Practical File

Relational Database Management System

Subject Code: (PGCA-2208)

Submitted To:

Prof. Satinder Pal Singh

Submitted By:

Kajal

MCA- I Sem

2329011

Department of Computer Applications

Guru Nanak Dev Engineering College

INDEX

Sr No.	Program	Page no.	Date	Remarks
1	Comparative study of various Database Management Systems			
2	• Data Definition Language (DDL) • Data Manipulation Language (DML) • Data Control Language (DCL)			
3	How to apply Constraints at various levels.			
4	View data in the required form using Operators, Functions and Joins.			
5	Creating different types of Views for tailored presentation of data			
6	How to apply Conditional Controls in PL/SQL			
7	Error Handling using Internal Exceptions and External Exceptions			
8	Using various types of Cursors			
9	How to run Stored Procedures and Functions			
10	Creating Packages and applying Triggers			
11	Creating Arrays and Nested Tables.			

Experiment:- 1

Comparative study of various Database Management Systems

A comparative study of various Database Management Systems (DBMS) involves analyzing different systems based on several criteria such as performance, scalability, reliability, security, ease of use, and cost. There are various types of DBMS available, including relational, NoSQL, NewSQL, and object-oriented databases. Here's a general overview of how different types of DBMS compare based on these criteria:

1. Relational Database Management Systems (RDBMS)

Examples: MySQL, PostgreSQL, Oracle, SQL Server

Advantages

- Mature technology with a well-defined standard (SQL).
- ACID transactions ensure data consistency and reliability.
- Suitable for complex queries and transactions.
- Support for joins and relationships between tables.

Disadvantages

- May not scale well horizontally.
- Schema changes can be complex.

Use Cases

- Business applications requiring complex queries and transactions.
- Situations where data integrity and consistency are critical.

2. NoSQL Database Management Systems

Examples: MongoDB, Cassandra, Redis, Couchbase

Advantages

- Schema flexibility; can handle semi-structured or unstructured data.
- Horizontal scalability; easy to distribute across multiple nodes.
- Suitable for large volumes of rapidly changing data.

Disadvantages

- Lack of ACID transactions in some cases.
- Limited support for complex queries.

Use Cases

- Big data and real-time applications.
- Content management systems and e-commerce platforms.
- IoT applications.

3. NewSQL Database Management Systems

Examples: Google Spanner, CockroachDB

Advantages

- Combines benefits of traditional RDBMS and NoSQL databases.
- Horizontal scalability with strong consistency.

Disadvantages

- Limited adoption and maturity compared to traditional RDBMS.

Use Cases

- Applications requiring global distribution and strong consistency.
- Financial applications and online transaction processing (OLTP) systems.

4. Object-Oriented Database Management Systems (OODBMS)

Examples db4o, ObjectDB

Advantages

- Store complex data types directly without the need for complex mapping.
- Support for object-oriented programming concepts.

Disadvantages

- Limited support and adoption compared to relational databases.
- Performance concerns for certain types of queries.

- Use Cases

- Object-oriented applications and systems with complex data structures.
- Applications where objects need to be persisted directly without mapping to tables.

Comparison Criteria:

1. Performance

- RDBMS: Good for complex queries and transactions.
- NoSQL: High performance for read and write operations, especially in large-scale applications.
- NewSQL: Balanced performance for both complex queries and scalable transactions.
- OODBMS: Performance depends on the complexity of object relationships.

2. Scalability

- RDBMS: Vertical scaling can be a limitation, but some offer limited horizontal scaling.
- NoSQL: Excellent horizontal scalability across distributed systems.
- NewSQL: Horizontal scalability with strong consistency.
- OODBMS: Limited scalability compared to other types.

3. Reliability and Consistency

- RDBMS: ACID transactions ensure data integrity and consistency.
- NoSQL: Eventual consistency in many cases, but some offer strong consistency options.
- NewSQL: ACID transactions with horizontal scalability.
- OODBMS: ACID transactions for object-oriented data.

4. Flexibility and Schema Design

- RDBMS: Schema is fixed and must be defined before data insertion.
- NoSQL: Flexible schema; can handle semi-structured and unstructured data.
- NewSQL: Similar to RDBMS with schema definition.
- OODBMS: Schema flexibility, supports complex object structures.

5. Security

- RDBMS: Mature security features, including role-based access control and encryption.
- NoSQL: Security features vary; some offer robust security mechanisms.
- NewSQL: Security features comparable to traditional RDBMS.
- OODBMS: Security features are usually present but may not be as extensive as RDBMS.

6. Ease of Use and Development

- RDBMS: Mature tools, well-established query language (SQL).
- NoSQL: Diverse query languages, may require learning different interfaces for different databases.
- NewSQL: SQL-based interfaces, similar to traditional RDBMS.
- OODBMS: Object-oriented query languages, suitable for developers familiar with object-oriented programming concepts.

7. Community and Support

- RDBMS: Large, active communities and extensive documentation.
- NoSQL: Active communities, but support may vary based on the specific database.
- NewSQL: Smaller communities compared to RDBMS and NoSQL.
- OODBMS: Smaller communities; limited resources and support compared to RDBMS.

Conclusion

- Choose an RDBMS when data integrity, ACID transactions, and complex queries are crucial.
- Opt for NoSQL databases for flexible schema, horizontal scalability, and handling large volumes of semi-structured/unstructured data.
- Consider NewSQL databases for applications requiring global distribution and strong consistency.
- Use OODBMS for applications with complex object structures and a need for direct object persistence.

Experiment:- 2

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

What is SQL?

SQL is a database language designed for the retrieval and management of data in a relational database.

SQL is the standard language for database management. All the RDBMS systems like MySQL, MS Access, Oracle, Sybase, Postgres, and SQL Server use SQL as their standard database language. SQL programming language uses various commands for different operations. We will learn about the like DCL, TCL, DQL, DDL and DML

Brief History of SQL

Here, are important landmarks from the history of SQL:

- 1970 – Dr. Edgar F. "Ted" Codd described a relational model for databases.
- 1974 – Structured Query Language appeared.
- 1978 – IBM released a product called System/R.
- 1986 – IBM developed the prototype of a relational database, which is standardized by ANSI.
- 1989- First ever version launched of SQL
- 1999 – SQL 3 launched with features like triggers, object-orientation, etc.
- SQL2003- window functions, XML-related features, etc.
- SQL2006- Support for XML Query Language
- SQL2011-improved support for temporal databases

Types of SQL

Here are five types of widely used SQL queries.

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language(DCL)

What is DDL?

Data Definition Language helps you to define the database structure or schema. Let's learn about DDL commands with syntax.

Five types of DDL commands in SQL are:

CREATE

CREATE statements is used to define the database structure schema:

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);
For example:

Create database university;
Create table students;
Create view for_students;

DROP

Drops commands remove tables and databases from RDBMS.

Syntax

DROP TABLE ;
For example:

Drop object_type object_name;
Drop database university;
Drop table student;

ALTER

Alters command allows you to alter the structure of the database.

Syntax:

To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;
To modify an existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);
For example:

Alter table guru99 add subject varchar;

TRUNCATE

This command used to delete all the rows from the table and free the space containing the table.

Syntax:

TRUNCATE TABLE table_name;
Example:

TRUNCATE table students;

What is Data Manipulation Language?

Data Manipulation Language (DML) allows you to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types of data modification in a database.
There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

- INSERT
- UPDATE
- DELETE

INSERT

This is a statement is a SQL query. This command is used to insert data into the row of a table.

Syntax:

INSERT INTO TABLE_NAME (col1, col2, col3,.... col N)
VALUES (value1, value2, value3, ... valueN);
Or
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3, ... valueN);
For example:

INSERT INTO students (RollNo, FIrstName, LastName) VALUES ('60', 'Tom', Erichsen');

UPDATE

This command is used to update or modify the value of a column in the table.

Syntax:

UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]
For example:

UPDATE students
SET FirstName = 'Jhon', LastName= 'Wick'
WHERE StudID = 3;

DELETE

This command is used to remove one or more rows from a table.

Syntax:

DELETE FROM table_name [WHERE condition];
For example:

DELETE FROM students
WHERE FirstName = 'Jhon';

What is DCL?

DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give “rights & permissions.” Other permission controls parameters of the database system.

Examples of DCL commands

Commands that come under DCL:

- Grant
- Revoke

Grant

This command is use to give user access privileges to a database.

Syntax:

GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
For example:

GRANT SELECT ON Users TO'Tom'@'localhost;

Revoke

It is useful to back permissions from the user.

Syntax:

REVOKE privilege_nameON object_nameFROM {user_name |PUBLIC |role_name}
For example:

REVOKE SELECT, UPDATE ON student FROM BCA, MCA;

Summary

- SQL is a database language designed for the retrieval and management of data in a relational database.
- It helps users to access data in the RDBMS system
- In the year 1974, the term Structured Query Language appeared
- Five types of SQL queries are 1) Data Definition Language (DDL) 2) Data Manipulation Language (DML) 3) Data Control Language(DCL) 4) Transaction Control Language(TCL) and, 5) Data Query Language (DQL)
- Data Definition Language(DDL) helps you to define the database structure or schema.
- Data Manipulation Language (DML) allows you to modify the database instance by inserting, modifying, and deleting its data.
- DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give "rights & permissions."
- Transaction control language or TCL commands deal with the transaction within the database.
- Data Query Language (DQL) is used to fetch the data from the database.

3. How to apply Constraints at various levels.

Experiment:- 3

How to apply Constraints at various levels.

Constraints in SQL

Constraints in SQL means we are applying certain conditions or restrictions on the database. This further means that before inserting data into the database, we are checking for some conditions. If the condition we have applied to the database holds true for the data which is to be inserted, then only the data will be inserted into the database tables.

Constraints in SQL can be categorized into two types:

1. **Column Level Constraint:**
   Column Level Constraint is used to apply a constraint on a single column.
2. **Table Level Constraint:**
   Table Level Constraint is used to apply a constraint on multiple columns.

Constraints available in SQL are:

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK
6. DEFAULT
7. CREATE INDEX

Now let us try to understand the different constraints available in SQL in more detail with the help of examples. We will use MySQL database for writing all the queries.

1. NOT NULL

- NULL means empty, i.e., the value is not available.
- Whenever a table's column is declared as NOT NULL, then the value for that column cannot be empty for any of the table's records.
- There must exist a value in the column to which the NOT NULL constraint is applied.

CREATE TABLE TableName (ColumnName1 datatype NOT NULL, ColumnName2 datatype,…., C olumnNameN datatype);
Example:

Create a student table and apply a NOT NULL constraint on one of the table's column while creating a table.

1. **CREATE TABLE** student(StudentID **INT** NOT NULL, Student_FirstName **VARCHAR**(20), Student_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Student_E mail_ID **VARCHAR**(40));

```
mysql> CREATE TABLE student(StudentID INT NOT NULL, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20), Student_
Email_ID VARCHAR(40));
Query OK, 0 rows affected (0.28 sec)
```

To verify that the not null constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| StudentID          | int(11)     | NO   |     | NULL    |       |
| Student_FirstName  | varchar(20) | YES  |     | NULL    |       |
| Student_LastName   | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID   | varchar(40) | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
5 rows in set (0.14 sec)
```

Syntax to apply the NOT NULL constraint on an existing table's column:

1. **ALTER TABLE** TableName CHANGE Old_ColumnName New_ColumnName Datatype N OT NULL;

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a NOT NULL constraint to one of the table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student CHANGE StudentID StudentID **INT** NOT NULL;

```
mysql> ALTER TABLE student CHANGE StudentID StudentID INT NOT NULL;
Query OK, 0 rows affected (0.26 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the not null constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| StudentID          | int(11)     | NO   |     | NULL    |       |
| Student_FirstName  | varchar(20) | YES  |     | NULL    |       |
| Student_LastName   | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID   | varchar(40) | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

2. UNIQUE

- Duplicate values are not allowed in the columns to which the UNIQUE constraint is applied.
- The column with the unique constraint will always contain a unique value.
- This constraint can be applied to one or more than one column of a table, which means more than one unique constraint can exist on a single table.
- Using the UNIQUE constraint, you can also modify the already created tables.

Syntax to apply the UNIQUE constraint on a single column:

1. **CREATE TABLE** TableName (ColumnName1 datatype **UNIQUE**, ColumnName2 datatype ,…., ColumnNameN datatype);

Example:

Create a student table and apply a UNIQUE constraint on one of the table's column while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT UNIQUE**, Student_FirstName **VARCH AR**(20), Student_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Stud ent_Email_ID **VARCHAR**(40));

```
mysql> CREATE TABLE student(StudentID INT UNIQUE, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20),
 Student_Email_ID VARCHAR(40));
Query OK, 0 rows affected (0.12 sec)
```

To verify that the unique constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  | UNI | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

Syntax to apply the UNIQUE constraint on more than one column:

1. **CREATE TABLE** TableName (ColumnName1 datatype, ColumnName2 datatype,…., Colu mnNameN datatype, **UNIQUE** (ColumnName1, ColumnName 2));

Example:

Create a student table and apply a UNIQUE constraint on more than one table's column while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT**, Student_FirstName **VARCHAR**(20), St udent_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Student_Email_ ID **VARCHAR**(40), **UNIQUE**(StudentID, Student_PhoneNumber));

```
mysql> CREATE TABLE student(StudentID INT, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20), Student_
Email_ID VARCHAR(40), UNIQUE(StudentID, Student_PhoneNumber));
Query OK, 0 rows affected (0.15 sec)
```

To verify that the unique constraint is applied to more than one table's column and the student table is created successfully, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+----------------------+-------------+------+-----+---------+-------+
| Field                | Type        | Null | Key | Default | Extra |
+----------------------+-------------+------+-----+---------+-------+
| StudentID            | int(11)     | YES  | MUL | NULL    |       |
| Student_FirstName    | varchar(20) | YES  |     | NULL    |       |
| Student_LastName     | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber  | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID     | varchar(40) | YES  |     | NULL    |       |
+----------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

Syntax to apply the UNIQUE constraint on an existing table's column:

1. **ALTER TABLE** TableName **ADD UNIQUE** (ColumnName);

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a UNIQUE constraint to one of the table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student **ADD UNIQUE** (StudentID);

```
mysql> ALTER TABLE student ADD UNIQUE(StudentID);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the unique constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  | UNI | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.02 sec)
```

3. PRIMARY KEY

- PRIMARY KEY Constraint is a combination of NOT NULL and Unique constraints.
- NOT NULL constraint and a UNIQUE constraint together forms a PRIMARY constraint.
- The column to which we have applied the primary constraint will always contain a unique value and will not allow null values.

Syntax of primary key constraint during table creation:

1. **CREATE TABLE** TableName (ColumnName1 datatype **PRIMARY KEY**, ColumnName2 datatype,…., ColumnNameN datatype);

Example:

Create a student table and apply the PRIMARY KEY constraint while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT PRIMARY KEY**, Student_FirstName **V ARCHAR**(20), Student_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(2 0), Student_Email_ID **VARCHAR**(40));

```
mysql> CREATE TABLE student(StudentID INT PRIMARY KEY, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20),
 Student_Email_ID VARCHAR(40));
Query OK, 0 rows affected (0.12 sec)
```

To verify that the primary key constraint is applied to the table's column and the student table is created successfully, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| StudentID          | int(11)     | NO   | PRI | NULL    |       |
| Student_FirstName  | varchar(20) | YES  |     | NULL    |       |
| Student_LastName   | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber| varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID   | varchar(40) | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
5 rows in set (0.10 sec)
```

Syntax to apply the primary key constraint on an existing table's column:

1. **ALTER TABLE** TableName **ADD PRIMARY KEY** (ColumnName);

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply the PRIMARY KEY constraint to the table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student **ADD PRIMARY KEY** (StudentID);

To verify that the primary key constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> ALTER TABLE student ADD PRIMARY KEY(StudentID);
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> DESC Student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | NO   | PRI | 0       |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

4. FOREIGN KEY

- A foreign key is used for referential integrity.
- When we have two tables, and one table takes reference from another table, i.e., the same column is present in both the tables and that column acts as a primary key in one table. That particular column will act as a foreign key in another table.

Syntax to apply a foreign key constraint during table creation:

1. **CREATE TABLE** tablename(ColumnName1 Datatype(**SIZE**) **PRIMARY KEY**, ColumnNameN Datatype(**SIZE**), **FOREIGN KEY**(ColumnName) **REFERENCES** PARENT_TABLE_NAME(Primary_Key_ColumnName));

Example:

Create an employee table and apply the FOREIGN KEY constraint while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

1. mysql> **CREATE TABLE** employee (Emp_ID **INT** NOT NULL **PRIMARY KEY**, Emp_Name **VARCHAR** (40), Emp_Salary **VARCHAR** (40));

```
mysql> CREATE TABLE employee(Emp_ID INT NOT NULL PRIMARY KEY, Emp_Name VARCHAR(40), Emp_Salary VARCHAR(40));
Query OK, 0 rows affected (0.13 sec)
```

To verify that the primary key constraint is applied to the employee table's column, we will execute the following query:

1. mysql> **DESC** employee;

```
mysql> DESC employee;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| Emp_ID     | int(11)     | NO   | PRI | NULL    |       |
| Emp_Name   | varchar(40) | YES  |     | NULL    |       |
| Emp_Salary | varchar(40) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Now, we will write a query to apply a foreign key on the department table referring to the primary key of the employee table, i.e., Emp_ID.

1. mysql> **CREATE TABLE** department(Dept_ID **INT** NOT NULL **PRIMARY KEY**, Dept_ Name **VARCHAR**(40), Emp_ID **INT** NOT NULL, **FOREIGN KEY**(Emp_ID) **REFEREN CES** employee(Emp_ID));

```
mysql> CREATE TABLE department(Dept_ID INT NOT NULL PRIMARY KEY, Dept_Name VARCHAR(40), Emp_ID INT NOT NULL, FOREIGN KEY(Emp_ID) REFERENCES employee
(Emp_ID));
Query OK, 0 rows affected (0.10 sec)
```

To verify that the foreign key constraint is applied to the department table's column, we will execute the following query:

1. mysql> **DESC** department;

```
mysql> DESC department;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| Dept_ID   | int(11)     | NO   | PRI | NULL    |       |
| Dept_Name | varchar(40) | YES  |     | NULL    |       |
| Emp_ID    | int(11)     | NO   | MUL | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Syntax to apply the foreign key constraint with constraint name:

1. **CREATE TABLE** tablename(ColumnName1 Datatype **PRIMARY KEY**, ColumnNameN Datatype(**SIZE**), **CONSTRAINT** ConstraintName **FOREIGN KEY**(ColumnName) **REFE RENCES** PARENT_TABLE_NAME(Primary_Key_ColumnName));

Example:

Create an employee table and apply the FOREIGN KEY constraint with a constraint name while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

1. mysql> **CREATE TABLE** employee (Emp_ID **INT** NOT NULL **PRIMARY KEY**, Emp_Name **VARCHAR** (40), Emp_Salary **VARCHAR** (40));

```
mysql> CREATE TABLE employee(Emp_ID INT NOT NULL PRIMARY KEY, Emp_Name VARCHAR(40), Emp_Salary VARCHAR(40));
Query OK, 0 rows affected (0.11 sec)
```

To verify that the primary key constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** employee;

```
mysql> DESC employee;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| Emp_ID     | int(11)     | NO   | PRI | NULL    |       |
| Emp_Name   | varchar(40) | YES  |     | NULL    |       |
| Emp_Salary | varchar(40) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Now, we will write a query to apply a foreign key with a constraint name on the department table referring to the primary key of the employee table, i.e., Emp_ID.

1. mysql> **CREATE TABLE** department(Dept_ID **INT** NOT NULL **PRIMARY KEY**, Dept_Name **VARCHAR**(40), Emp_ID **INT** NOT NULL, **CONSTRAINT** emp_id_fk **FOREIGN KEY**(Emp_ID) **REFERENCES** employee(Emp_ID));

```
mysql> CREATE TABLE department(Dept_ID INT NOT NULL PRIMARY KEY, Dept_Name VARCHAR(40), Emp_ID INT NOT NULL, CONSTRAINT emp_id_fk FOREIGN KEY(Emp_ID
) REFERENCES employee(Emp_ID));
Query OK, 0 rows affected (0.12 sec)
```

To verify that the foreign key constraint is applied to the department table's column, we will execute the following query:

1. mysql> **DESC** department;

```
mysql> DESC department;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| Dept_ID   | int(11)     | NO   | PRI | NULL    |       |
| Dept_Name | varchar(40) | YES  |     | NULL    |       |
| Emp_ID    | int(11)     | NO   | MUL | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

Syntax to apply the foreign key constraint on an existing table's column:

1. **ALTER TABLE** Parent_TableName **ADD FOREIGN KEY** (ColumnName) **REFERENCE S** Child_TableName (ColumnName);

Example:

Consider we have an existing table employee and department. Later, we decided to apply a FOREIGN KEY constraint to the department table's column. Then we will execute the following query:

1. mysql> **DESC** employee;

```
mysql> DESC employee;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| Emp_ID     | int(11)     | NO   | PRI | NULL    | auto_increment |
| Emp_Name   | varchar(40) | YES  |     | NULL    |                |
| Emp_Salary | varchar(40) | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
3 rows in set (0.01 sec)
```

1. mysql> **ALTER TABLE** department **ADD FOREIGN KEY** (Emp_ID) **REFERENCES** em ployee (Emp_ID);

```
mysql> ALTER TABLE department ADD FOREIGN KEY(Emp_ID) REFERENCES employee(Emp_ID);
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the foreign key constraint is applied to the department table's column, we will execute the following query:

1. mysql> **DESC** department;

```
mysql> DESC department;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| Dept_ID   | int(11)     | NO   | PRI | NULL    | auto_increment |
| Dept_Name | varchar(40) | YES  |     | NULL    |                |
| Emp_ID    | int(11)     | NO   | MUL | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
3 rows in set (0.10 sec)
```

5. CHECK

- Whenever a check constraint is applied to the table's column, and the user wants to insert the value in it, then the value will first be checked for certain conditions before inserting the value into that column.
- **For example:** if we have an age column in a table, then the user will insert any value of his choice. The user will also enter even a negative value or any other invalid value. But, if the user has applied check constraint on the age column with the condition age greater than 18. Then in such cases, even if a user tries to insert an invalid value such as zero or any other value less than 18, then the age column will not accept that value and will not allow the user to insert it due to the application of check constraint on the age column.

Syntax to apply check constraint on a single column:

1. **CREATE TABLE** TableName (ColumnName1 datatype **CHECK** (ColumnName1 Conditio n), ColumnName2 datatype,…., ColumnNameN datatype);

Example:

Create a student table and apply CHECK constraint to check for the age less than or equal to 15 while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT**, Student_FirstName **VARCHAR**(20), St udent_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Student_Email_ ID **VARCHAR**(40), Age **INT CHECK**(Age <= 15));

```
mysql> CREATE TABLE student(StudentID INT, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20), Student_Ema
il_ID VARCHAR(40), Age INT CHECK( Age <= 15));
Query OK, 0 rows affected (0.22 sec)
```

To verify that the check constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  |     | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
| Age                 | int(11)     | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

Syntax to apply check constraint on multiple columns:

1. **CREATE TABLE** TableName (ColumnName1 datatype, ColumnName2 datatype **CHECK** (ColumnName1 Condition AND ColumnName2 Condition),…., ColumnNameN datatype);

Example:

Create a student table and apply CHECK constraint to check for the age less than or equal to 15 and a percentage greater than 85 while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT**, Student_FirstName **VARCHAR**(20), St udent_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Student_Email_ ID **VARCHAR**(40), Age **INT**, Percentage **INT**, **CHECK**(Age <= 15 AND Percentage > 85));

```
mysql> CREATE TABLE student(StudentID INT, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20), Student_Ema
il_ID VARCHAR(40), Age INT, Percentage INT, CHECK( Age <= 15 AND Percentage > 85));
Query OK, 0 rows affected (0.13 sec)
```

To verify that the check constraint is applied to the age and percentage column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| StudentID          | int(11)     | YES  |     | NULL    |       |
| Student_FirstName  | varchar(20) | YES  |     | NULL    |       |
| Student_LastName   | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber| varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID   | varchar(40) | YES  |     | NULL    |       |
| Age                | int(11)     | YES  |     | NULL    |       |
| Percentage         | int(11)     | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
7 rows in set (0.01 sec)
```

Syntax to apply check constraint on an existing table's column:

1. **ALTER TABLE** TableName **ADD CHECK** (ColumnName Condition);

Example:

Consider we have an existing table student. Later, we decided to apply the CHECK constraint on the student table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student **ADD CHECK** (Age <=15);

```
mysql> ALTER TABLE student ADD CHECK( Age <=15 );
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the check constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

6. DEFAULT

```
mysql> DESC student;
+--------------------+-------------+------+-----+---------+-------+
| Field              | Type        | Null | Key | Default | Extra |
+--------------------+-------------+------+-----+---------+-------+
| StudentID          | int(11)     | YES  |     | NULL    |       |
| Student_FirstName  | varchar(20) | YES  |     | NULL    |       |
| Student_LastName   | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber| varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID   | varchar(40) | YES  |     | NULL    |       |
| Age                | int(11)     | YES  |     | NULL    |       |
+--------------------+-------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

Whenever a default constraint is applied to the table's column, and the user has not specified the value to be inserted in it, then the default value which was specified while applying the default constraint will be inserted into that particular column.

Syntax to apply default constraint during table creation:

1. **CREATE TABLE** TableName (ColumnName1 datatype **DEFAULT** Value, ColumnName2 datatype,…., ColumnNameN datatype);

Example:

Create a student table and apply the default constraint while creating a table.

1. mysql> **CREATE TABLE** student(StudentID **INT**, Student_FirstName **VARCHAR**(20), St udent_LastName **VARCHAR**(20), Student_PhoneNumber **VARCHAR**(20), Student_Email_ ID **VARCHAR**(40) **DEFAULT** "anuja.k8@gmail.com");

```
mysql> CREATE TABLE student(StudentID INT, Student_FirstName VARCHAR(20), Student_LastName VARCHAR(20), Student_PhoneNumber VARCHAR(20), Student_Ema
il_ID VARCHAR(40) DEFAULT "anuja.k8@gmail.com");
Query OK, 0 rows affected (0.13 sec)
```

To verify that the default constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------------------+-------+
| Field               | Type        | Null | Key | Default             | Extra |
+---------------------+-------------+------+-----+---------------------+-------+
| StudentID           | int(11)     | YES  |     | NULL                |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL                |       |
| Student_LastName    | varchar(20) | YES  |     | NULL                |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL                |       |
| Student_Email_ID    | varchar(40) | YES  |     | anuja.k8@gmail.com  |       |
+---------------------+-------------+------+-----+---------------------+-------+
5 rows in set (0.01 sec)
```

Syntax to apply default constraint on an existing table's column:

1. **ALTER TABLE** TableName **ALTER** ColumnName **SET DEFAULT** Value;

Example:

Consider we have an existing table student. Later, we decided to apply the DEFAULT constraint on the student table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student **ALTER** Student_Email_ID **SET DEFAULT** "anuja.k8@g mail.com";

```
mysql> ALTER TABLE student ALTER Student_Email_ID SET DEFAULT "anuja.k8@gmail.com";
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the default constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+--------------------+-------+
| Field               | Type        | Null | Key | Default            | Extra |
+---------------------+-------------+------+-----+--------------------+-------+
| StudentID           | int(11)     | YES  |     | NULL               |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL               |       |
| Student_LastName    | varchar(20) | YES  |     | NULL               |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL               |       |
| Student_Email_ID    | varchar(40) | YES  |     | anuja.k8@gmail.com |       |
+---------------------+-------------+------+-----+--------------------+-------+
5 rows in set (0.01 sec)
```

7. CREATE INDEX

CREATE INDEX constraint is used to create an index on the table. Indexes are not visible to the user, but they help the user to speed up the searching speed or retrieval of data from the database.

Syntax to create an index on single column:

1. **CREATE INDEX** IndexName **ON** TableName (ColumnName 1);

Example:

Create an index on the student table and apply the default constraint while creating a table.

1. mysql> **CREATE INDEX** idx_StudentID **ON** student (StudentID);

```
mysql> CREATE INDEX idx_StudentID ON student(StudentID);
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the create index constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  | MUL | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.07 sec)
```

Syntax to create an index on multiple columns:

1. **CREATE INDEX** IndexName **ON** TableName (ColumnName 1, ColumnName 2, ColumnN ame N);

Example:

1. mysql> **CREATE INDEX** idx_Student **ON** student (StudentID, Student_PhoneNumber);

```
mysql> CREATE INDEX idx_Student ON student(StudentID, Student_PhoneNumber);
Query OK, 0 rows affected (0.16 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the create index constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  | MUL | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

Syntax to create an index on an existing table:

1. **ALTER TABLE** TableName **ADD INDEX** (ColumnName);

Consider we have an existing table student. Later, we decided to apply the DEFAULT constraint on the student table's column. Then we will execute the following query:

1. mysql> **ALTER TABLE** student **ADD INDEX** (StudentID);

```
mysql> ALTER TABLE student ADD INDEX(StudentID);
Query OK, 0 rows affected (0.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

To verify that the create index constraint is applied to the student table's column, we will execute the following query:

1. mysql> **DESC** student;

```
mysql> DESC student;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| StudentID           | int(11)     | YES  | MUL | NULL    |       |
| Student_FirstName   | varchar(20) | YES  |     | NULL    |       |
| Student_LastName    | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID    | varchar(40) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.06 sec)
```

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:

Operator	Description
+	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%';


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	106	Ashutosh Kumar	Patna	India

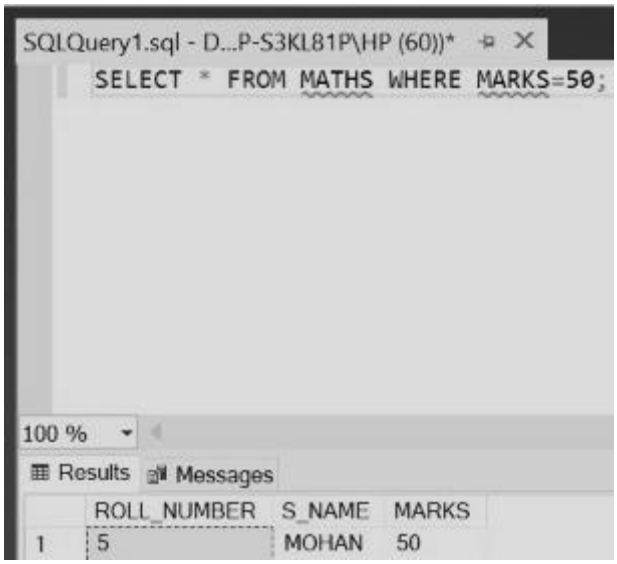
Comparison Operators

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:

Operator	Description
=	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;

Output:



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.

Operator	Description
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';

Output:


Results Messages

	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.

Operators	Description
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;

Output:

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 Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- NATURAL JOIN

Consider the two tables below as follows:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	xxxxxxxxxx	18
2	PRATIK	BIHAR	xxxxxxxxxx	19
3	RIYANKA	SILIGURI	xxxxxxxxxx	20
4	DEEP	RAMNAGAR	xxxxxxxxxx	18
5	SAPTARHI	KOLKATA	xxxxxxxxxx	19
6	DHANRAJ	BARABAJAR	xxxxxxxxxx	20
7	ROHIT	BALURGHAT	xxxxxxxxxx	18
8	NIRAJ	ALIPUR	xxxxxxxxxx	19

StudentCourse
The simplest Join is INNER JOIN.

A. INNER JOIN

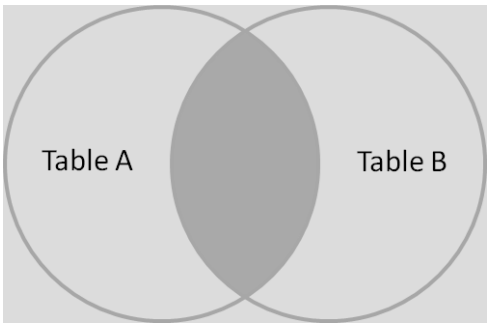
The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
***Note**: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*



Example Queries (INNER JOIN)

This query will show the names and age of students enrolled in different courses.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

B. LEFT JOIN

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT **OUTER JOIN.**

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

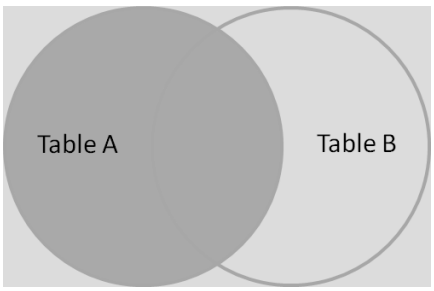


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
***Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*

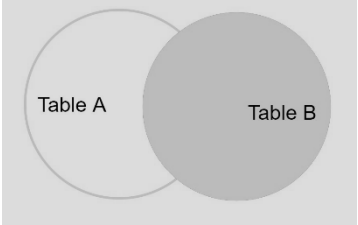
Example Queries(LEFT JOIN):

SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

C. RIGHT JOIN
RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.
Syntax:
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
***Note**: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*



Example Queries(RIGHT JOIN):
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

D. FULL JOIN
FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.

Syntax:
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;

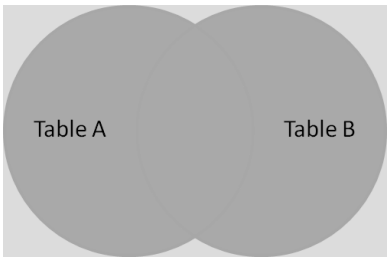


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
Example Queries(FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4
NULL	5
NULL	4

Left JOIN (Video)
Right JOIN (Video)
Full JOIN (Video)
SQL | JOIN (Cartesian Join, Self Join)

E. Natural join (?)

Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables.

Both table must have at list one common column with same column name and same data type.

The two table are joined using Cross join.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

Example:

Employee		
Emp_id	**Emp_name**	**Dept_id**
1	**Ram**	**10**
2	**Jon**	**30**
3	**Bob**	**50**

Department	
Dept_id	**Dept_name**
10	**IT**
30	**HR**
40	**TIS**

Query: Find all Employees and their respective departments.
Solution: (Employee) ? (Department)

Emp_id	Emp_name	Dept_id	Dept_id	Dept_name
1	**Ram**	**10**	**10**	**IT**
2	**Jon**	**30**	**30**	**HR**
Employee data			**Department data**	

SQL | Functions (Aggregate and Scalar Functions)
For doing operations on data SQL has many built-in functions, they are categorized in two categories and further sub-categorized in different seven functions under each category. The categories are:

1. **Aggregate functions:**
   These functions are used to do operations from the values of the column and a single value is returned.
   1. AVG()

2. COUNT()
3. FIRST()
4. LAST()
5. MAX()
6. MIN()
7. SUM()

2. **Scalar functions:**
   These functions are based on user input, these too returns single value.
   1. UCASE()
   2. LCASE()
   3. MID()
   4. LEN()
   5. ROUND()
   6. NOW()
   7. FORMAT()

Students-Table

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

Aggregate Functions

AVG(): It returns the average value after calculating from values in a numeric column.

Syntax:

SELECT AVG(column_name) FROM table_name;

Queries:

- Computing average marks of students.

SELECT AVG(MARKS) AS AvgMarks FROM Students;

Output:

AvgMarks
80

- Computing average age of students.

SELECT AVG(AGE) AS AvgAge FROM Students;

Output:

AvgAge
19.4

COUNT(): It is used to count the number of rows returned in a SELECT statement. It can't be used in MS ACCESS.

Syntax:

SELECT COUNT(column_name) FROM table_name;

Queries:

- Computing total number of students.

SELECT COUNT(*) AS NumStudents FROM Students;

Output:

NumStudents
5

- Computing number of students with unique/distinct age.

SELECT COUNT(DISTINCT AGE) AS NumStudents FROM Students;

Output:

NumStudents
4

FIRST(): The FIRST() function returns the first value of the selected column.
Syntax:
SELECT FIRST(column_name) FROM table_name;
Queries:

- Fetching marks of first student from the Students table.

SELECT FIRST(MARKS) AS MarksFirst FROM Students;
Output:

MarksFirst
90

- Fetching age of first student from the Students table.

SELECT FIRST(AGE) AS AgeFirst FROM Students;
Output:

AgeFirst
19

LAST(): The LAST() function returns the last value of the selected column. It can be used only in MS ACCESS.
Syntax:
SELECT LAST(column_name) FROM table_name;
Queries:

- Fetching marks of last student from the Students table.

SELECT LAST(MARKS) AS MarksLast FROM Students;
Output:

MarksLast
85

- Fetching age of last student from the Students table.

SELECT LAST(AGE) AS AgeLast FROM Students;
Output:

AgeLast
18

MAX(): The MAX() function returns the maximum value of the selected column.
Syntax:
SELECT MAX(column_name) FROM table_name;
Queries:

- Fetching maximum marks among students from the Students table.

SELECT MAX(MARKS) AS MaxMarks FROM Students;
Output:

MaxMarks
95

- Fetching max age among students from the Students table.

SELECT MAX(AGE) AS MaxAge FROM Students;
Output:

MaxAge
21

MIN(): The MIN() function returns the minimum value of the selected column.
Syntax:
SELECT MIN(column_name) FROM table_name;
Queries:

- Fetching minimum marks among students from the Students table.

SELECT MIN(MARKS) AS MinMarks FROM Students;
Output:

MinMarks
50

- Fetching minimum age among students from the Students table.

SELECT MIN(AGE) AS MinAge FROM Students;
Output:

MinAge
18

SUM(): The SUM() function returns the sum of all the values of the selected column.
Syntax:
SELECT SUM(column_name) FROM table_name;
Queries:

- Fetching summation of total marks among students from the Students table.

SELECT SUM(MARKS) AS TotalMarks FROM Students;
Output:

TotalMarks
400

- Fetching summation of total age among students from the Students table.

SELECT SUM(AGE) AS TotalAge FROM Students;
Output:

TotalAge
97

Scalar Functions

UCASE(): It converts the value of a field to uppercase.
Syntax:
SELECT UCASE(column_name) FROM table_name;
Queries:

- Converting names of students from the table Students to uppercase.

SELECT UCASE(NAME) FROM Students;
Output:

NAME
HARSH
SURESH
PRATIK
DHANRAJ
RAM

LCASE(): It converts the value of a field to lowercase.
Syntax:
SELECT LCASE(column_name) FROM table_name;
Queries:

- Converting names of students from the table Students to lowercase.

SELECT LCASE(NAME) FROM Students;
Output:

NAME
harsh
suresh
pratik
dhanraj
ram

MID(): The MID() function extracts texts from the text field.
Syntax:
SELECT MID(column_name,start,length) AS some_name FROM table_name;

specifying length is optional here, and start signifies start position ( ) starting from 1
Queries:

- Fetching first four characters of names of students from the Students table.

SELECT MID(NAME,1,4) FROM Students;
Output:

NAME
HARS
SURE
PRAT
DHAN
RAM

LEN(): The LEN() function returns the length of the value in a text field.
Syntax:
SELECT LENGTH(column_name) FROM table_name;
Queries:

- Fetching length of names of students from Students table.

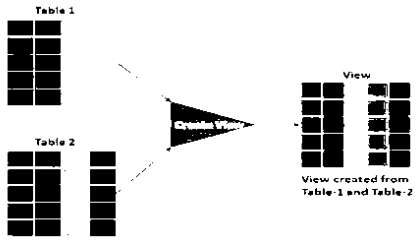
SELECT LENGTH(NAME) FROM Students;
Output:

NAME
5
6
6
7
3

Experiment 5

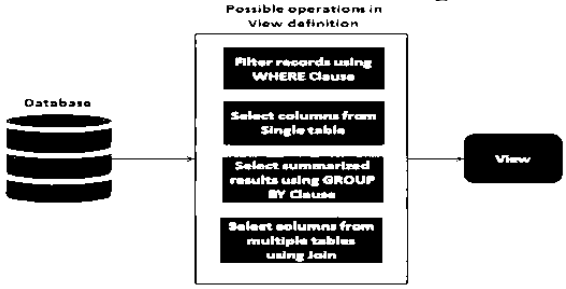
Creating different types of Views for tailored presentation of data

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.

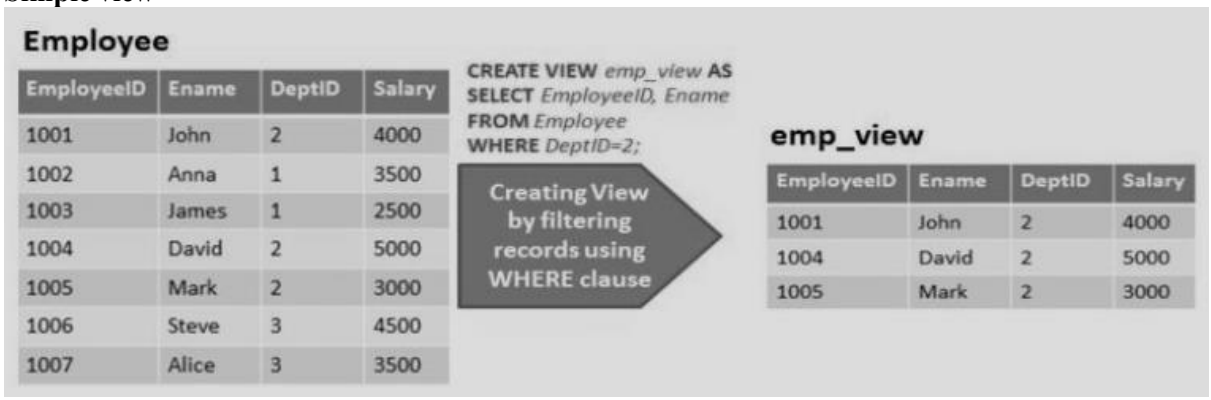
Complex View: A view based on multiple tables, which contain GROUP BY clause and functions.

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

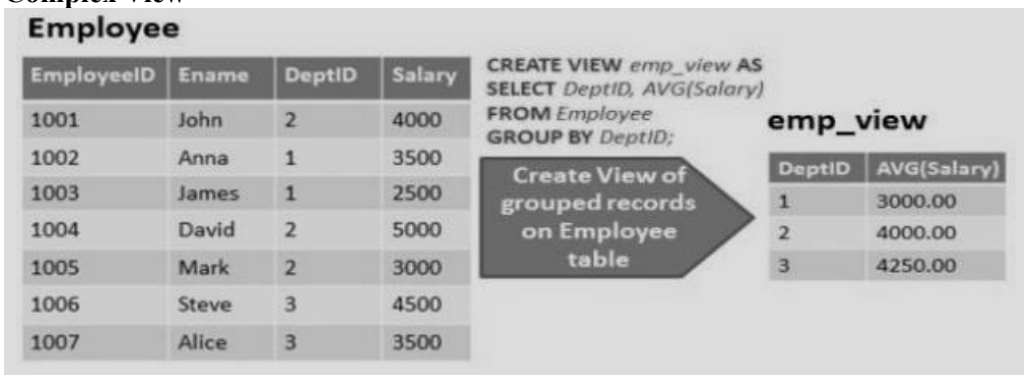
Materialized View: A view that stores the definition as well as data. It creates replicas of data by storing it physically.

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.

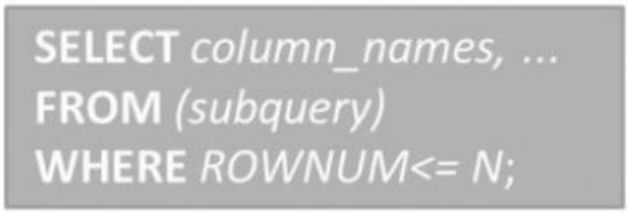
Simple view



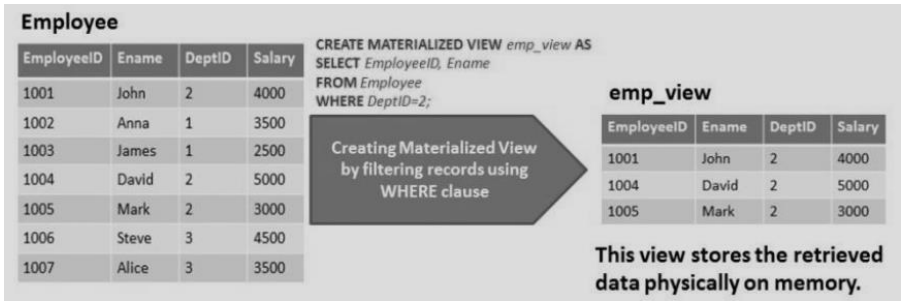
Complex View



Inline view



Materialized View



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

conditio

n then --

do

somethi

ng 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.Tamed system exceptionshese exceptions don't occur frequently.These exceptions have two parts code and an associated message.

The way to handle to 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. Thi 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:

Output:

CustomerName	Contact Name
Naveen	Tulasi

```
-- 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';
```

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;
```

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_lis
      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

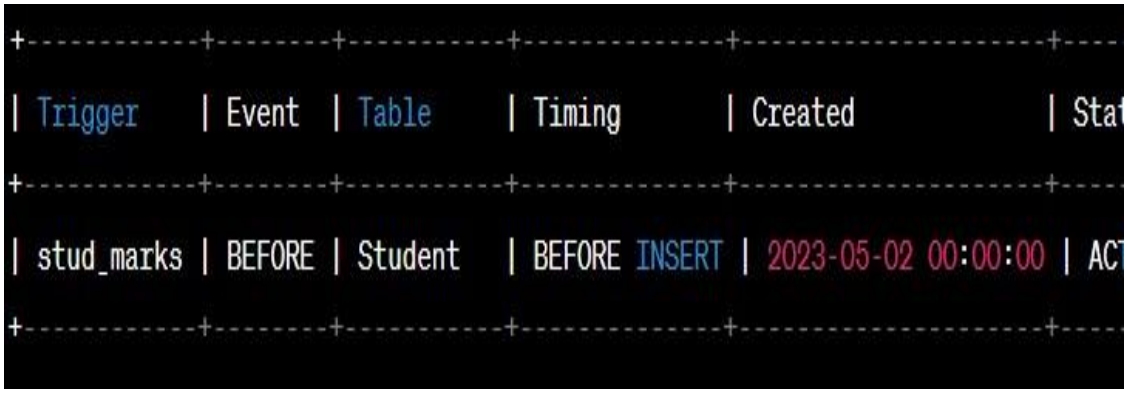
Student;

```
+-------+-------------+------+-----+---------+----------------+
| 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	Stat
stud_marks	BEFORE	Student	BEFORE INSERT	2023-05-02 00:00:00	AC

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:

```
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:

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