_UNIXTIME()
Formats date as a UNIX timestamp
HOUR()
Extracts the hour
LAST_DAY
Returns the last day of the month for the argument
LOCALTIME(), LOCALTIME
Synonym for NOW()
LOCALTIMESTAMP,
LOCALTIMESTAMP()
Synonym for NOW()
MAKEDATE()
Creates a date from the year and day of year
MAKETIME
MAKETIME()
MICROSECOND()
Returns the microseconds from argument
MINUTE()
Returns the minute from the argument
MONTH()
Return the month from the date passed
MONTHNAME()
Returns the name of the month
NOW()
Returns the current date and time
PERIOD_ADD()

Adds a period to a year-month
PERIOD_DIFF()
Returns the number of months between periods
QUARTER()
Returns the quarter from a date argument
SEC_TO_TIME()
Converts seconds to 'HH:MM:SS' format
SECOND()
Returns the second (0-59)
STR_TO_DATE()
Converts a string to a date
SUBDATE()
When invoked with three arguments a synonym for
DATE_SUB()
SUBTIME()
Subtracts times
SYSDATE()
Returns the time at which the function executes
TIME_FORMAT()
Formats as time
TIME_TO_SEC()
Returns the argument converted to seconds
TIME()
Extracts the time portion of the expression passed
TIMEDIFF()
Subtracts time
TIMESTAMP()
With a single argument, this function returns the date or
datetime expression. With two arguments, the sum of the
arguments
TIMESTAMPADD()
Adds an interval to a datetime expression
TIMESTAMPDIFF()
Subtracts an interval from a datetime expression
TO_DAYS()
Returns the date argument converted to days
UNIX_TIMESTAMP()
Returns a UNIX timestamp
UTC_DATE()
Returns the current UTC date
UTC_TIME()
Returns the current UTC time
UTC_TIMESTAMP()
Returns the current UTC date and time
WEEK()
Returns the week number
WEEKDAY()
Returns the weekday index
WEEKOFYEAR()
Returns the calendar week of the date (1-53)

YEAR()

Returns the year

YEARWEEK()

Returns the year and week

ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE_ADD(). The related function SUBDATE() is a synonym for DATE_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE_ADD().

```
mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);
```

```
+-----+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1998-02-02 |
+-----+
```

1 row in set (0.00 sec)

```
mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);
```

```
+-----+
| ADDDATE('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1998-02-02 |
+-----+
```

1 row in set (0.00 sec)

When invoked with the days form of the second argument, MySQL treats it as an integer number of days to be added to expr.

```
mysql> SELECT ADDDATE('1998-01-02', 31);
```

```
+-----+
| DATE_ADD('1998-01-02', INTERVAL 31 DAY) |
+-----+
| 1998-02-02 |
+-----+
```

1 row in set (0.00 sec)

SQL – Temporary Tables

There are RDBMS, which support temporary tables. Temporary Tables are a great feature that lets you store and process intermediate results by using the same selection, update, and join capabilities that you can use with typical SQL Server tables.

The temporary tables could be very useful in some cases to keep temporary data. The most important thing that should be known for temporary tables is that they will be deleted when the current client session terminates.

Temporary tables are available in MySQL version 3.23 onwards. If you use an older version of MySQL than 3.23, you can't use temporary tables, but you can use heap tables.

As stated earlier, temporary tables will only last as long as the session is alive. If you run the code in a PHP script, the temporary table will be destroyed automatically when the script finishes executing. If you are connected to the MySQL database server through the MySQL client program, then the temporary table will exist until you close the client or manually destroy the table.

Example:

Here is an example showing you usage of temporary table:

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
-> product_name VARCHAR(50) NOT NULL
-> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
-> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
-> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> INSERT INTO SALESSUMMARY
```

```
-> (product_name, total_sales, avg_unit_price, total_units_sold)
```

```
-> VALUES
-> ('cucumber', 100.25, 90, 2);
mysql> SELECT * FROM SALESSUMMARY;
+-----+-----+-----+-----+
| product_name | total_sales | avg_unit_price | total_units_sold |
+-----+-----+-----+-----+
| cucumber | 100.25 | 90.00 | 2 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

When you issue a **SHOW TABLES** command, then your temporary table would not be listed out in the list. Now if you will log out of the MySQL session and then you will issue a SELECT command, then you will find no data available in the database. Even your temporary table would also not exist.

Dropping Temporary Tables:

By default, all the temporary tables are deleted by MySQL when your database connection gets terminated. Still you want to delete them in between, then you do so by issuing DROP TABLE command.

Following is the example on dropping a temporary table.

```
mysql> CREATE TEMPORARY TABLE SALESSUMMARY (
-> product_name VARCHAR(50) NOT NULL
-> , total_sales DECIMAL(12,2) NOT NULL DEFAULT 0.00
-> , avg_unit_price DECIMAL(7,2) NOT NULL DEFAULT 0.00
-> , total_units_sold INT UNSIGNED NOT NULL DEFAULT 0
);
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO SALESSUMMARY
-> (product_name, total_sales, avg_unit_price, total_units_sold)
-> VALUES
-> ('cucumber', 100.25, 90, 2);
mysql> SELECT * FROM SALESSUMMARY;
+-----+-----+-----+-----+
| product_name | total_sales | avg_unit_price | total_units_sold |
+-----+-----+-----+-----+
| cucumber | 100.25 | 90.00 | 2 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
mysql> DROP TABLE SALESSUMMARY;
mysql> SELECT * FROM SALESSUMMARY;
ERROR 1146: Table 'TUTORIALS.SALESSUMMARY' doesn't exist
```

SQL – Clone Tables

There may be a situation when you need an exact copy of a table and CREATE TABLE ... SELECT... doesn't suit your purposes because the copy must include the same indexes, default values, and so forth.

If you are using MySQL RDBMS, you can handle this situation by the following steps:

- ☐ Use SHOW CREATE TABLE command to get a CREATE TABLE statement that specifies the source table's structure, indexes and all.
- ☐ Modify the statement to change the table name to that of the clone table and execute the statement. This way you will have exact clone table.
- ☐ Optionally, if you need the table contents copied as well, issue an INSERT INTO ... SELECT statement, too.

Example:

Try out the following example to create a clone table for **TUTORIALS_TBL** whose structure is as follows:

Step 1:

Get complete structure about table.

```
SQL> SHOW CREATE TABLE TUTORIALS_TBL \G;
***** 1. row *****
```

Table: TUTORIALS_TBL

```
Create Table: CREATE TABLE `TUTORIALS_TBL` (  
  `tutorial_id` int(11) NOT NULL auto_increment,  
  `tutorial_title` varchar(100) NOT NULL default "",  
  `tutorial_author` varchar(40) NOT NULL default "",  
  `submission_date` date default NULL,  
  PRIMARY KEY (`tutorial_id`),  
  UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)  
) TYPE=MyISAM  
1 row in set (0.00 sec)
```

Step 2:

Rename this table and create another table.

```
SQL> CREATE TABLE `CLONE_TBL` (  
-> `tutorial_id` int(11) NOT NULL auto_increment,  
-> `tutorial_title` varchar(100) NOT NULL default "",  
-> `tutorial_author` varchar(40) NOT NULL default "",  
-> `submission_date` date default NULL,  
-> PRIMARY KEY (`tutorial_id`),  
-> UNIQUE KEY `AUTHOR_INDEX` (`tutorial_author`)  
-> ) TYPE=MyISAM;  
Query OK, 0 rows affected (1.80 sec)
```

Step 3:

After executing step 2, you will clone a table in your database. If you want to copy data from old table, then you can do it by using INSERT INTO... SELECT statement.

```
SQL> INSERT INTO CLONE_TBL (tutorial_id,  
-> tutorial_title,  
-> tutorial_author,  
-> submission_date)  
-> SELECT tutorial_id,tutorial_title,  
-> tutorial_author,submission_date,  
-> FROM TUTORIALS_TBL;
```

Query OK, 3 rows affected (0.07 sec)

Records: 3 Duplicates: 0 Warnings: 0

Finally, you will have exact clone table as you wanted to have.

SQL – Sub Queries

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that subqueries must follow:

- ☐ Subqueries must be enclosed within parentheses.
- ☐ A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- ☐ An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- ☐ Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- ☐ The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- ☐ A subquery cannot be immediately enclosed in a set function.
- ☐ The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ] FROM table1 [, table2 ]
WHERE column_name OPERATOR (SELECT column_name [, column_name ]
FROM table1 [, table2 ][WHERE])
```

Example:

Consider the CUSTOMERS table having the following records:

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

Now, let us check following subquery with SELECT statement:

```
SQL> SELECT * FROM CUSTOMERS WHERE ID IN (SELECT ID FROM CUSTOMERS
WHERE SALARY > 4500);
```

This would produce the following result:

```
+---+-----+---+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+---+-----+---+-----+-----+
```

Subqueries with the INSERT Statement:

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ][ WHERE VALUE OPERATOR ]
```

Example:

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy complete CUSTOMERS table into CUSTOMERS_BKP, following is the syntax:

```
SQL> INSERT INTO CUSTOMERS_BKP
SELECT * FROM CUSTOMERS
WHERE ID IN (SELECT ID FROM CUSTOMERS);
```

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table SET column_name = new_value [ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME FROM TABLE_NAME) [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> UPDATE CUSTOMERS SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP WHERE AGE >= 27);
```

This would impact two rows and finally CUSTOMERS table would have the following records:

```
+---+-----+---+-----+-----+
| ID | NAME | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
```

1	Ramesh	35	Ahmedabad	125.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	2125.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Subqueries with the DELETE Statement:

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME [ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME FROM TABLE_NAME)[ WHERE ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example deletes records from CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> DELETE FROM CUSTOMERS WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP WHERE AGE > 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records:

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

SQL – Using Sequences

A sequence is a set of integers 1, 2, 3, ... that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value, and sequences provide an easy way to generate them.

This chapter describes how to use sequences in MySQL.

Using AUTO_INCREMENT column:

The simplest way in MySQL to use sequences is to define a column as AUTO_INCREMENT and leave rest of the things to MySQL to take care.

Example:

Try out the following example. This will create table and after that it will insert few rows in this table where it is not required to give record ID because its auto-incremented by MySQL.

```
mysql> CREATE TABLE INSECT
-> (
-> id INT UNSIGNED NOT NULL AUTO_INCREMENT,
-> PRIMARY KEY (id),
-> name VARCHAR(30) NOT NULL, # type of insect
-> date DATE NOT NULL, # date collected
-> origin VARCHAR(30) NOT NULL # where collected
);
Query OK, 0 rows affected (0.02 sec)
mysql> INSERT INTO INSECT (id,name,date,origin) VALUES
-> (NULL,'housefly','2001-09-10','kitchen'),
-> (NULL,'millipede','2001-09-10','driveway'),
-> (NULL,'grasshopper','2001-09-10','front yard');
Query OK, 3 rows affected (0.02 sec)
Records: 3 Duplicates: 0 Warnings: 0
mysql> SELECT * FROM INSECT ORDER BY id;
```

```
+---+-----+-----+-----+
| id | name | date | origin |
+---+-----+-----+-----+
| 1 | housefly | 2001-09-10 | kitchen |
| 2 | millipede | 2001-09-10 | driveway |
| 3 | grasshopper | 2001-09-10 | front yard |
+---+-----+-----+-----+
3 rows in set (0.00 sec)
```

Obtain AUTO_INCREMENT Values:

LAST_INSERT_ID() is a SQL function, so you can use it from within any client that understands how to issue SQL statements. Otherwise PERL and PHP scripts provide exclusive functions to retrieve auto-incremented value of last record.

PERL Example:

Use the mysql_insertid attribute to obtain the AUTO_INCREMENT value generated by a query. This attribute is accessed through either a database handle or a statement handle, depending on how you issue the query. The following example references it through the database handle:

```
$dbh->do ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')");
my $seq = $dbh->{mysql_insertid};
```

PHP Example:

After issuing a query that generates an AUTO_INCREMENT value, retrieve the value by calling mysql_insert_id():

```
mysql_query ("INSERT INTO INSECT (name,date,origin)
VALUES('moth','2001-09-14','windowsill')", $conn_id);
$seq = mysql_insert_id ($conn_id);
```

Renumbering an Existing Sequence:

There may be a case when you have deleted many records from a table and you want to resequence all the records. This can be done by using a simple trick but you should be very careful to do so if your table is having join, with other table.

If you determine that resequencing an AUTO_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again. The following example shows how to renumber the id values in the insect table using this technique:

```
mysql> ALTER TABLE INSECT DROP id;
mysql> ALTER TABLE insect
-> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
-> ADD PRIMARY KEY (id);
```

Starting a Sequence at a Particular Value:

By default, MySQL will start sequence from 1 but you can specify any other number as well at the time of table creation. Following is the example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE INSECT
-> (
-> id INT UNSIGNED NOT NULL AUTO_INCREMENT = 100,
-> PRIMARY KEY (id),
-> name VARCHAR(30) NOT NULL, # type of insect
-> date DATE NOT NULL, # date collected
-> origin VARCHAR(30) NOT NULL # where collected
);
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

```
mysql> ALTER TABLE t AUTO_INCREMENT = 100;
```

SQL – Handling Duplicates

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 unique records instead of fetching duplicate records.

The SQL **DISTINCT** keyword, which we already have discussed, is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

Syntax:

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

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

Example:

Consider the CUSTOMERS table having the following records:

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

First, let us see how the following SELECT query returns duplicate salary records:

```
SQL> SELECT SALARY FROM CUSTOMERS ORDER BY SALARY;
```

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

SALARY
1500.00
2000.00
2000.00
4500.00
6500.00
8500.00
10000.00

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry:

SALARY
1500.00
2000.00
4500.00
6500.00
8500.00
10000.00

SQL – Injection

If you take user input through a webpage and insert it into a SQL database, there's a chance that you have left yourself wide open for a security issue known as SQL Injection.

This lesson will teach you how to help prevent this from happening and help you secure your scripts and SQL statements in your server side scripts such as PERL Script.

Injection usually occurs when you ask a user for input, like their name, and instead of a name they give you a SQL statement that you will unknowingly run on your database.

Never trust user provided data, process this data only after validation; as a rule, this is done by pattern matching.

In the example below, the **name** is restricted to alphanumerical chars plus underscore and to a length between 8 and 20 chars (modify these rules as needed).

```
if (preg_match("/^\w{8,20}$/", $_GET['username'], $matches))  
{
```

```
$result = mysql_query("SELECT * FROM CUSTOMERS
WHERE name=$matches[0]");
}
else
{
echo "user name not accepted";
}
```

To demonstrate the problem, consider this excerpt:

```
// supposed input
$name = "Qadir"; DELETE FROM CUSTOMERS;";
mysql_query("SELECT * FROM CUSTOMERS WHERE name='{ $name}'");
```

The function call is supposed to retrieve a record from the CUSTOMERS table where the name column matches the name specified by the user. Under normal circumstances, \$name would only contain alphanumeric characters and perhaps spaces, such as the string ilia. But here, by appending an entirely new query to \$name, the call to the database turns into disaster: the injected DELETE query removes all records from CUSTOMERS.

Fortunately, if you use MySQL, the mysql_query() function does not permit query stacking or executing multiple SQL queries in a single function call. If you try to stack queries, the call fails. However, other PHP database extensions, such as SQLite and PostgreSQL, happily perform stacked queries, executing all of the queries provided in one string and creating a serious security problem

Preventing SQL Injection:

You can handle all escape characters smartly in scripting languages like PERL and PHP. The MySQL extension for PHP provides the function mysql_real_escape_string() to escape input characters that are special to MySQL.

```
if (get_magic_quotes_gpc())
{
$name = stripslashes($name);
}
$name = mysql_real_escape_string($name);
mysql_query("SELECT * FROM CUSTOMERS WHERE name='{ $name}'");
```

The LIKE Quandary:

To address the LIKE quandary, a custom escaping mechanism must convert user-supplied '%' and '_' characters to literals. Use addslashes(), a function that let's you specify a character range to escape.

```
$sub = addslashes(mysql_real_escape_string("%str"), "%_");
// $sub == \%str\_
mysql_query("SELECT * FROM messages
WHERE subject LIKE '{ $sub}%')");
```

SQL – Database Tuning

It takes time to become a Database Expert or an expert Database Administrator. This all comes with lot of experience in various database designs and good trainings.

But the following list may be helpful for the beginners to have a nice database performance –

- ☐ Use 3BNF database design explained in this tutorial in RDBMS Concepts chapter.
- ☐ Avoid number-to-character conversions because numbers and characters compare differently and lead to performance downgrade.
- ☐ While using SELECT statement, only fetch whatever information is required and avoid using * in your SELECT queries because it would load the system unnecessarily.
- ☐ Create your indexes carefully on all the tables where you have frequent search operations. Avoid index on the tables where you have less number of search operations and more number of insert and update operations.
- ☐ A full-table scan occurs when the columns in the WHERE clause do not have an index

associated with them. You can avoid a full-table scan by creating an index on columns that are used as conditions in the WHERE clause of an SQL statement.

- Be very careful of equality operators with real numbers and date/time values. Both of these can have small differences that are not obvious to the eye but that make an exact match impossible, thus preventing your queries from ever returning rows.
- Use pattern matching judiciously. LIKE COL% is a valid WHERE condition, reducing the returned set to only those records with data starting with the string COL. However, COL%Y does not further reduce the returned results set since %Y cannot be effectively evaluated. The effort to do the evaluation is too large to be considered. In this case, the COL% is used, but the %Y is thrown away. For the same reason, a leading wildcard %COL effectively prevents the entire filter from being used.
- Fine tune your SQL queries examining the structure of the queries (and subqueries), the SQL syntax, to discover whether you have designed your tables to support fast data manipulation and written the query in an optimum manner, allowing your DBMS to manipulate the data efficiently.
- For queries that are executed on a regular basis, try to use procedures. A procedure is a potentially large group of SQL statements. Procedures are compiled by the database engine and then executed. Unlike an SQL statement, the database engine need not optimize the procedure before it is executed.
- Avoid using the logical operator OR in a query if possible. OR inevitably slows down nearly any query against a table of substantial size.
- You can optimize bulk data loads by dropping indexes. Imagine the history table with many thousands of rows. That history table is also likely to have one or more indexes. When you think of an index, you normally think of faster table access, but in the case of batch loads, you can benefit by dropping the index(es).
- When performing batch transactions, perform COMMIT at after a fair number of records creation in stead of creating them after every record creation.
- Plan to defragment the database on a regular basis, even if doing so means developing a weekly routine.

Built-In Tuning Tools

Oracle has many tools for managing SQL statement performance but among them two are very popular. These two tools are –

- **Explain plan** – tool identifies the access path that will be taken when the SQL statement is executed.

□

tkprof – measures the performance by time elapsed during each phase of SQL statement processing.

If you want to simply measure the elapsed time of a query in Oracle, you can use the SQL*Plus command SET TIMING ON.

Check your RDBMS documentation for more detail on the above-mentioned tools and defragmenting the database.

PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.

□ Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

PL/SQL - Environment Setup

PL/SQL is not a stand-alone programming language; it is a tool within the Oracle programming environment. SQL* Plus is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt. These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen. To run PL/SQL programs, you should have Oracle RDBMS Server installed in your machine which will take care of executing SQL commands. Most recent version of Oracle RDBMS is 11g.

You will have to download either 32bit or 64 bit version of the installation as per your operating system. Usually there are two files, as I have downloaded for 64 bit Windows7. You will also use similar steps on your operating system, does not matter if it is Linux or Solaris.

□

win64_11gR2_database_1of2.zip

□

win64_11gR2_database_2of2.zip

After downloading above two files, you will need to unzip them in a single directory **database** and under that you will find following sub-directories:

Finally, click on **setup** file to start the installation and follow the given steps till the end. If everything has been done successfully then its time to verify your installation. At your command prompt use the following command if you are using Windows:

```
sqlplus "/ as sysdba"
```

If everything is fine, you should have SQL prompt where you will write your PL/SQL commands and scripts:

Text Editor

Running large programs from command prompt may land you in inadvertently losing some of the work. So a better option is to use command files. To use the command files:

- Type your code in a text editor, like Notepad, Notepad+, or EditPlus, etc.
- Save the file with the .sql extension in the home directory.
- Launch SQL*Plus command prompt from the directory where you created your PL/SQL file.
- Type @file_name at the SQL*Plus command prompt to execute your program.

If you are not using a file to execute PL/SQL scripts, then simply copy your PL/SQL code and then right click on the black window having SQL prompt and use **paste** option to paste complete code at the command prompt. Finally, just press enter to execute the code, if it is not already executed.

PL/SQL - Basic Syntax

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

S.N. Sections & Description

1

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 section is again optional and contains exception(s) that handle errors in the program.

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

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

The 'Hello World' Example:

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

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

```
Hello World
PL/SQL procedure successfully completed.
```

PL/SQL - Data Types

PL/SQL variables, constants and parameters must have a valid data type which specifies a storage format, constraints, and valid range of values. This tutorial will take you through **SCALAR** and **LOB** data types available in PL/SQL and other two data types will be covered in other chapters.

Category

Description

Scalar

Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.

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.

Composite

Data items that have internal components that can be accessed individually. For example, collections and records.

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:

Date Type

Description

Numeric

Numeric values on which arithmetic operations are performed.

Character

Alphanumeric values that represent single characters or strings of characters.

Boolean

Logical values on which logical operations are performed.

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 subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

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

Data Type

Description

PLS_INTEGER

Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits

BINARY_INTEGER

Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits

BINARY_FLOAT

Single-precision IEEE 754-format floating-point number

BINARY_DOUBLE

Double-precision IEEE 754-format floating-point number

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.

DEC(prec, scale)

ANSI specific fixed-point type with maximum precision of 38 decimal digits.

DECIMAL(prec, scale)

IBM specific fixed-point type with maximum precision of 38 decimal digits.

NUMERIC(pre, scale) Floating type with maximum precision of 38 decimal digits.

DOUBLE PRECISION ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

FLOAT

ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

INT

ANSI specific integer type with maximum precision of 38 decimal digits

INTEGER

ANSI and IBM specific integer type with maximum precision of 38 decimal digits

SMALLINT

ANSI and IBM specific integer type with maximum precision of 38 decimal digits

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:

Data Type

Description

CHAR

Fixed-length character string with maximum size of 32,767 bytes

VARCHAR2

Variable-length character string with maximum size of 32,767 bytes

RAW

Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL

NCHAR

Fixed-length national character string with maximum size of 32,767 bytes

NVARCHAR2

Variable-length national character string with maximum size of 32,767 bytes

LONG

Variable-length character string with maximum size of 32,760 bytes

LONG RAW

Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL

ROWID

Physical row identifier, the address of a row in an ordinary table

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.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in:

- ☐ SQL statements
- ☐ Built-in SQL functions (such as TO_CHAR)
- ☐ PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The **DATE** datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year, for example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field:

Field Name

Valid Datetime Values

Valid Interval Values

YEAR

-4712 to 9999 (excluding year 0)

Any nonzero integer

MONTH

01 to 12

0 to 11

DAY

01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)

Any nonzero integer

HOUR

00 to 23

0 to 23

MINUTE

00 to 59

0 to 59

SECOND

00 to 59.9(n), where 9(n) is the precision of time fractional seconds

0 to 59.9(n), where 9(n) is the precision of interval fractional seconds

TIMEZONE_HOUR

-12 to 14 (range accommodates daylight savings time changes)

Not applicable

TIMEZONE_MINUTE 00 to 59

Not applicable

TIMEZONE_REGION Found in the dynamic performance view

V\$TIMEZONE_NAMES

Not applicable

TIMEZONE_ABBR

Found in the dynamic performance view

V\$TIMEZONE_NAMES

Not applicable

PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data.

Following are the predefined PL/SQL LOB data types:

Data Type

Description

Size

BFILE

Used to store large binary objects in operating system files outside the database.

System-dependent. Cannot exceed 4 gigabytes (GB).

BLOB

Used to store large binary objects in the database.

8 to 128 terabytes (TB)

CLOB

Used to store large blocks of character data in the database.

8 to 128 TB

NCLOB

Used to store large blocks of NCHAR data in the database.

8 to 128 TB

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;
```

```
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype:

```
DECLARE
```

```
SUBTYPE name IS char(20);
```

```
SUBTYPE message IS varchar2(100);
```

```
salutation name;
```

```
greetings message;
```

```
BEGIN
```

```
salutation := 'Reader ';
```

```
greetings := 'Welcome to the World of PL/SQL';
```

```
dbms_output.put_line('Hello ' || salutation || greetings);
```

```
END;
```

```
/
```

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

```
Hello Reader Welcome to the World of PL/SQL
```

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

NULLs in PL/SQL

PL/SQL NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type. Note that NULL is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

PL/SQL - Variables

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.

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* must be a valid PL/SQL data type or any user defined data type which we already have discussed in 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:

```
counter binary_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, sometime program would produce unexpected result. 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.

```

PL/SQL - Constants and Literals

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Declaring a Constant

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed. For example:

```

PI CONSTANT NUMBER := 3.141592654;
DECLARE
-- constant declaration
pi constant number := 3.141592654;
-- other declarations
radius number(5,2);
dia number(5,2);
circumference number(7, 2);
area number (10, 2);
BEGIN
-- processing
radius := 9.5;
dia := radius * 2;
circumference := 2.0 * pi * radius;
area := pi * radius * radius;
-- output
dbms_output.put_line('Radius: ' || radius);
dbms_output.put_line('Diameter: ' || dia);
dbms_output.put_line('Circumference: ' || circumference);
dbms_output.put_line('Area: ' || area);
END;
/

```

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

```

Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53
PL/SQL procedure successfully completed.

```

The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals:

- ☐ Numeric Literals
- ☐ Character Literals
- ☐ String Literals
- ☐ BOOLEAN Literals
- ☐ Date and Time Literals

The following table provides examples from all these categories of literal values.

Literal Type

Example:

Numeric Literals

050 78 -14 0 +32767

6.6667 0.0 -12.0 3.14159 +7800.00

6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3

Character Literals

'A' '%' '9' ' ' 'z' '('

String Literals

'Hello, world!'

'Tutorials Point'

'19-NOV-12'

BOOLEAN Literals

TRUE, FALSE, and NULL.

Date and Time Literals

DATE '1978-12-25';

TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown below:

```
DECLARE
```

```
message varchar2(20):= 'That"s tutorialspoint.com!';
```

```
BEGIN
```

```
dbms_output.put_line(message);
```

```
END;
```

```
/
```

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

```
That's tutorialspoint.com!
```

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

PL/SQL - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following type of operators:

- ☐ Arithmetic operators
- ☐ Relational operators
- ☐ Comparison operators
- ☐ Logical operators
- ☐ String operators

This tutorial will explain the arithmetic, relational, comparison and logical operators one by one.

The String operators will be discussed under the chapter: **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5 then:

Operator

Description

Example

+

Adds two operands

A + B will give 15

-

Subtracts second operand from the first

A - B will give 5

*

Multiplies both operands

A * B will give 50

/

Divides numerator by de-numerator

A / B will give 2

**

Exponentiation operator, raises one operand to the power of other

A ** B will give 100000

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
-----------------	--------------------	----------------

=	Checks if the values of two operands are equal or not, if yes then condition becomes true. (A = B) is not true.	!=
<>		~=
~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. (A != B) is true.	>
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition (A > B) is not true. becomes true.	<
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. (A < B) is true.	>=
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. (A >= B) is not true.	<=
<=	Checks if the value of left operand is less than or	

equal to the value of right operand, if yes then condition becomes true.

(A <= B) is true.

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

Operator

Description

Example

LIKE

The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.

If 'Zara Ali' like 'Z% A_i'

returns a Boolean true, whereas,

'Nuha Ali' like 'Z% A_i' returns

a Boolean false.

BETWEEN

The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.

If x = 10 then, x between 5 and

20 returns true, x between 5 and

10 returns true, but x between

11 and 20 returns false.

IN

The IN operator tests set membership. x IN (set) means that x is equal to any member of set.

If x = 'm' then, x in ('a', 'b', 'c')

returns boolean false but x in

('m', 'n', 'o') returns Boolean

true.

IS NULL

The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.

If x = 'm', then 'x is null' returns

Boolean false.

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

Operator

Description

Example

and

Called logical AND operator. If both the operands are true then condition becomes true.

(A and B) is false.

or

Called logical OR Operator. If any of the two operands is true then condition becomes true.

(A or B) is true.

not

Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.

not (A and B) is true.

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with $3*2$ and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator

Operation

**

exponentiation

+, -

identity, negation

*, /

multiplication, division

+, -, ||

addition, subtraction, concatenation

=, <, >, <=, >=, <>, !=, ~=, ^=,

IS NULL, LIKE, BETWEEN, IN comparison

NOT

logical negation

AND

conjunction

OR

inclusion

PL/SQL - Conditions

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.

IF-THEN Statement

It is the simplest form of **IF** control 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.

Syntax:

Syntax for IF-THEN statement is:

IF condition THEN

S;

END IF;

Where *condition* is a Boolean or relational condition and S is a simple or compound statement.

Example of an IF-THEN statement is:

```
IF (a <= 20) THEN  
c:= c+1;  
END IF;
```

If the Boolean expression *condition* evaluates to true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

Flow Diagram:

IF-THEN-ELSE Statement

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

Syntax:

Syntax for the IF-THEN-ELSE statement is:

```
IF condition THEN  
S1;  
ELSE  
S2;  
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed. For example,

```
IF color = red THEN  
dbms_output.put_line('You have chosen a red car')  
ELSE  
dbms_output.put_line('Please choose a color for your car');  
END IF;
```

If the Boolean expression *condition* evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Flow Diagram:

IF-THEN-ELSIF Statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are few points to keep in mind.

- ☐ It's ELSIF, not ELSEIF
- ☐ An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- ☐ An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- ☐ Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax:

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

```
IF(boolean_expression 1)THEN  
S1; -- Executes when the boolean expression 1 is true  
ELSIF( boolean_expression 2) THEN  
S2; -- Executes when the boolean expression 2 is true  
ELSIF( boolean_expression 3) THEN  
S3; -- Executes when the boolean expression 3 is true  
ELSE  
S4; -- executes when the none of the above condition is true  
END IF;
```

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.

Syntax:

The syntax for case statement in PL/SQL is:

```
CASE selector
WHEN 'value1' THEN S1;
WHEN 'value2' THEN S2;
WHEN 'value3' THEN S3;
...
ELSE Sn; -- default case
END CASE;
```

Flow Diagram:**Searched CASE Statement**

The searched **CASE** statement has no selector and its **WHEN** clauses contain search conditions that give Boolean values.

Syntax:

The syntax for searched case statement in PL/SQL is:

```
CASE
WHEN selector = 'value1' THEN S1;
WHEN selector = 'value2' THEN S2;
WHEN selector = 'value3' THEN S3;
...
ELSE Sn; -- default case
END CASE;
```

Flow Diagram:**Nested IF-THEN-ELSE Statements**

It is always legal in PL/SQL programming to nest **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax:

```
IF( boolean_expression 1)THEN
-- executes when the boolean expression 1 is true
IF(boolean_expression 2) THEN
-- executes when the boolean expression 2 is true
sequence-of-statements;
END IF;
ELSE
-- executes when the boolean expression 1 is not true
else-statements;
END IF;
```

PL/SQL - Loops

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:

Basic Loop Statement

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax:

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP
Sequence of statements;
END LOOP;
```

Here, sequence of statement(s) may be a single statement or a block of statements. An **EXIT** statement or an **EXIT WHEN** statement is required to break the loop.

WHILE LOOP Statement

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

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

FOR LOOP Statement

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
FOR counter IN initial_value .. final_value LOOP
sequence_of_statements;
END LOOP;
```

Following are some special characteristics of PL/SQL for loop:

- The *initial_value* and *final_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`.

- The *initial_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.

- PL/SQL allows determine the loop range dynamically at run time.

Nested Loops

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows:

```
LOOP
Sequence of statements1
LOOP
Sequence of statements2
END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
sequence_of_statements1
FOR counter2 IN initial_value2 .. final_value2 LOOP
sequence_of_statements2
END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
sequence_of_statements1
WHILE condition2 LOOP
sequence_of_statements2
END LOOP;
END LOOP;
```

EXIT Statement

The **EXIT** statement in PL/SQL programming language has following two usages:

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a EXIT statement in PL/SQL is as follows:

```
EXIT;
```

Flow Diagram:**CONTINUE Statement**

The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax:

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

Flow Diagram:**GOTO Statement**

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

NOTE: Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;
```

```
..
```

```
..
```

```
<< label >>
```

```
statement;
```

Flow Diagram:**PL/SQL - Strings**

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings:

- ❑ **Fixed-length strings:** In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.

- ❑ **Variable-length strings:** In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.

- ❑ **Character large objects (CLOBs):** These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

'this isn't what it looks like'

Declaring String Variables

Oracle database provides numerous string datatypes, like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```
DECLARE
```

```
name varchar2(20);
```

```
company varchar2(30);
```

```
introduction clob;
```

```
choice char(1);
```

```
BEGIN
```

```

name := 'John Smith';
company := 'Infotech';
introduction := ' Hello! I'm John Smith from Infotech.';
choice := 'y';
IF choice = 'y' THEN
dbms_output.put_line(name);
dbms_output.put_line(company);
dbms_output.put_line(introduction);
END IF;
END;
/

```

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

```

John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.
PL/SQL procedure successfully completed

```

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

```

red_flag CHAR(1) := 'Y';
red_flag CHAR := 'Y';

```

PL/SQL - Arrays

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VRRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) OF <element_type>
```

Where,

□

varray_type_name is a valid attribute name,

□

n is the number of elements (maximum) in the varray,

□

element_type is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
Type created.
```

The basic syntax for creating a VRRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) OF <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

PL/SQL - Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.

PL/SQL provides two kinds of subprograms:

- **Functions:** these subprograms return a single value, mainly used to compute and return a value.
- **Procedures:** these subprograms do not return a value directly, mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure** and we will cover **PL/SQL function** in next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

S.N.

Parts & Description

1

Declarative Part

It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

2

Executable Part

This is a mandatory part and contains statements that perform the designated action.

3

Exception-handling

This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
```

```
BEGIN
```

```
< procedure_body >
```

```
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN

represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

PL/SQL - Functions

A PL/SQL function is same as a procedure except that it returns a value.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
< function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
-
- RETURN clause specifies that data type you are going to return from the function.
- *function-body* contains the executable part.
- *function-body* must contain a **RETURN statement**.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

PL/SQL - Cursors

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, 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 the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement.

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 involves four steps:

- ❑ Declaring the cursor for initializing in the memory
- ❑ Opening the cursor for allocating memory
- ❑ Fetching the cursor for retrieving data
- ❑ Closing the cursor to release 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 memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open 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 above opened cursor as follows:

```
CLOSE c_customers;
```

Example:

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

```
DECLARE  
c_id customers.id%type;  
c_name customers.name%type;  
c_addr customers.address%type;  
CURSOR c_customers is  
SELECT id, name, address FROM customers;  
BEGIN  
OPEN c_customers;  
LOOP  
FETCH c_customers into c_id, c_name, c_addr;  
dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);  
EXIT WHEN c_customers%notfound;  
END LOOP;  
CLOSE c_customers;  
END;  
/
```

When the above code is executed at 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.
```

PL/SQL - Records

A PL/SQL **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

PL/SQL can handle the following types of records:

- ☐ Table-based
- ☐ Cursor-based records
- ☐ User-defined records

Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursor-based** records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
customer_rec customers%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;
/
```

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

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000
PL/SQL procedure successfully completed.
```

Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
CURSOR customer_cur is
SELECT id, name, address
FROM customers;
customer_rec customer_cur%rowtype;
BEGIN
OPEN customer_cur;
LOOP
FETCH customer_cur into customer_rec;
EXIT WHEN customer_cur%notfound;
DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
END LOOP;
END;
/
```

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

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal
PL/SQL procedure successfully completed.
```

User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields. Suppose you want to keep track of your books in a library.

You might want to track the following attributes about each book:

- ☐ Title
- ☐ Author

- Subject
- Book ID

Defining a Record

The record type is defined as:

```
TYPE
type_name IS RECORD
( field_name1 datatype1 [NOT NULL] [:= DEFAULT EXPRESSION],
  field_name2 datatype2 [NOT NULL] [:= DEFAULT EXPRESSION],
  ...
  field_nameN datatypeN [NOT NULL] [:= DEFAULT EXPRESSION]);
record-name type_name;
```

Here is the way you would declare the Book record:

```
DECLARE
TYPE books IS RECORD
(title varchar(50),
 author varchar(50),
 subject varchar(100),
 book_id number);
book1 books;
book2 books;
```

Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of record:

```
DECLARE
type books is record
(title varchar(50),
 author varchar(50),
 subject varchar(100),
 book_id number);
book1 books;
book2 books;
BEGIN
-- Book 1 specification
book1.title := 'C Programming';
book1.author := 'Nuha Ali ';
book1.subject := 'C Programming Tutorial';
book1.book_id := 6495407;
-- Book 2 specification
book2.title := 'Telecom Billing';
book2.author := 'Zara Ali';
book2.subject := 'Telecom Billing Tutorial';
book2.book_id := 6495700;
-- Print book 1 record
dbms_output.put_line('Book 1 title : ' || book1.title);
dbms_output.put_line('Book 1 author : ' || book1.author);
dbms_output.put_line('Book 1 subject : ' || book1.subject);
dbms_output.put_line('Book 1 book_id : ' || book1.book_id);
-- Print book 2 record
dbms_output.put_line('Book 2 title : ' || book2.title);
dbms_output.put_line('Book 2 author : ' || book2.author);
dbms_output.put_line('Book 2 subject : ' || book2.subject);
dbms_output.put_line('Book 2 book_id : ' || book2.book_id);
END;
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
```

Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
PL/SQL procedure successfully completed.

PL/SQL - Exceptions

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here, you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others*

THEN:

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling goes here >
WHEN exception1 THEN
exception1-handling-statements
WHEN exception2 THEN
exception2-handling-statements
WHEN exception3 THEN
exception3-handling-statements
.....
WHEN others THEN
exception3-handling-statements
END;
```

Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
c_id customers.id%type := 8;
c_name customers.name%type;
c_addr customers.address%type;
BEGIN
SELECT name, address INTO c_name, c_addr
FROM customers
WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
WHEN no_data_found THEN
dbms_output.put_line('No such customer!');
WHEN others THEN
dbms_output.put_line('Error!');
END;
/
```

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

No such customer!

PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND** which is captured in **EXCEPTION** block.

PL/SQL - Triggers

Triggers are stored programs, which are automatically executed or fired when some events

occur. Triggers are, in fact, written to be executed in response to any of the following events:

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

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

Creating Triggers

The syntax for creating a trigger is:

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

Where,

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

Example:

To start with, we will be using the CUSTOMERS table:

Select * from customers;

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

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

```
CREATE OR REPLACE TRIGGER display_salary_changes
```

```

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

```

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

Trigger created.

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

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

PL/SQL - Packages

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- ❑ Package specification
- ❑ Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```

CREATE PACKAGE cust_sal AS
PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/

```

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

Package created.

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal* package created above.

```

CREATE OR REPLACE PACKAGE BODY cust_sal AS
PROCEDURE find_sal(c_id customers.id%TYPE) IS
c_sal customers.salary%TYPE;
BEGIN

```

```

SELECT salary INTO c_sal
FROM customers
WHERE id = c_id;
dbms_output.put_line('Salary: '|| c_sal);
END find_sal;
END cust_sal;
/

```

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

Package body created.

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema, the following program uses the *find_sal* method of the *cust_sal* package:

```

DECLARE
code customers.id%type := &cc_id;
BEGIN
cust_sal.find_sal(code);
END;
/

```

When the above code is executed at SQL prompt, it prompt to enter customer ID, and when you enter an ID, it displays the corresponding salary as follows:

Enter value for cc_id: 1

Salary: 3000

PL/SQL procedure successfully completed.

PL/SQL - Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- ☐ Index-by tables or Associative array
- ☐ Nested table
- ☐ Variable-size array or Varray

Index-By Table

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique, and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here we are creating an index-by table named **table_name** whose keys will be of *subscript_type* and associated values will be of *element_type*

```

TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;
table_name type_name;

```

Example:

Following example how to create a table to store integer values along with names and later it prints the same list of names.

```

DECLARE
TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
salary_list salary;
name VARCHAR2(20);
BEGIN
-- adding elements to the table
salary_list('Rajnish') := 62000;
salary_list('Minakshi') := 75000;
salary_list('Martin') := 100000;
salary_list('James') := 78000;
-- printing the table
name := salary_list.FIRST;

```

```

WHILE name IS NOT null LOOP
dbms_output.put_line
('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
name := salary_list.NEXT(name);
END LOOP;
END;
/

```

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

```

Salary of Rajnish is 62000
Salary of Minakshi is 75000
Salary of Martin is 100000
Salary of James is 78000
PL/SQL procedure successfully completed.

```

Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

An **nested table** is created using the following syntax:

```

TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name type_name;

```

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause. A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example:

The following examples illustrate the use of nested table:

```

DECLARE
TYPE names_table IS TABLE OF VARCHAR2(10);
TYPE grades IS TABLE OF INTEGER;
names names_table;
marks grades;
total integer;
BEGIN
names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
marks:= grades(98, 97, 78, 87, 92);
total := names.count;
dbms_output.put_line('Total '|| total || ' Students');
FOR i IN 1 .. total LOOP
dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
end loop;
END;
/

```

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

```

Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92
PL/SQL procedure successfully completed.

```

PL/SQL - Transactions

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting an Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place:

- ☐ The first SQL statement is performed after connecting to the database.
- ☐ At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place:

- ☐ A COMMIT or a ROLLBACK statement is issued.
- ☐ A DDL statement, like CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- ☐ A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- ☐ User disconnects from the database.
- ☐ User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
- ☐ SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
- ☐ A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is:

COMMIT;

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

ROLLBACK [TO SAVEPOINT < savepoint_name>];

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes:

ROLLBACK;

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is:

SAVEPOINT < savepoint_name >;

Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as:

SET AUTOCOMMIT ON;

You can turn-off auto commit mode using the following command:

SET AUTOCOMMIT OFF;

PL/SQL - Date & Time

PL/SQL provides two classes of date and time related data types:

- ☐ Datetime data types
- ☐ Interval data types

The Datetime data types are:

- ☐ DATE

- **TIMESTAMP**
- **TIMESTAMP WITH TIME ZONE**
- **TIMESTAMP WITH LOCAL TIME ZONE**

The Interval data types are:

- **INTERVAL YEAR TO MONTH**
- **INTERVAL DAY TO SECOND**

The Datetime Data Types

Following are the Datetime data types:

□ **DATE** - it stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as:

□

TIMESTAMP - it is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.

□

TIMESTAMP WITH TIME ZONE - it is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This datatype is useful for collecting and evaluating date information across geographic regions.

□

TIMESTAMP WITH LOCAL TIME ZONE - it is another variant of TIMESTAMP that includes a time zone offset in its value.

Examples:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
08/31/2012 5:25:34 PM
```

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS') FROM DUAL;
```

Output:

```
31-08-2012 05:26:14
```

```
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output:

```
01/31/2013 5:26:31 PM
```

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output:

```
8/31/2012 5:26:55.347000 PM
```

The Interval Data Types

Following are the Interval data types:

- **INTERVAL YEAR TO MONTH** - it stores a period of time using the YEAR and MONTH datetime fields.
- **INTERVAL DAY TO SECOND** - it stores a period of time in terms of days, hours, minutes, and seconds.

PL/SQL - DBMS Output

The **DBMS_OUTPUT** is a built-in package that enables you to display output, display debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.

Example:

```
DECLARE
lines dbms_output.chararr;
num_lines number;
BEGIN
-- enable the buffer with default size 20000
dbms_output.enable;
dbms_output.put_line('Hello Reader!');
```

```

dbms_output.put_line('Hope you have enjoyed the tutorials!');
dbms_output.put_line('Have a great time exploring pl/sql!');
num_lines := 3;
dbms_output.get_lines(lines, num_lines);
FOR i IN 1..num_lines LOOP
dbms_output.put_line(lines(i));
END LOOP;
END;
/

```

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

```

Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!
PL/SQL procedure successfully completed.

```

PL/SQL - Object Oriented

PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An object type allows you to create composite types. Using objects allow you implementing real world objects with specific structure of data and methods for operating it. Objects have attributes and methods. Attributes are properties of an object and are used for storing an object's state; and methods are used for modeling its behaviors.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Below is an example to create a simple **address** object consisting of few attributes:

```

CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
street varchar2(30),
city varchar2(20),
state varchar2(10),
pincode varchar2(10)
);
/

```

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

```
Type created.
```

Let's create one more object **customer** where we will wrap **attributes** and **methods** together to have object oriented feeling:

```

CREATE OR REPLACE TYPE customer AS OBJECT
(code number(5),
name varchar2(30),
contact_no varchar2(12),
addr address,
member procedure display
);
/

```

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

```
Type created.
```

Instantiating an Object

Defining an object type provides a blueprint for the object. To use this object, you need to create instances of this object. You can access the attributes and methods of the object using the instance name and **the access operator (.)** as follows:

```

DECLARE
residence address;
BEGIN
residence := address('103A', 'M.G.Road', 'Jaipur', 'Rajasthan', '201301');
dbms_output.put_line('House No: '|| residence.house_no);
dbms_output.put_line('Street: '|| residence.street);
dbms_output.put_line('City: '|| residence.city);
dbms_output.put_line('State: '|| residence.state);
dbms_output.put_line('Pincode: '|| residence.pincode);
END;

```

/

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

House No: 103A

Street: M.G.Road

City: Jaipur

State: Rajasthan

Pincode: 201301

PL/SQL procedure successfully completed.

Inheritance for PL/SQL Objects:

PL/SQL allows creating object from existing base objects. To implement inheritance, the base objects should be declared as NOT FINAL. The default is FINAL.

The following programs illustrate inheritance in PL/SQL Objects. Let us create another object named TableTop which is inheriting from the Rectangle object. Creating the base *rectangle* object:

```
CREATE OR REPLACE TYPE rectangle AS OBJECT
```

```
(length number,
```

```
width number,
```

```
member function enlarge( inc number) return rectangle,
```

```
NOT FINAL member procedure display) NOT FINAL
```

/

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

Type created.

Creating the base type body:

```
CREATE OR REPLACE TYPE BODY rectangle AS
```

```
MEMBER FUNCTION enlarge(inc number) return rectangle IS
```

```
BEGIN
```

```
return rectangle(self.length + inc, self.width + inc);
```

```
END enlarge;
```

```
MEMBER PROCEDURE display IS
```

```
BEGIN
```

```
dbms_output.put_line('Length: '|| length);
```

```
dbms_output.put_line('Width: '|| width);
```

```
END display;
```

```
END;
```

/

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

Type body created.

Creating the child object *tabletop*:

```
CREATE OR REPLACE TYPE tabletop UNDER rectangle
```

```
(
```

```
material varchar2(20),
```

```
OVERRIDING member procedure display
```

```
)
```

/

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

Type created.

Creating the type body for the child object *tabletop*:

```
CREATE OR REPLACE TYPE BODY tabletop AS
```

```
OVERRIDING MEMBER PROCEDURE display IS
```

```
BEGIN
```

```
dbms_output.put_line('Length: '|| length);
```

```
dbms_output.put_line('Width: '|| width);
```

```
dbms_output.put_line('Material: '|| material);
```

```
END display;
```

/

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

Type body created.

Using the *tabletop* object and its member functions:

```
DECLARE
```

```

t1 tabletop;
t2 tabletop;
BEGIN
t1:= tabletop(20, 10, 'Wood');
t2 := tabletop(50, 30, 'Steel');
t1.display;
t2.display;
END;
/

```

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

```

Length: 20
Width: 10
Material: Wood
Length: 50
Width: 30
Material: Steel
PL/SQL procedure successfully completed.

```

Abstract Objects in PL/SQL

The NOT INSTANTIABLE clause allows you to declare an abstract object. You cannot use an abstract object as it is; you will have to create a subtype or child type of such objects to use its functionalities.

For example,

```

CREATE OR REPLACE TYPE rectangle AS OBJECT
(length number,
width number,
NOT INSTANTIABLE NOT FINAL MEMBER PROCEDURE display)
NOT INSTANTIABLE NOT FINAL
/

```

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

Type created.