

GURU NANAK DEV ENGINEERING COLLEGE

PRACTICAL FILE

Relational Database Management System Laboratory (PGCA-2208)



Submitted To:

Prof. Satinderpal Singh

Assistant Professor

Submitted By:

Ajaypal Singh Mahal

University Roll No: 2303281

(MCA 1st Semester)

Batch: 2023-2025

Department of Computer Application

INDEX:

Sr.No.	List of experiments	Pages	Date	Remarks
1	Comparative study of various Database Management System	1-2	05/10/2023	
2	Data Definition Language(DDL),Data Manipulation Language(DML) and Data Control Language(DCL):	3-4	05/10/2023	
3	How to apply constraints at various levels	5-7	06/10/2023	
4	View data in the required form using Operators, Functions and Joins	8-15	06/10/2023	
5	Creating different types of views for tailored presentation of data	16-18	12/10/2023	
6	How to apply Conditional Controls in PL/SQL	19-20	12/10/2023	
7	Error handling using Internal Exceptions and External Exceptions	21-23	13/10/2023	
8	Using various types of cursors	24-26	13/10/2023	
9	How to run Stored Procedures and Functions	27-30	19/10/2023	
10	Creating Packages and applying Triggers	31-36	19/10/2023	
11	Creating Arrays And Nested Tables	37-39	20/10/2023	

Experiment-1:

Comparative study of various Database Management System:

A collection of information which is managed such that it can be updated and easily accessed is called a database. A software package which can be used to manipulate, validate and retrieve this database is called a Database Management System.

For example, Airlines use this software package to book tickets and confirm reservations which are then managed to keep a track of the schedule.

There are majorly four types of database:

Network Database: When the details of multiple members can be linked to the files of multiple owners and vice versa, it is called a network database.

Hierarchical Database: When the data stored in the form of records and is connected to each other through links is called hierarchical database. Each record comprises fields and each field comprises only one value.

Relational Database: When the data is organised as a set of tables comprising rows and columns with a pre-defined relationship with one another, it is called a relational database.

Object-oriented Database – the information is represented as objects, with different types of relationships possible between two or more objects. Such databases use an object-oriented programming language for development.

Components of Database Management Software

There four main components on which the working of a DBMS depends. This includes:

Data: The main component is the data. The entire database is set based on the data and the information processed based on it. This data acts as a bridge between the software and hardware components of DBMS. This can further be divided into three varieties:

User Data – The actual data based on which the work is done

Metadata – This is the data of the data, i.e., managing the data required to enter the information

Application MetaData – This is the structure and format of the queries

To simplify it, in a table, the information given in each table is the User Data, the number of tables, rows and columns is the MetaData the structure we choose is the Application MetaData.

Hardware: These are the general hardware devices which help us save and enter the data like hard disks, magnetic tapes, etc.

Software: The software acts as a medium of communication between the user and the database. Based on the user's requirement, the database can be modified and updated. To perform operations on the data, query languages like SQL are used.

Users: No function can be performed without the Users. Thus, they form the fourth most important component of DBMS. The information entered into a database is used by the User or the administrator to perform their business operations and responsibilities.

Benefits of Using Database Management System

Major Organisations and Banking firms choose to work using the Database Management System. It is because this system program helps the user and the administrator easily manage the data and information on the database.

Given below are a few advantages of using a Database Management System:

Securing the data is easy. The administrator can restrict the usage of the database to a few people only. Thus, access to the database is restricted only to the administering team of an Organisation or Business which can keep the data safe

A single file can manage the entire database which is why duplicity and redundancy cannot happen. This makes the data more consistent and easy to update

Since tables can be created in DBMS, interrelating the data and elements with each other is convenient

The backup and recovery of data is managed by the software, which ensures that the database is secure at all times

It allows multi-users to operate the database at a single time, making it more efficient

Important Terms related to Database Management System

A few other important terms related to DBMS have been discussed in brief below. To understand this concept better, one must be aware of the following terms.

Data Manipulation Languages (DML) – This is a programming language used to insert or modify the data present in a database. These are of two types: SQL and DDL.

Structured Query Language (SQL) – A programming language generally used for the relational database management system, which comprises tables.

Data Definition Language (DDL) – It is a syntax which helps in modifying data present in the form of tables or indexes

Primary Key – Each file has a unique key. Using the Primary Key, a specific file can be identified

Foreign Key – The relation between a field in one table and component identified by a primary key can be detected using a Foreign Key

Experiment-2:

Data Definition Language(DDL),Data Manipulation Language(DML) and Data Control Language(DCL):

Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like CREATE, DROP, INSERT, etc. to carry out the required tasks.

SQL commands are like instructions to a table. It is used to interact with the database with some operations. It is also used to perform specific tasks, functions, and queries of data. SQL can perform various tasks like creating a table, adding data to tables, dropping the table, modifying the table, set permission for users.

DDL (Data Definition Language)

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database. DDL is a set of SQL commands used to create, modify, and delete database structures but not data. These commands are normally not used by a general user, who should be accessing the database via an application.

List of DDL commands:

CREATE: This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).

DROP: This command is used to delete objects from the database.

ALTER: This is used to alter the structure of the database.

TRUNCATE: This is used to remove all records from a table, including all spaces allocated for the records are removed.

COMMENT: This is used to add comments to the data dictionary.

RENAME: This is used to rename an object existing in the database.

```
1 CREATE TABLE public.customer_details
2 (
3     customer_id character varying NOT NULL,
4     customer_name character varying(255) NOT NULL,
5     location character varying(255) NOT NULL,
6     amount_spent numeric NOT NULL,
7     order_id character varying NOT NULL
8 );
```

Data Output	Messages	Notifications	Explain
CREATE TABLE			
Query returned successfully in 210 msec.			

DML(Data Manipulation Language)

The SQL commands that deal with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements. It is the component of the SQL statement that controls access to data and to the database. Basically, DCL statements are grouped with DML statements.

List of DML commands:

INSERT: It is used to insert data into a table.

UPDATE: It is used to update existing data within a table.

DELETE: It is used to delete records from a database table.

```
1 INSERT INTO public.customers(  
2     customer_id, sale_date, sale_amount, salesperson, store_state, order_id)  
3     VALUES (1005,'2019-12-12',4200,'R K Rakesh','MH','1007');
```

Data Output Messages Notifications Explain

INSERT 0 1

Query returned successfully in 93 msec.

DCL (Data Control Language)

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

GRANT: This command gives users access privileges to the database.

Syntax:

GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

REVOKE: This command withdraws the user's access privileges given by using the GRANT command.

Syntax:

REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2.

```
2 --CREATE TABLE demo(id INT IDENTITY(1, 1),EmpName varchar(50));  
3 SELECT IDENT_CURRENT('demo') as [IdentityBeforeInsert];  
4 BEGIN TRANSACTION  
5 INSERT INTO dbo_demo ( EmpName ) VALUES('Raj');  
6 INSERT INTO dbo_demo ( EmpName ) VALUES('Sonu');  
7 INSERT INTO dbo_demo ( EmpName ) VALUES('Hari');  
8 INSERT INTO dbo_demo ( EmpName ) VALUES('Cheena');  
9 SELECT IDENT_CURRENT('demo') as [IdentityAfterInsert];  
10 ROLLBACK TRAN  
11 SELECT IDENT_CURRENT('demo') as [IdentityAfterRollbackInsert];
```

150 %

Results Messages

	IdentityBeforeInsert
1	1

	IdentityAfterInsert
1	4

	IdentityAfterRollbackInsert
1	4

Experiment-3:

How to apply constraints at various levels:

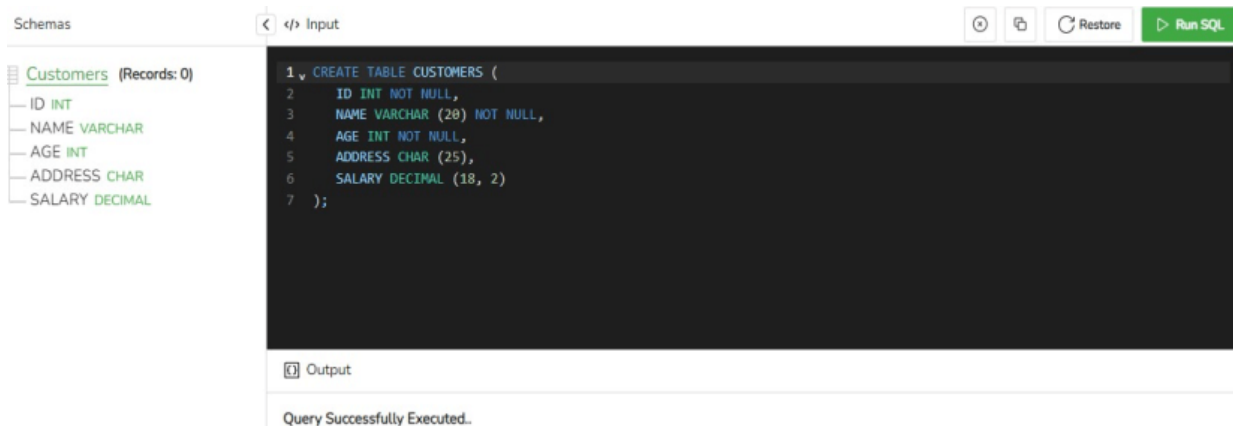
SQL Constraints

SQL Constraints are the rules applied to a data columns or the complete table to limit the type of data that can go into a table. When you try to perform any INSERT, UPDATE, or DELETE operation on the table, RDBMS will check whether that data violates any existing constraints and if there is any violation between the defined constraint and the data action, it aborts the operation and returns an error.

We can define a column level or a table level constraints. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

SQL Create Constraints

We can create constraints on a table at the time of a table creation using the CREATE TABLE statement, or after the table is created, we can use the ALTER TABLE statement to create or delete table constraints.



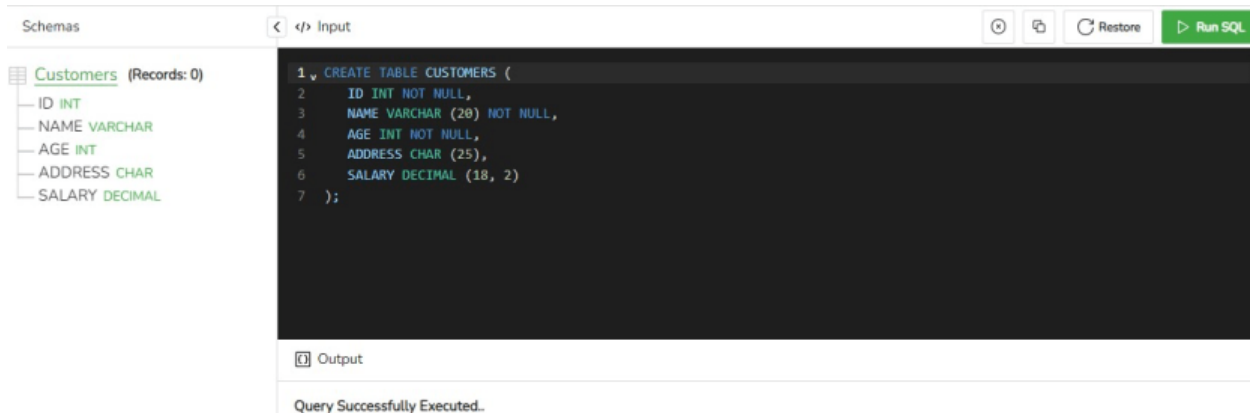
The screenshot shows a database IDE interface. On the left, a 'Schemas' pane lists a 'Customers' table with 0 records and its columns: ID (INT), NAME (VARCHAR), AGE (INT), ADDRESS (CHAR), and SALARY (DECIMAL). The main editor displays the following SQL code:

```
1 CREATE TABLE CUSTOMERS (  
2   ID INT NOT NULL,  
3   NAME VARCHAR (20) NOT NULL,  
4   AGE INT NOT NULL,  
5   ADDRESS CHAR (25),  
6   SALARY DECIMAL (18, 2)  
7 );
```

Below the editor, the 'Output' pane shows the message 'Query Successfully Executed..'. The top right of the IDE contains buttons for 'Run SQL' and 'Restore'.

NOT NULL Constraint

When applied to a column, NOT NULL constraint ensure that a column cannot have a NULL value. Following is the example to create a NOT NULL constraint:



This screenshot is identical to the one above, showing the same SQL IDE interface with the 'Customers' table schema on the left, the same CREATE TABLE SQL code in the main editor, and the 'Query Successfully Executed..' message in the output pane.

UNIQUE Key Constraint

When applied to a column, UNIQUE Key constraint ensure that a column accepts only UNIQUE values. Following is the example to create a UNIQUE Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only UNIQUE values.



The screenshot shows a database IDE interface. On the left, a 'Schemas' pane displays a tree view for a 'Customers' table with columns: ID INT, NAME VARCHAR, AGE INT, ADDRESS CHAR, and SALARY DECIMAL. The main editor area contains the following SQL code:

```
1 CREATE TABLE CUSTOMERS (  
2   ID INT NOT NULL UNIQUE,  
3   NAME VARCHAR (20) NOT NULL,  
4   AGE INT NOT NULL,  
5   ADDRESS CHAR (25),  
6   SALARY DECIMAL (18, 2)  
7 );
```

Below the code editor, the 'Output' pane shows the message: 'Query Successfully Executed..'

DEFAULT Value Constraint

When applied to a column, DEFAULT Value constraint provides a default value for a column when none is specified. Following is the example to create a DEFAULT constraint on column NAME. Once this constraint is created, column NAME will set to "Not Available" value if NAME is not set to a value.



The screenshot shows a database IDE interface. On the left, a 'Schemas' pane displays a tree view for a 'Customers' table with columns: ID INT, NAME VARCHAR, AGE INT, ADDRESS CHAR, and SALARY DECIMAL. The main editor area contains the following SQL code:

```
1 CREATE TABLE CUSTOMERS (  
2   ID INT NOT NULL UNIQUE,  
3   NAME VARCHAR (20) DEFAULT 'Not Available',  
4   AGE INT NOT NULL,  
5   ADDRESS CHAR (25),  
6   SALARY DECIMAL (18, 2)  
7 );
```

Below the code editor, the 'Output' pane shows the message: 'Query Successfully Executed..'

PRIMARY Key Constraint

When applied to a column, PRIMARY Key constraint ensure that a column accepts only UNIQUE value and there can be a single PRIMARY Key on a table but multiple columns can constitute a PRIMARY Key. Following is the example to create a PRIMARY Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only unique values.



The screenshot shows a database IDE interface. On the left, a 'Schemas' pane displays a tree view for a 'Customers' table with columns: ID INT, NAME VARCHAR, AGE INT, ADDRESS CHAR, and SALARY DECIMAL. The main editor area contains the following SQL code:

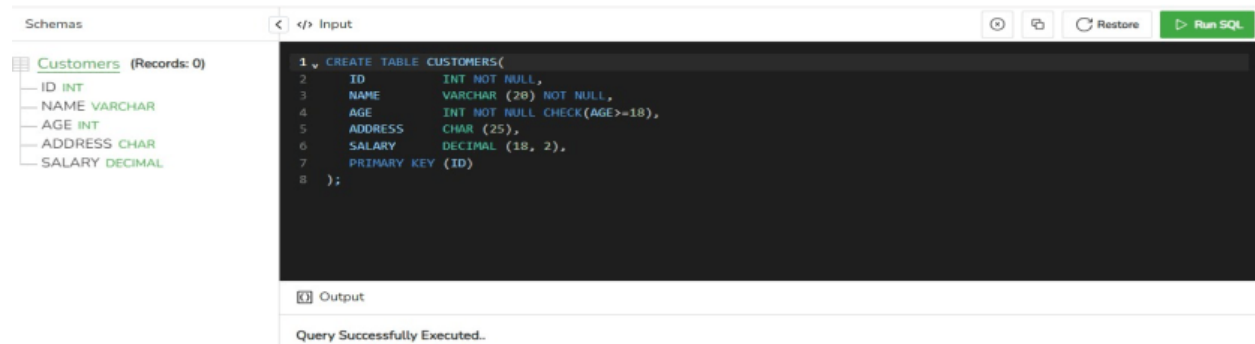
```
1 CREATE TABLE CUSTOMERS(  
2   ID          INT NOT NULL,  
3   NAME        VARCHAR (20) NOT NULL,  
4   AGE         INT NOT NULL,  
5   ADDRESS     CHAR (25),  
6   SALARY      DECIMAL (18, 2),  
7   PRIMARY KEY (ID)  
8 );
```

Below the code editor, the 'Output' pane shows the message: 'Query Successfully Executed..'

CHECK Value Constraint

When applied to a column, CHECK Value constraint works like a validation and it is used to check the validity of the data entered into the particular column of the table. table and uniquely identifies a row/record in that table.

Following is an example to create a CHECK validation on AGE column which will not accept if its value is below to 18.



The screenshot shows a SQL IDE interface. On the left, a 'Schemas' pane displays a tree view with a 'Customers' table (Records: 0) containing columns: ID INT, NAME VARCHAR, AGE INT, ADDRESS CHAR, and SALARY DECIMAL. The main editor area shows the following SQL code:

```
1 CREATE TABLE CUSTOMERS(  
2     ID          INT NOT NULL,  
3     NAME        VARCHAR (28) NOT NULL,  
4     AGE         INT NOT NULL CHECK(AGE>=18),  
5     ADDRESS     CHAR (25),  
6     SALARY      DECIMAL (18, 2),  
7     PRIMARY KEY (ID)  
8 );
```

Below the editor, the 'Output' pane shows the message: 'Query Successfully Executed.'

Experiment-4:

View data in the required form using Operators, Functions and Joins:

Structured Query Language is a computer language that we use to interact with a relational database. In this article we will see all types of SQL operators.

In simple operator can be defined as an entity used to perform operations in a table.

Operators are the foundation of any programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. SQL operators have three different categories.

Types of SQL Operators

Arithmetic operator

Comparison operator

Logical operator

Arithmetic Operators

We can use various arithmetic operators on the data stored in the tables. Arithmetic Operators are:

OperatorDescription

+

The addition is used to perform an addition operation on the data values.

–

This operator is used for the subtraction of the data values.

/

This operator works with the ‘ALL’ keyword and it calculates division operations.

*

This operator is used for multiplying data values.

%

Modulus is used to get the remainder when data is divided by another.

Example Query:

```
SELECT * FROM employee WHERE emp_city NOT LIKE 'A%';
```

	emp_id	emp_name	emp_city	emp_country
1	101	Utkarsh Tripathi	Varanasi	India
2	102	Abhinav Singh	Varanasi	India
3	103	Utkarsh Raghuvanshi	Varanasi	India
4	106	Ashutosh Kumar	Patna	India

Comparison Operators

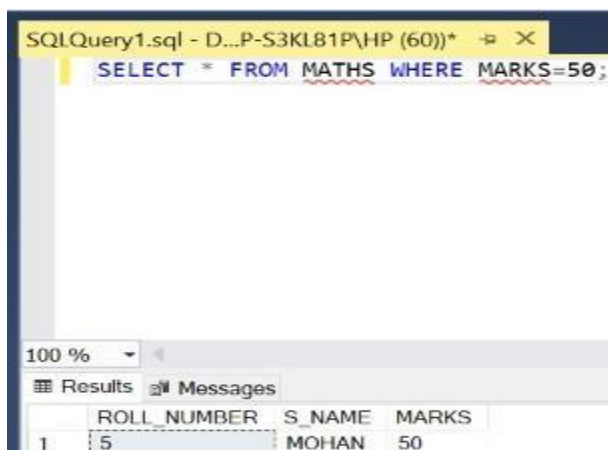
Another important operator in SQL is a comparison operator, which is used to compare one expression's value to other expressions. SQL supports different types of comparison operator, which is described below:

OperatorDescription

- = Equal to.
- > Greater than.
- < Less than.
- >= Greater than equal to.
- <= Less than equal to.
- <> Not equal to.

Example Query:

```
SELECT * FROM MATHS WHERE MARKS=50;
```



The screenshot shows a SQL query editor window titled 'SQLQuery1.sql - D...P-S3KL81P\HP (60))'. The query entered is 'SELECT * FROM MATHS WHERE MARKS=50;'. Below the query, the results are displayed in a table with columns 'ROLL_NUMBER', 'S_NAME', and 'MARKS'. The results table has one row with values 5, MOHAN, and 50.

	ROLL_NUMBER	S_NAME	MARKS
1	5	MOHAN	50

Logical Operators

The Logical operators are those that are true or false. They return true or false values to combine one or more true or false values.

OperatorDescription

AND

Logical AND compares two Booleans as expressions and returns true when both expressions are true.

OR

Logical OR compares two Booleans as expressions and returns true when one of the expressions is true.

NOT

Not takes a single Boolean as an argument and change its value from false to true or from true to false.

Example Query:

```
SELECT * FROM employee WHERE emp_city =
```

```
'Allahabad' AND emp_country = 'India';
```

	emp_id	emp_name	emp_city	emp_country
1	104	Utkarsh Singh	Allahabad	India
2	105	Sudhanshu Yadav	Allahabad	India

Special Operators

Operators	Description
-----------	-------------

ALL

ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list of results from a query. The ALL must be preceded by the comparison operators and evaluated to TRUE if the query returns no rows.

ANY

ANY compares a value to each value in a list of results from a query and evaluates to true if the result of an inner query contains at least one row.

BETWEEN

The SQL BETWEEN operator tests an expression against a range. The range consists of a beginning, followed by an AND keyword and an end expression.

IN

The IN operator checks a value within a set of values separated by commas and retrieves the rows from the table that match.

EXISTS

The EXISTS checks the existence of a result of a subquery. The EXISTS subquery tests whether a subquery fetches at least one row. When no data is returned then this operator returns 'FALSE'.

SOME SOME operator evaluates the condition between the outer and inner tables and evaluates to true if the final result returns any one row. If not, then it evaluates to false.

UNIQUE The UNIQUE operator searches every unique row of a specified table.

Example Query:

```
SELECT * FROM employee WHERE emp_id BETWEEN 101 AND 104;
```

Results		Messages		
	emp_id	emp_name	emp_city	emp_country
1	101	Utkarsh Tripathi	Varanasi	India
2	102	Abhinav Singh	Varanasi	India
3	103	Utkarsh Raghuvanshi	Varanasi	India
4	104	Utkarsh Singh	Allahabad	India

SQL Server Functions

Functions in SQL Server are the database objects that contains a set of SQL statements to perform a specific task. A function accepts input parameters, perform actions, and then return the result. We should note that functions always return either a single value or a table. The main purpose of functions is to replicate the common task easily. We can build functions one time and can use them in multiple locations based on our needs. SQL Server does not allow to use of the functions for inserting, deleting, or updating records in the database tables.

The following are the rules for creating SQL Server functions:

A function must have a name, and the name cannot begin with a special character such as @, \$, #, or other similar characters.

SELECT statements are the only ones that operate with functions.

We can use a function anywhere such as AVG, COUNT, SUM, MIN, DATE, and other functions with the SELECT query in SQL.

Whenever a function is called, it compiles.

Functions must return a value or result.

Functions use only input parameters.

We cannot use TRY and CATCH statements in functions.

Types of Functions

SQL Server categorizes the functions into two types:

System Functions

User-Defined Functions

System Functions

Functions that are defined by the system are known as system functions. In other words, all the built-in functions supported by the server are referred to as System functions. The built-in functions save us time while performing the specific task. These types of functions usually work with the SQL SELECT statement to calculate values and manipulate data.

Here is the list of some system functions used in the SQL Server:

String Functions (LEN, SUBSTRING, REPLACE, CONCAT, TRIM)

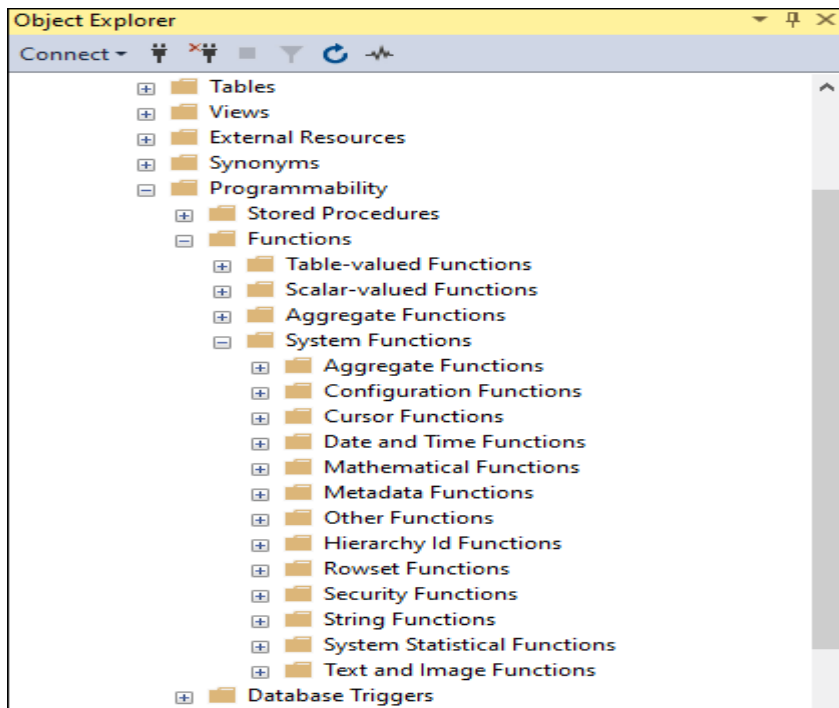
Date and Time Functions (datetime, datetime2, smalldatetime)

Aggregate Functions (COUNT, MAX, MIN, SUM, AVG)

Mathematical Functions (ABS, POWER, PI, EXP, LOG)

Ranking Functions (RANK, DENSE_RANK, ROW_NUMBER, NTILE)

The following picture shows all the built-in database functions used in the SQL Server:



User-Defined Functions

Functions that are created by the user in the system database or a user-defined database are known as user-defined functions. The UDF functions accept parameters, perform actions, and returns the result. These functions help us to simplify our development by encapsulating complex business logic and making it available for reuse anywhere based on the needs. The user-defined functions make the code needed to query data a lot easier to write. They also improve query readability and functionality, as well as allow other users to replicate the same procedures.

SQL Server categorizes the user-defined functions mainly into two types:

Scalar Functions

Table-Valued Functions

Scalar Functions

Scalar function in SQL Server always accepts parameters, either single or multiple and returns a single value. The scalar functions are useful in the simplification of our code. Suppose we might have a complex computation that appears in a number of queries. In such a case, we can build a scalar function that encapsulates the formula and uses it in each query instead of in each query.

Example:

```
CREATE FUNCTION udfNet_Sales(  
    @quantity INT,  
    @price DEC(10,2),  
    @discount DEC(3,2)  
)  
RETURNS DEC(10,2)  
AS  
BEGIN  
    RETURN @quantity * @price * (1 - @discount);  
END;
```

Output:

Results		Messages	
net_sales			
1	10000.00		

Table-Valued Functions

Table-valued functions in SQL Server are the user-defined function that returns data of a table type. Since this function's return type is a table, we can use it the same way as we use a table.

Inline Table-Values Functions

This UDF function returns a table variable based on the action performed by the function. A single SELECT statement should be used to determine the value of the table variable.

Example

The below example will create a table-values function and retrieve the data of the employee table:

```
--It creates a table-valued function to get employees  
CREATE FUNCTION fudf_GetEmployee()  
RETURNS TABLE  
AS  
RETURN (SELECT * FROM Employee)
```

Output:

id	emp_name	performance	salary
1	John Doe	11	62000
2	Mary Greenspan	13	55000
3	Grace Smith	14	85000
4	Mike Johnson	15	250000
5	Rose Dell	13	58000
6	Nancy Jackson	12	55000
7	Peter Bush	13	42000

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

SQL Statement:

[Get your own SQL server](#)

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

Edit the SQL Statement, and click "Run SQL" to see the result.

[Run SQL »](#)

Result:

Number of Records: 196

OrderID	CustomerName	OrderDate
10248	Wilman Kala	1996-07-04
10249	Tradição Hipermercados	1996-07-05
10250	Hanari Carnes	1996-07-08

Different Types of SQL JOINS

Here are the different types of the JOINS in SQL:

(INNER) JOIN: Returns records that have matching values in both tables

LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table

RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table

FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table

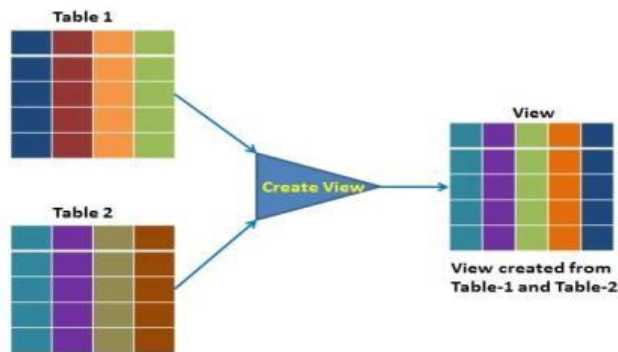
Experiment-5:

Creating different types of views for tailored presentation of data:

Database Administrator and Database Users will face two challenges: writing complex SQL queries and securing database access. Sometimes SQL queries become more complicated due to the use of multiple joins, subqueries, and GROUP BY in a single query. To simplify such queries, you can use some proxy over the original table. Also, Sometimes from the security side, the database administrator wants to restrict direct access to the database. For example, if a table contains various columns but the user only needs 3 columns of data in such case DBA will create a virtual table of 3 columns. For both purposes, you can use the view. Views can act as a proxy or virtual table. Views reduce the complexity of SQL queries and provide secure access to underlying tables.

What is a View?

Views are a special version of tables in SQL. They provide a virtual table environment for various complex operations. You can select data from multiple tables, or you can select specific data based on certain criteria in views. It does not hold the actual data; it holds only the definition of the view in the data dictionary.

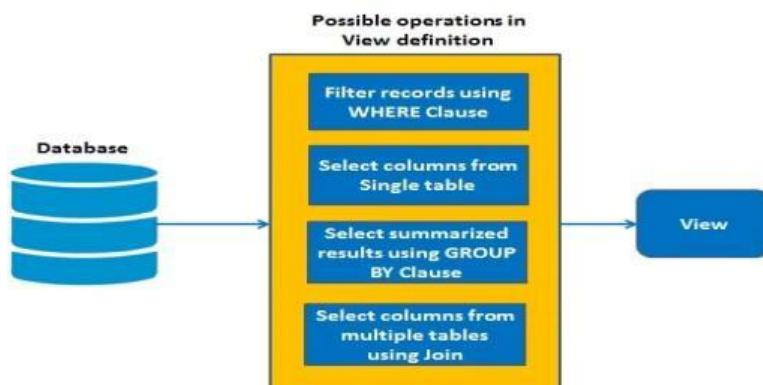


The view is a query stored in the data dictionary, on which the user can query just like they do on tables. It does not use the physical memory, only the query is stored in the data dictionary. It is computed dynamically, whenever the user performs any query on it. Changes made at any point in view are reflected in the actual base table.

The view has primarily two purposes:

Simplify the complex SQL queries.

Provide restriction to users from accessing sensitive data.



Types of Views

Simple View: A view based on only a single table, which doesn't contain GROUP BY clause and any functions.

Complex View: A view based on multiple tables, which contain GROUP BY clause and functions.

Inline View: A view based on a subquery in FROM Clause, that subquery creates a temporary table and simplifies the complex query.

Materialized View: A view that stores the definition as well as data. It creates replicas of data by storing it physically.

View	Description
Simple View	A view based on the only a single table, which doesn't contain GROUP BY clause and any functions.
Complex View	A view based on multiple tables, which contain GROUP BY clause and functions
Inline view	A view based on a subquery in FROM Clause, that subquery creates a temporary table and simplifies the complex query.
Materialized view	A view that stores the definition as well as data. It creates replicas of data by storing it physically.

Simple View

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
CREATE VIEW emp_view AS  
SELECT EmployeeID, Ename  
FROM Employee  
WHERE DeptID=2;
```

Creating View
by filtering
records using
WHERE clause

emp_view

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1004	David	2	5000
1005	Mark	2	3000

Complex View

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
CREATE VIEW emp_view AS  
SELECT DeptID, AVG(Salary)  
FROM Employee  
GROUP BY DeptID;
```

Create View of
grouped records
on Employee
table

emp_view

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

Inline View

```
SELECT column_names, ...  
FROM (subquery)  
WHERE ROWNUM<= N;
```

Materialized View

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
CREATE MATERIALIZED VIEW emp_view AS  
SELECT EmployeeID, Ename  
FROM Employee  
WHERE DeptID=2;
```

Creating Materialized View
by filtering records using
WHERE clause

emp_view

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1004	David	2	5000
1005	Mark	2	3000

This view stores the retrieved data physically on memory.

Experiment-6:

How to apply Conditional Controls in PL/SQL:

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction of flow of program execution. Decision-making statements available in pl/SQL are:

if then statement

if then else statements

nested if-then statements

if-then-elseif-then-else ladder

if then statement :if then statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

if condition then

-- do something

end if;

if – then- else:The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:-

if (condition) then

-- Executes this block if

-- condition is true

else

-- Executes this block if

-- condition is false

nested-if-then: A nested if-then is an if statement that is the target of another if statement. Nested if-then statements mean an if statement inside another if statement. Yes, PL/SQL allows us to nest if statements within if-then statements. i.e, we can place an if then statement inside another if then statement. Syntax:-

if (condition1) then

-- Executes when condition1 is true

```
if (condition2) then
```

```
-- Executes when condition2 is true
```

```
end if;
```

```
end if;
```

if-then-elsif-then-else ladder Here, a user can decide among multiple options. The if then statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. Syntax:-

```
if (condition) then
```

```
--statement
```

```
elsif (condition) then
```

```
--statement
```

```
.
```

```
.
```

```
else
```

```
--statement
```

```
Endif
```

Example:

```
DECLARE
    a number(3) := 500;
BEGIN
    -- check the boolean condition using if statement
    IF( a < 20 ) THEN
        -- if condition is true then print the following
        dbms_output.put_line('a is less than 20 ');
    ELSE
        dbms_output.put_line('a is not less than 20 ');
    END IF;
    dbms_output.put_line('value of a is : ' || a);
END;
```

Output:

```
a is not less than 20
value of a is : 500
PL/SQL procedure successfully completed.
```

Experiment-7:

Error handling using Internal Exceptions and External Exceptions:

An exception is an error which disrupts the normal flow of program instructions. PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

There are two types of exceptions defined in PL/SQL

User defined exception.

System defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can 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;

System defined exceptions:

These exceptions are predefined in PL/SQL which get raised WHEN certain database rule is violated.

System-defined exceptions are further divided into two categories:

Named system exceptions.

Unnamed system exceptions.

Named system exceptions: They have a predefined name by the system like ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED etc. the list is quite big.

Unnamed system exceptions: Oracle doesn't provide name for some system exceptions called unnamed system exceptions. These exceptions don't occur frequently. These exceptions have two parts code and an associated message.

The way to handle these exceptions is to assign name to them using Pragma EXCEPTION_INIT

Syntax:

PRAGMA EXCEPTION_INIT(exception_name, -error_number);

Example:

```
DECLARE
  exp exception;
  pragma exception_init (exp, -20015);
  n int:=10;

BEGIN
  FOR i IN 1..n LOOP
    dbms_output.put_line(i*i);
    IF i*i=36 THEN
      RAISE exp;
    END IF;
  END LOOP;

EXCEPTION
  WHEN exp THEN
    dbms_output.put_line('Welcome to GeeksforGeeks');

END;
```

Output:

```
1
4
9
16
25
36
Welcome to GeeksforGeeks
```


User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is –

DECLARE

my-exception EXCEPTION;

```
DECLARE
  c_id customers.id%type := &cc_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    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);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Output:

```
Enter value for cc_id: -6 (let's enter a value -6)
old 2: c_id customers.id%type := &cc_id;
new 2: c_id customers.id%type := -6;
ID must be greater than zero!
```

Experiment-8:

Using various types of cursors:

Cursor is a Temporary Memory or Temporary Work Station. It is Allocated by Database Server at the Time of Performing DML(Data Manipulation Language) operations on the Table by the User. Cursors are used to store Database Tables.

There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors. These are explained as following below.

Implicit Cursors: Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.

Explicit Cursors: Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

How To Create Explicit Cursor?

Declare Cursor Object

Syntax:

```
DECLARE cursor_name CURSOR FOR SELECT * FROM table_name
```

Query:

```
DECLARE s1 CURSOR FOR SELECT * FROM studDetails
```

Open Cursor Connection

Syntax:

```
OPEN cursor_connection
```

Query:

```
OPEN s1
```

Fetch Data from the Cursor There is a total of 6 methods to access data from the cursor. They are as follows:

FIRST is used to fetch only the first row from the cursor table.

LAST is used to fetch only the last row from the cursor table.

NEXT is used to fetch data in a forward direction from the cursor table.

PRIOR is used to fetch data in a backward direction from the cursor table.

ABSOLUTE n is used to fetch the exact nth row from the cursor table.

RELATIVE n is used to fetch the data in an incremental way as well as a decremental way.

Syntax:

FETCH NEXT/FIRST/LAST/PRIOR/ABSOLUTE n/RELATIVE n FROM cursor_name

Query:

FETCH FIRST FROM s1

FETCH LAST FROM s1

FETCH NEXT FROM s1

FETCH PRIOR FROM s1

FETCH ABSOLUTE 7 FROM s1

FETCH RELATIVE -2 FROM s1

Close cursor connection

Syntax:

CLOSE cursor_name

Query:

CLOSE s1

Deallocate cursor memory

Syntax:

DEALLOCATE cursor_name

Query:

DEALLOCATE s1

How To Create an Implicit Cursor?

An implicit cursor is a cursor that is automatically created by PL/SQL when you execute a SQL statement. You don't need to declare or open an implicit cursor explicitly. Instead, PL/SQL manages the cursor for you behind the scenes.

To create an implicit cursor in PL/SQL, you simply need to execute a SQL statement. For example, to retrieve all rows from the EMP table, you can use the following code:

Query:

BEGIN

FOR emp_rec IN SELECT * FROM emp LOOP

DBMS_OUTPUT.PUT_LINE('Employee name: ' || emp_rec.ename);

END LOOP;

END;

In PL/SQL, when we perform INSERT, UPDATE or DELETE operations, an implicit cursor is automatically created. This cursor holds the data to be inserted or identifies the rows to be updated or deleted. You can refer to this cursor as the SQL cursor in your code. This SQL cursor has several useful attributes.

%FOUND is true if the most recent SQL operation affected at least one row.

%NOTFOUND is true if it didn't affect any rows.

%ROWCOUNT is returns the number of rows affected.

%ISOPEN checks if the cursor is open.

In addition to these attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS are specific to the FORALL statement, which is used to perform multiple DML operations at once. %BULK_ROWCOUNT returns the number of rows affected by each DML operation, while %BULK_EXCEPTION returns any exception that occurred during the operations.

Example:

```
CREATE TABLE Emp(  
    EmpID INT PRIMARY KEY,  
    Name VARCHAR(50),  
    Country VARCHAR(50),  
    Age int(2),  
    Salary int(10)  
);  
-- Insert some sample data into the Customers table  
INSERT INTO Emp (EmpID, Name, Country, Age, Salary)  
VALUES (1, 'Shubham', 'India', '23', '30000'),  
       (2, 'Aman ', 'Australia', '21', '45000'),  
       (3, 'Naveen', 'Sri lanka', '24', '40000'),  
       (4, 'Aditya', 'Austria', '21', '35000'),  
       (5, 'Nishant', 'Spain', '22', '25000');  
Select * from Emp;
```

Output:

EmpID	Name	Country	Age	Salary
1	Shubham	India	23	30000
2	Aman	Australia	21	45000
3	Naveen	Sri lanka	24	40000
4	Aditya	Austria	21	35000
5	Nishant	Spain	22	25000

Experiment-9:

How to run Stored Procedures and Functions:

Stored procedures are prepared SQL code that you save so you can reuse it over and over again. So if you have an SQL query that you write over and over again, save it as a stored procedure and call it to run it. You can also pass parameters to stored procedures so that the stored procedure can act on the passed parameter values.

Stored Procedures are created to perform one or more DML operations on Database. It is nothing but the group of SQL statements that accepts some input in the form of parameters and performs some task and may or may not return a value.

Syntax:

Creating a Procedure

```
CREATE PROCEDURE procedure_name  
  
(parameter1 data_type, parameter2 data_type, ...)  
  
AS  
  
BEGIN  
    — SQL statements to be executed
```

```
END
```

To Execute the procedure

```
EXEC procedure_name parameter1_value, parameter2_value, ..
```

Parameter Explanation

The most important part is the parameters. Parameters are used to pass values to the Procedure. There are different types of parameters, which are as follows:

BEGIN: This is what directly executes or we can say that it is an executable part.

END: Up to this, the code will get executed.

Example:

```
-- Create a new database named "SampleDB"
CREATE DATABASE SampleDB;

-- Switch to the new database
USE SampleDB;

-- Create a new table named "Customers"
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(50),
    ContactName VARCHAR(50),
    Country VARCHAR(50)
);

-- Insert some sample data into the Customers table
INSERT INTO Customers (CustomerID, CustomerName, ContactName, Country)
VALUES (1, 'Shubham', 'Thakur', 'India'),
      (2, 'Aman ', 'Chopra', 'Australia'),
      (3, 'Naveen', 'Tulasi', 'Sri lanka'),
      (4, 'Aditya', 'Arpan', 'Austria'),
      (5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain');

-- Create a stored procedure named "GetCustomersByCountry"
CREATE PROCEDURE GetCustomersByCountry
    @Country VARCHAR(50)
AS
BEGIN
    SELECT CustomerName, ContactName
    FROM Customers
    WHERE Country = @Country;
END;

-- Execute the stored procedure with parameter "Sri lanka"
EXEC GetCustomersByCountry @Country = 'Sri lanka';
```

Output:

CustomerName	Contact Name
Naveen	Tulasi

The **CREATE FUNCTION** statement is used for creating a stored function and user-defined functions. A stored function is a set of SQL statements that perform some operation and return a single value.

Just like Mysql in-built function, it can be called from within a Mysql statement.

By default, the stored function is associated with the default database.

The CREATE FUNCTION statement require CREATE ROUTINE database privilege.

Syntax:

The syntax for CREATE FUNCTION statement in Mysql is:

```
CREATE FUNCTION function_name(func_parameter1, func_parameter2, ..)
    RETURN datatype [characteristics]
    func_body
```

Parameters used:

function_name:

It is the name by which stored function is called. The name should not be same as native(built_in) function. In order to associate routine explicitly with a specific database function name should be given as database_name.func_name.

func_parameter:

It is the argument whose value is used by the function inside its body. You can't specify to these parameters IN, OUT, INOUT. The parameter declaration inside parenthesis is provided as func_parameter type. Here, type represents a valid Mysql datatype.

datatype:

It is datatype of value returned by function.

characteristics:

The CREATE FUNCTION statement is accepted only if at least one of the characteristics { DETERMINISTIC, NO SQL, or READS SQL DATA } is specified in its declaration.

func_body is the set of Mysql statements that perform operation. It's structure is as follows:

BEGIN

 Mysql Statements

 RETURN expression;

END

The function body must contain one RETURN statement.

Example:

Consider following Employee Table-

emp_id	fname	lname	start_date
1	Michael	Smith	2001-06-22
2	Susan	Barker	2002-09-12
3	Robert	Tvler	2000-02-09
4	Susan	Hawthorne	2002-04-24

```
DELIMITER //
```

```
CREATE FUNCTION no_of_years(date1 date) RETURNS int DETERMINISTIC
```

```
BEGIN
```

```
  DECLARE date2 DATE;
```

```
  Select current_date()into date2;
```

```
  RETURN year(date2)-year(date1);
```

```
END
```

```
//
```

```
DELIMITER ;
```

Calling of above function:

```
Select emp_id, fname, lname, no_of_years(start_date) as 'years' from employee;
```

Output:

emp_id	fname	lname	years
1	Michael	Smith	18
2	Susan	Barker	17
3	Robert	Tvler	19
4	Susan	Hawthorne	17

Experiment-10:

Creating Packages and applying Triggers:

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 the 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 the 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. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
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 the SQL prompt, it produces the following result –

Package body created.

Example:

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records –

```

Select * from customers;

+-----+-----+-----+-----+-----+
| ID | NAME   | AGE | ADDRESS | SALARY |
+-----+-----+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |
| 2 | Khilan | 25 | Delhi     | 3000.00 |
| 3 | kaushik | 23 | Kota      | 3000.00 |
| 4 | Chaitali | 25 | Mumbai    | 7500.00 |
| 5 | Hardik | 27 | Bhopal    | 9500.00 |
| 6 | Komal | 22 | MP        | 5500.00 |
+-----+-----+-----+-----+

```

The Package Specification

```

CREATE OR REPLACE PACKAGE c_package AS
  -- Adds a customer
  PROCEDURE addCustomer(c_id customers.id%type,
    c_name customers.Name%type,
    c_age customers.age%type,
    c_addr customers.address%type,
    c_sal customers.salary%type);

  -- Removes a customer
  PROCEDURE delCustomer(c_id customers.id%TYPE);
  --Lists all customers
  PROCEDURE listCustomer;

END c_package;

/

```

Output:

Package created.

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
    PROCEDURE addCustomer(c_id customers.id%type,
        c_name customers.Name%type,
        c_age customers.age%type,
        c_addr customers.address%type,
        c_sal customers.salary%type)
    IS
    BEGIN
        INSERT INTO customers (id,name,age,address,salary)
            VALUES(c_id, c_name, c_age, c_addr, c_sal);
    END addCustomer;

    PROCEDURE delCustomer(c_id customers.id%type) IS
    BEGIN
        DELETE FROM customers
            WHERE id = c_id;
```

```
END delCustomer;
```

```
PROCEDURE listCustomer IS
CURSOR c_customers is
    SELECT name FROM customers;
TYPE c_list is TABLE OF customers.Name%type;
name_list c_list := c_list();
counter integer :=0;
BEGIN
    FOR n IN c_customers LOOP
        counter := counter +1;
        name_list.extend;
        name_list(counter) := n.name;
        dbms_output.put_line('Customer(' ||counter|| ')'||name_list
    END LOOP;
END listCustomer;
```

```
END c_package;
```

```
/
```

Package body created.

Using The Package

The following program uses the methods declared and defined in the package c_package.

```
DECLARE
    code customers.id%type:= 8;
BEGIN
    c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
    c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
    c_package.listcustomer;
    c_package.delcustomer(code);
    c_package.listcustomer;
END;
/
```

Output:

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
```

PL/SQL procedure successfully completed

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

Benefits of Triggers

Triggers can be written for the following purposes –

Generating some derived column values automatically

Enforcing referential integrity

Event logging and storing information on table access

Auditing

Synchronous replication of tables

Imposing security authorizations

Preventing invalid transactions

Syntax:

```
create trigger [trigger_name]
```

```
[before | after]
```

```
{insert | update | delete}
```

```
on [table_name]
```

```
[for each row]
```

```
[trigger_body]
```

BEFORE and AFTER Trigger

BEFORE triggers run the trigger action before the triggering statement is run. AFTER triggers run the trigger action after the triggering statement is run.

Example:

Suppose the Database Schema

Query

```
mysql>>desc Student;
```

Field	Type	Null	Key	Default	Extra
tid	int(4)	NO	PRI	NULL	auto_increment
name	varchar(30)	YES		NULL	
subj1	int(2)	YES		NULL	
subj2	int(2)	YES		NULL	
subj3	int(2)	YES		NULL	
total	int(3)	YES		NULL	
per	int(3)	YES		NULL	

SQL Trigger to the problem statement.

```
CREATE TRIGGER stud_marks
BEFORE INSERT ON Student
FOR EACH ROW
SET NEW.total = NEW.subj1 + NEW.subj2 + NEW.subj3,
    NEW.per = (NEW.subj1 + NEW.subj2 + NEW.subj3) * 60 / 100;
```

Output:

Trigger	Event	Table	Timing	Created	Status
stud_marks	BEFORE	Student	BEFORE INSERT	2023-05-02 00:00:00	ACTIVE

Experiment-11:

Creating Arrays And Nested Tables:

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.

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

```
table_name type_name;
```

creating an index-by table named table_name, the keys of which will be of the subscript_type and associated values will be of the element_type

Example:

```
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;
```

Output:

```
Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000

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.

A 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 the declaration of an index-by table, but there is no INDEX BY clause.

A nested table can be stored in a database column. It can further 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:

Consider the following 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


```

DECLARE
    CURSOR c_customers IS
        SELECT name FROM customers;
    TYPE c_list IS TABLE OF customerS.No.ame%type;
    name_list c_list := c_list();
    counter integer :=0;
BEGIN
    FOR n IN c_customers LOOP
        counter := counter +1;
        name_list.extend;
        name_list(counter) := n.name;
        dbms_output.put_line('Customer('||counter||'):'||name_list
    END LOOP;
END;
/

```

Output:

```

Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

```

PL/SQL procedure successfully completed.