



Empowerment Through Quality Technical Education

**AJEENKYA DY PATIL SCHOOL OF ENGINEERING**

Dr. D. Y. Patil Knowledge City, Charholi (Bk), Lohegaon, Pune – 412 105

Website: <https://dypsoe.in/>

## **LAB MANUAL**

### **Database Management Systems Laboratory**

**(310246)**

**TE (COMP) 2019 COURSE**

**Course Coordinator**

**Dr. Nilesh Mali**

**Dr. Tushar phadtare**

**Prof. Minal Toley**

**DEPARTMENT OF**

**COMPUTER ENGINEERING**



## **Department of Computer Engineering**

**Vision:** “To achieve excellence in technical and socio-economic fields.”

**Mission:**

**M1:** To develop excellent learning centers through continuous up gradation in proximity with Academia, R&D centers and industries.

**M2:** To pursue research of local and global relevance.

**M3:** To encourage students to consider "start-ups" as a career option through Entrepreneurship Development Cell.

**M4:** Uplift and groom the learners to emerge as committed professionals.

## Table of Contents

### Contents

1. Guidelines to manual usage.....	4
2. Laboratory Objective.....	7
3. Laboratory Equipment/Software.....	7
4. Laboratory Experiment list.....	8
4.1. Experiment No. 1.....	10
4.2. Experiment No. 2.....	<b>Error! Bookmark not defined.</b>
4.3. Experiment No. 3.....	<b>Error! Bookmark not defined.</b>
4.4. Experiment No. 4.....	<b>Error! Bookmark not defined.</b>
4.5. Experiment No. 5.....	<b>Error! Bookmark not defined.</b>
4.6. Experiment No. 6.....	<b>Error! Bookmark not defined.</b>
5. Appendix.....	<b>Error! Bookmark not defined.</b>

## 1. Guidelines to manual usage

This manual assumes that the facilitators are aware of collaborative learning methodologies.

This manual will provide a tool to facilitate the session on Digital Communication modules in collaborative learning environment.

The facilitator is expected to refer this manual before the session.

### Icon of Graduate Attributes

<b>K</b> Applying Knowledge	<b>A</b> Problem Analysis	<b>D</b> Design & Development	<b>I</b> Investigation of problems
<b>M</b> Modern Tool Usage	<b>E</b> Engineer & Society	<b>E</b> Environment Sustainability	<b>T</b> Ethics
<b>T</b> Individual & Team work	<b>O</b> Communication	<b>M</b> Project Management & Finance	<b>I</b> Life-Long Learning

## Disk Approach- Digital Blooms Taxonomy



- 1: Remembering / Knowledge**
- 2: Comprehension / Understanding**
- 3: Applying**
- 4: Analyzing**
- 5: Evaluating**
- 6: Creating / Design**

**Course Name:** Database Management Systems Laboratory

**Course Code:** 310246

**Course Outcomes:** On completion of the course, learners will be able to

**CO1:** Design E-R Model for given requirements and convert the same into database tables.

**CO2:** Design schema in appropriate normal form considering actual requirements

**CO3:** Implement SQL queries for given requirements, using different SQL concepts

**CO4:** Implement PL/SQL Code block for given requirements

**CO5:** Implement NoSQL queries using MongoDB

**CO6:** Design and develop application considering actual requirements and using database concepts

**CO to PO Mapping:**

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
<b>CO1</b>	-	1	3	-	3	1	1	1	3	1	-	1
<b>CO2</b>	2	2	3	-	2	-	1	-	3	-	1	-
<b>CO3</b>	-	1	2	-	2	1	-	1	3	-	-	2
<b>CO4</b>	-	1	2	-	2	-	-	-	3	2	1	-
<b>CO5</b>	-	1	2	-	2	-	2	-	3	1	-	1
<b>CO6</b>	2	2	3	-	3	1	-	-	3	-	2	1

**CO to PSO Mapping:**

	PSO1	PSO2	PSO3
<b>CO1</b>	3	-	-
<b>CO2</b>	2	-	-
<b>CO3</b>	3	-	-
<b>CO4</b>	2	-	-
<b>CO5</b>	2	-	-
<b>CO6</b>	3	-	-

## **2. Laboratory Objective**

- To develop Database programming skills
- To develop basic Database administration skills
- To develop skills to handle NoSQL database
- To learn, understand and execute process of software application development

## **3. Laboratory Equipment/Software**

**Operating System recommended:** - 64-bit Open source Linux or its derivative **Programming tools recommended:** - MYSQL/Oracle, MongoDB, ERD plus, ER Win



#### 4. Laboratory Experiment list

Sr. No	Title
	<b>Prerequisite practical assignments or installation (if any)</b>
1	Installation of Mysql Server and Client
2	Installation of MongoDB
	<b>List of Assignments</b>
1	<p><b>ER Modeling and Normalization</b></p> <p>Decide a case study related to real time application in group of 2-3 students and formulate a problem statement for application to be developed. Propose a Conceptual Design using ER features using tools like ERD plus, ER Win etc. (Identifying entities, relationships between entities, attributes, keys, cardinalities, generalization, specialization etc.) Convert the ER diagram into relational tables and normalize Relational data model.</p>
2	Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index.
3	SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, delete with operators, functions, and set operator
4	Design at least 10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View.
5	<p><b>Unnamed PL/SQL code block:</b></p> <p>Use of Control structure and Exception handling is mandatory. Write a PL/SQL block of code for the following requirements: -</p> <p>Schema:</p> <p>1 Borrower (Rollin, Name, DateofIssue, NameofBook, Status)</p> <p>2 Fine(Roll_no,Date,Amt)</p> <ul style="list-style-type: none"> <li>o Accept roll_no &amp; name of book from user.</li> <li>o Check the number of days (from date of issue), if days are between 15 to 30 then fine amount will be Rs 5per day.</li> <li>o If no. of days&gt;30, per day fine will be Rs 50 per day &amp; for days less than 30, Rs. 5 per day.</li> </ul> <p>After submitting the book, status will change from I to R.</p>

	<ul style="list-style-type: none"> <li>o If condition of fine is true, then details will be stored into fine table.</li> </ul>
6	<p><b>PL/SQL Stored Procedure and Stored Function.</b></p> <p>Write a Stored Procedure namely proc_Grade for the categorization of student. If marks scored by students in examination is <math>\leq 1500</math> and <math>\geq 990</math> then student will be placed in distinction category if marks scored are between 989 and 900 category is first class, if marks 899 and 825 category is Higher Second Class Write a PL/SQL block for using procedure created with above requirement.</p> <p>Stud_Marks(name, total_marks) Result(Roll, Name, Class)</p>
7	<p><b>Cursors</b></p> <p>Write a PL/SQL block of code using parameterized Cursor that will merge the data available in the newly created table N_RollCall with the data available in the table O_RollCall. If the data in the first table already exist in the second table then that data should be skipped.</p>
8	<p>Database Trigger (All Types: Row level and Statement level triggers, Before and After Triggers).</p> <p>Write a database trigger on Library table. The System should keep track of the records that are being updated or deleted. The old value of updated or deleted records should be added in Library_Audit table. Frame the problem statement for writing Database Triggers of all types, in-line with above statement.</p>
9	<p><b>Database Connectivity</b></p> <p>Write a program to implement MySQL/Oracle database connectivity with any front end language to implement Database navigation operations (add, delete, edit etc.)</p>
10	<p><b>MongoDB Queries:</b></p> <p>Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators etc.).</p>
11	<p><b>MongoDB - Map reduces operations:</b></p> <p>Implement Map reduces operation with suitable example using MongoDB.</p>
12	<p><b>Database Connectivity:</b></p> <p>Write a program to implement MongoDB database connectivity with any front end language to implement Database navigation operations (add, delete, edit etc.)</p>

	<b>Content Beyond Syllabus</b>
1	
2	

## Experiment No. 1

### **Title : ER Modeling and Normalization**

Decide a case study related to real time application in group of 2-3 students and formulate a problem statement for application to be developed. Propose a Conceptual Design using ER features using tools like ERD plus, ER Win etc. (Identifying entities, relationships between entities, attributes, keys, cardinalities, generalization, specialization etc.) Convert the ER diagram into relational tables and normalize Relational data model.

**Course Objective:** To develop Database programming skills

**Software Required:** - Any tool for drawing ER diagram

### **Theory: -**

Entity Relationship (E R) Model : The Entity Relationship (ER) model is one of several high-level, or semantic, data models used in database design. The goal is to create a simple description of the data that closely matches how users and developers think of the data. A database can be modeled as : a collection of entities, relationship among entities.

An Entity is real-world object that exists and is distinguishable from other objects. A relationship is an association among several (Two or more) entities. Entities are represented by means of their properties, called attributes. An entity set is a set of entities of the same type that share the same properties. Each entity set has a Key. Each Attribute has a Domain.

### **Types of Attributes**

**Simple attribute** – Simple attributes are atomic values, which cannot be divided further.

For example, a Customer's ID number is an atomic value of 6 digits.

**Composite attribute** – Composite attributes are made of more than one simple attribute.

For example, a customer's complete name may have first-name, middle-initial and last-name.


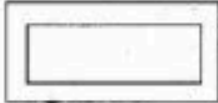


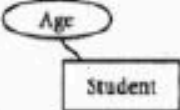
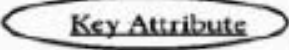

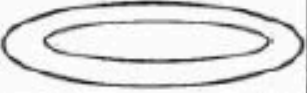
**Single-value attribute** – Single-value attributes contain single value.

For example – Customer\_ID, Social\_Security\_Number.

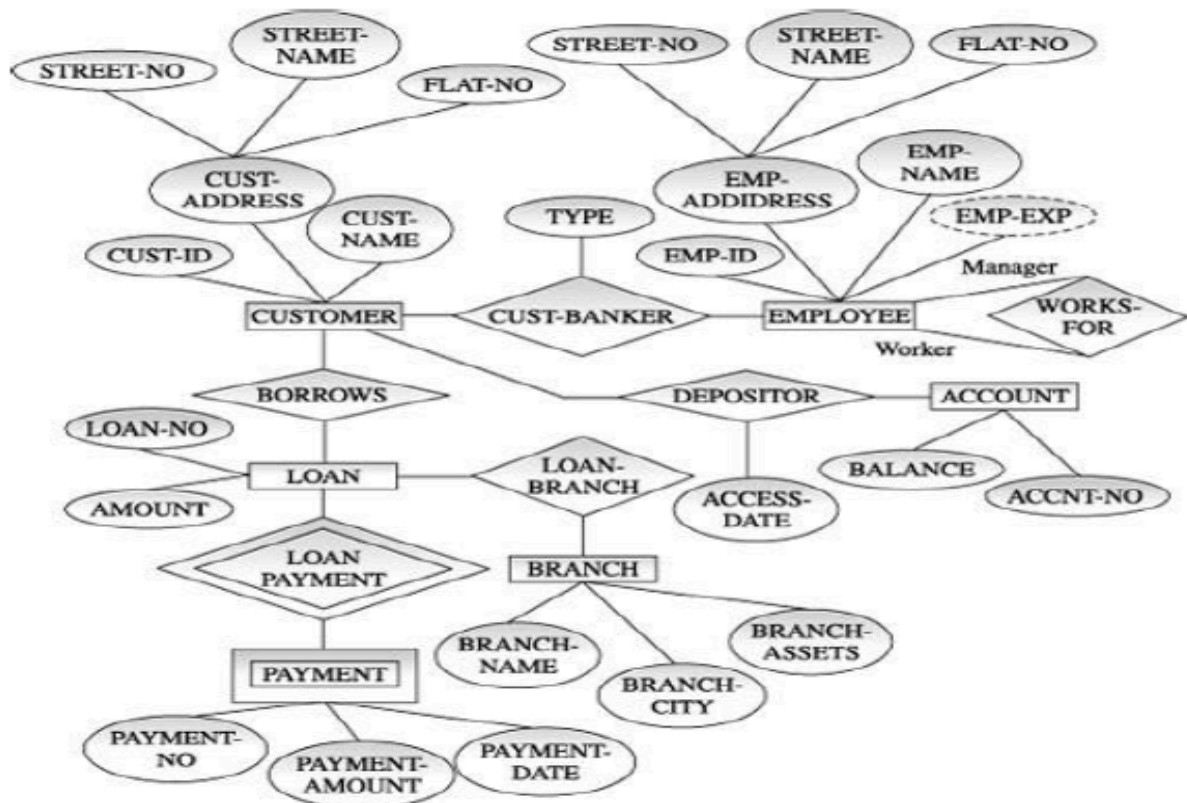
**Multi-value attribute** – Multi-value attributes may contain more than one values.

For example, a person can have more than one phone number, email\_address, etc.

**Derived attribute** – Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, age can be derived from date\_of\_birth.

ER Component	Description (how it is represented)	Notation
Entity – Strong	Simple rectangular box	
Entity – Weak	Double rectangular boxes	
Relationships	Rhombus symbol - Strong	
between Entities	Rhombus within rhombus – Weak	
Attributes	Ellipse Symbol connected to the entity	
Key Attribute for Entity	Underline the attribute name inside Ellipse	
Derived Attribute for	Dotted ellipse inside main ellipse Entity	
Multivalued Attribute	Double Ellipse for Entity	

## Example of ER diagram



## Relational Model

The relational model is a depiction of how each piece of stored information relates to the other stored information. It shows how tables are linked, what type of the links are between tables, what keys are used, what information is referenced between tables. It's an essential part of developing a normalized database structure to prevent repeat and redundant data storage.

### Different types of keys:

A **super key** is a set of one or more attributes which; taken collectively, allow us to identify uniquely an entity in the entity set.

A **primary key** is a candidate key (there may be more than one) chosen by the DB designer to identify entities in an entity set.

A super key may contain extraneous attributes, and we are often interested in the smallest super key. A super key for which no subset is a super key is called a **candidate key**.

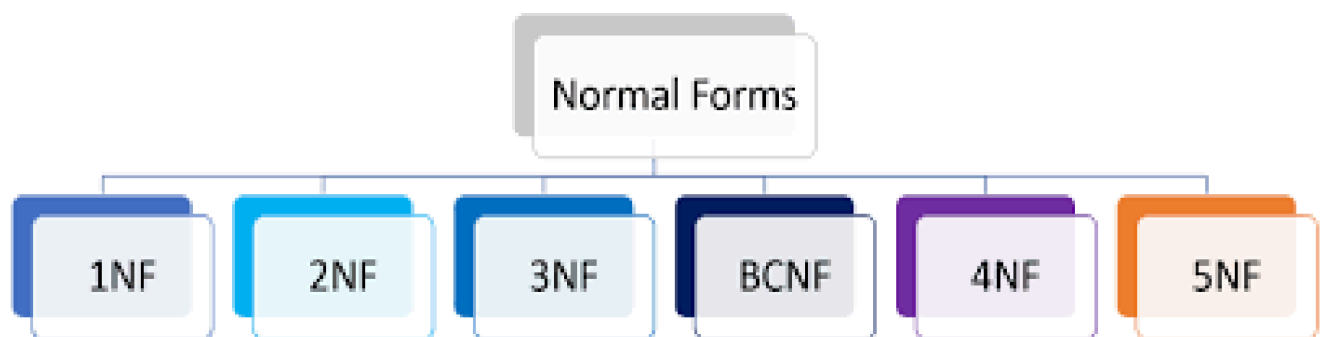
An entity does not possess sufficient attributes to form a primary key is called a **weak entity set**. One that does have a primary key is called a **strong entity set**.

A **foreign key** is a field in a relational table that matches the primary key column of another table. The foreign key can be used to cross-reference tables.

## Normalization

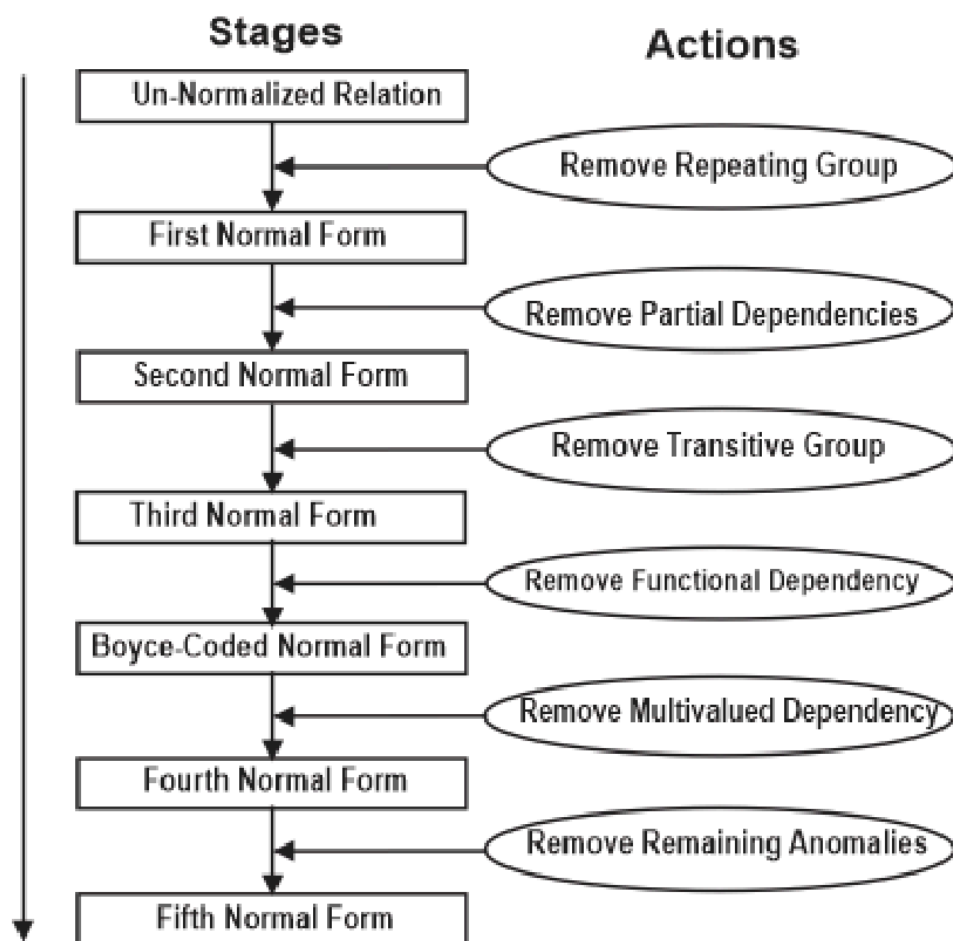
Database normalization is a technique for designing relational database tables to minimize duplication of information and, in so doing, to safeguard the database against certain types of logical or structural problems, namely data anomalies. In this we will write the normalization tables that is entities of “Roadway Travels.”

**Normalization:** In relational databases, normalization is a process that eliminates redundancy, organizes data efficiently; Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a tablet). Both of these are worthy goals as they reduce the amt of space a database consumes and ensure that data is logically stores.



The Normal Form: the database community has developed a series of guidelines for ensuring that databases are normalized. These are referred to as normal forms and are numbered from one (the lowest form to normalization, referred to as first form or 1NF) through five (fifth normal form or 5NF).

In practical applications, you'll often see 1NF, 2NF, and 3NF along with occasional 4NF. Fifth normal form is very rarely seen and won't be discussed in this article. It's important to point out that they are guidelines and guidelines only. Occasionally, it becomes necessary to stray from them to meet practical business requirements.



However, when variations take place, it's extremely important to evaluate any possible requirements they could have on your system and account for possible inconsistencies. That said, let's explore the normal form.

### Conclusion:

Students are able to design ER diagram and convert it into table with Normalized tables.



**Course Outcome:**

- Design E-R Model for given requirements and convert the same into database tables
- Design schema in appropriate normal form considering actual requirements

## Experiment No. 2

**Aim:** Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View and Index.

**Objective:**

1. Creating Tables using MySQL database
2. Creating View using MySQL database
3. Creating Index and Sequence on tables
4. Creating Synonym

**Software Required:** - Mysql

**Theory:**

**Data Definition Languages (DDL)**

Data definition language contains the commands which are used to create and destroy the databases and their objects (like table, view, index etc)

Following are commands under DDL

1. To create the database instance- CREATE
2. To alter the structure of database- ALTER
3. To rename database instances- RENAME
4. To drop the instances- DROP

**A) DDL Commands on database object: Table**

The table creation command requires the following details –

- Name of the table
- Name of the fields
- Definitions for

each field Syntax

Here is a generic SQL syntax to create a MySQL table –

```
CREATE TABLE table_name (column_name column_type);
```

Now, we will create the following table in the **data1** database.

```
root@host# mysql -u root -p
```

```
Enter password:*****
```

```
mysql> use data1;
```

```
Database changed
```

Here, a few items need explanation –

- Field Attribute **NOT NULL** is being used because we do not want this field to be NULL. So, if a user will try to create a record with a NULL value, then MySQL will raise an error.
- Field Attribute **AUTO\_INCREMENT** tells MySQL to go ahead and add the next available number to the id field.
- Keyword **PRIMARY KEY** is used to define a column as a primary key. You can use multiple columns separated by a comma to define a primary key.

### 1) Creating Table from Existing Table

Consider the existing table Employee

```
Create table table_name as select * from existing_table_name;
```

For Example

```
Create table newEmployee1 as select * from Employee;
```

```
// The newly created table newEmployee1 will include all the field and records as in Employee table
```

### 2) Creating Table having specific field but all the records from existing table

Syntax:

```
Create table table_name as select filed1, field2... from existing_table Name;
```

For example:

```
Create table newEmployee2 as select Employee_no,Employee_name from Employee;
```

## **B) Modifying Table**

ALTER TABLE query is used to modify structure of a table which is already exists in the database. We can add, delete, or modify column

- 1) Adding new column in a table Syntax:

```
ALTER TABLE table_name ADD Column-name datatype;
```

For example

```
Create table Employee1(Employee_no integer (3),  
Employee_name varchar (20),  
joining_date date,  
Salary integer (6));
```

```
ALTER TABLE Employee1 ADD Department varchar (15);
```

- 2) Dropping column from table

Syntax: ALTER TABLE table\_name DROP COLUMN Column\_name;

```
ALTER TABLE Employee1 DROP column Department;
```

- 3) Modifying Column of a table

Syntax: ALTER TABLE table\_name MODIFY COLUMN column\_name data\_type

```
ALTER TABLE Employee1 MODIFY COLUMN Employee_no varchar (4);
```

- 4) Renaming Table

Syntax: rename table current\_table to new\_table\_name;

```
Rename table Employee1 to Emptable;
```

- 5) Drop Table (User to delete the table) Syntax: drop table table\_name;

```
drop table Emptable;
```

## DDL Commands on Views

In SQL a view is virtual table containing the records of one or more tables based on SQL statement executed. Just like a real table, view contains rows and columns. You can add SQL functions, WHERE and JOIN statement to a view and presents the data as if the data were coming from one single table.

The changes made in a table get automatically reflected in original table and vice versa.

### A) Creating View

Consider we have existing table as Employee (Employee\_no, Employee\_name, joining\_date,salary)

- 1) Creating view having all records and fields from existing

table Syntax:

CREATE or replace VIEW view\_name as

SELECT Column1,Column2,... FROM table\_name Where Condition;

```
create table Employee2(Employee_no int,Employee_name  
varchar(10),joining_date date,salary int);
```

```
Create or replace view Emp_view1 as select * from Employee2;
```

- 2) Creating view having specific fields but all the records from

existing table Syntax:

Create or replace view view\_name as

Select field1,field2... from existing\_table\_name;

```
Create or replace view Emp_view2 as select Employee_no,Employee_name from Employee;
```

### B) Updating View

Updating query is used to update the records of view. Updation in view reflects the original table also.

Syntax:

UPDATE view\_name

Set field\_name=new\_Value where condition;

```
UPDATE Emp_view1 set Salary=73000 where Employee_no=101;
```

### C) Dropping View

Syntax: DROP view view\_name

```
DROP view Emp_view2;
```

## DDL Commands on Index

An Index is a pointer to data in a table. An index in a database is similar to the alphabetical index of a book presents at the end of book.

Indexes can be created or dropped with no effect on data.

### A) Creating Index

CREATE INDEX statement is used to create an INDEX. In this statement we have to mention name of the index, the table and column and whether the index is in ascending or descending order.

Syntax:

CREATE INDEX Syntax

Creates an index on a table. Duplicate values are

allowed: CREATE INDEX *index\_name*

ON *table\_name* (*column1*, *column2*, ...);

CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not

allowed: CREATE UNIQUE INDEX *index\_name*

ON *table\_name* (*column1*, *column2*, ...);

ASC or DESC is placed at end of column for ascending or descending order.

```
CREATE INDEX emp_ind1 on Emp(Emo_no ASC);
```

### B) Displaying Index

To display index information regarding table following query is used

Syntax: show index from table\_name;

```
Show index from Emp;
```

### C) Dropping Index

To drop index of a table following query is

used Syntax: Drop index index\_name on  
table\_name;

```
Drop index emp1_ind1 on emp;
```

### DDL Commands of Sequence:

A Sequence is a set of integers. Sequences are generated in order as per requirements. Sequences are used to create unique values for the rows.

Create table emp2(empno int auto\_increment,ename varchar(10),sal int, primary key(empno));

### DDL Commands on Synonym

A synonym is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects. You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

**Syntax:** In Oracle the commands is as follows

```
create synonym Cutomers for LongTablePrefix_Customers;
```

### **Program:**

```
//Create Table mysql>
```

```
use dbms;
```

```
Database changed
```

```
mysql> create table Employee(
```

```
-> employee_no INT NOT NULL AUTO_INCREMENT,
```

- > employee\_name varchar(10) NOT NULL,
- > joining\_date DATE,
- > salary INT,
- > PRIMARY KEY(employee\_no)
- > );

Query OK, 0 rows affected (0.34 sec)

1)Creating Table from Existing Table

mysql> Create table newEmployee1 as select \* from

Employee; Query OK, 0 rows affected (0.37 sec)

Records: 0 Duplicates: 0 Warnings: 0

2)Creating Table having specific field but all the records from existing table

mysql> Create table newEmployee2 as select Employee\_no,Employee\_name from Employee;

Query OK, 0 rows affected (0.30 sec)

Records: 0 Duplicates: 0 Warnings: 0

B)Modifying Table

1)Adding new column in a table

mysql> Create table Employee1(Employee\_no integer(3),

- > Employee\_name varchar(20),

- > joining\_date date,

- > Salary integer(6));

Query OK, 0 rows affected (0.35 sec)

mysql> ALTER TABLE Employee1 ADD Department

varchar(15); Query OK, 0 rows affected (0.41 sec)

Records: 0 Duplicates: 0 Warnings: 0

2)Dropping column from table

ALTER TABLE Employee1 DROP column

Department; Query OK, 0 rows affected (0.50 sec)

Records: 0 Duplicates: 0 Warnings: 0

3)Modifying Column of a table



```
mysql> ALTER TABLE Employee1 MODIFY COLUMN Employee_no varchar(4);
```

Query OK, 0 rows affected (0.89 sec)

Records: 0 Duplicates: 0 Warnings: 0

#### 4) Renaming Table

```
mysql> Rename table Employee1 to Emptable;
```

Query OK, 0 rows affected (0.38 sec)

#### 5) Drop Table (User to delete the table)

```
mysql> drop table
```

```
Emptable;
```

Query OK, 0 rows affected (0.20 sec)

Creating view

```
mysql> create table Employee2(Employee_no int,Employee_name
varchar(10),joining_date date,salary int);
```

Query OK, 0 rows affected (0.31 sec)

```
mysql> Create or replace view Emp_view1 as select * from
Employee2; Query OK, 0 rows affected (0.06 sec)
```

Updating View

```
mysql> insert into Emp_view1
```

```
values(101,'gajanan','2017-01-06',45000); Query OK, 1 row affected
(0.14 sec)
```

```
mysql> select * from Emp_view1;
```

```
+-----+-----+-----+-----+
| Employee_no | Employee_name | joining_date | salary |
+-----+-----+-----+-----+
|      101 | gajanan      | 2017-01-06 | 45000 |
+-----+-----+-----+-----+
+ 1 row in set (0.00 sec)
```

```
mysql> UPDATE Emp_view1 set Salary=73000 where
Employee_no=101; Query OK, 1 row affected (0.05 sec)
```

Rows matched: 1 Changed: 1 Warnings: 0

```
mysql> select * from Emp_view1;
```

```
+-----+-----+-----+-----+
| Employee_no | Employee_name | joining_date | salary |
+-----+-----+-----+-----+
|          101 | gajanan      | 2017-01-06 | 73000 |
+-----+-----+-----+-----+
+ 1 row in set (0.00 sec)
```

Creating Index

```
mysql> create table emp(empno int,name varchar(7),salary
int); Query OK, 0 rows affected (0.32 sec)
```

```
mysql> insert into emp
values(101,'gajanan',45000); Query OK, 1 row
affected (0.09 sec)
```

```
mysql> insert into emp values(1,'gajanan1',65000);
ERROR 1406 (22001): Data too long for column 'name' at row
1
```

```
mysql> insert into emp values(1,'gaju1',65000);
Query OK, 1 row affected (0.08 sec)
```

```
mysql> insert into emp
values(1011,'gaju2',3300); Query OK, 1 row
affected (0.03 sec)
```

```
mysql> CREATE INDEX emp1_ind1 on
emp(empno); Query OK, 0 rows affected (0.38 sec)
```

Records: 0 Duplicates: 0 Warnings: 0

mysql> show index from emp;

```
+-----+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-----+-----+-----+-----+-----+-----+-----+
| emp   |            | emp1_ind1 | 1 | empno      | A         | 3 | NULL | NULL | YES | BTREE      |         |               |
+-----+-----+-----+-----+-----+-----+-----+
+ 1 row in set (0.00 sec)
```

mysql>

### Creating Sequence

mysql> Create table emp2(empno int auto\_increment,ename varchar(10),sal int, primary key(empno)); Query OK, 0 rows affected (0.27 sec)

mysql> insert into emp2

values(1,'gajanan',20000); Query OK, 1 row affected (0.10 sec)

mysql> insert into emp2

values(default,'gajanan',20000); Query OK, 1 row affected (0.07 sec)

mysql> insert into emp2

values(default,'gajanan',20000); Query OK, 1 row affected (0.05 sec)

```
mysql> select * from emp2;
```

```
+-----+-----+-----+
```

```
| empno | ename | sal |
```

```
+-----+-----+-----+
```

```
|      1 | gajanan | 20000 |
```

```
|      2 | gajanan | 20000 |
```

```
|      3 | gajanan | 20000 |
```

```
+-----+-----+-----+
```

```
+ 3 rows in set (0.00 sec)
```

### **Conclusion:**

In this way, we developed SQL DDL statements which demonstrate the use of SQL objects such as Table, view, Index, Sequence, Synonym.

### **Outcome:**

Implement SQL queries for given requirements, using different SQL concepts

### **Experiment No. 3**

**Title:** SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, delete with operators, functions, and set operator

**Objective:**

- To list and understand DML commands and SQL Set Operations and functions
- Use database techniques such as SQL DML statements

**Software Required:** MySQL

**Theory:**

**DML command**

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

**1) *INSERT command***

Insert command is used to insert data into a table. Following is its general syntax,

**INSERT** into *table-name* values(data1,data2,...)

Lets see an example,

Consider a table **Student** with following fields.

S_id	S_Name	age
------	--------	-----

*INSERT into Student values(101,'Adam',15);*

The above command will insert a record into **Student** table.

S_id	S_Name	age
101	Adam	15

## **2) UPDATE command**

Update command is used to update a row of a table. Following is its general syntax,

**UPDATE** *table-name* set column-name = value *where* condition;

Lets see an example,

*update Student set age=18 where s\_id=102;*

### **Example to Update multiple columns**

*UPDATE Student set s\_name='Abhi',age=17 where s\_id=103;*

## **3) Delete command**

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row. Following is its general syntax,

**DELETE** from *table-name*;

### **Example to Delete all Records from a Table**

*DELETE from Student;*

The above command will delete all the records from **Student** table.

### **Example to Delete a particular Record from a Table**

Consider **Student** table

*DELETE from Student where s\_id=103;*

SQL Functions

SQL provides many built-in functions to perform operations on data. These functions are useful while performing mathematical calculations, string concatenations, sub-strings etc. SQL functions are divided into two categories,

- Aggregate Functions
- Scalar Functions

### **Aggregate Functions**

These functions return a single value after calculating from a group of values. Following are some frequently used Aggregate functions.

#### **1) AVG()**

Average returns average value after calculating from values in a numeric column. Its general Syntax is,

*SELECT **AVG**(column\_name) from table\_name e.g.*

*SELECT **avg**(salary) from Emp;*

#### **2) COUNT()**

Count returns the number of rows present in the table either based on some condition or without condition.

Its general Syntax is,

*SELECT **COUNT**(column\_name) from table-name;*

#### **Example using COUNT()**

Consider following **Emp** table

eid	name	age	salary
401	Anu	22	9000
402	Shane	29	8000

SQL query to count employees, satisfying specified condition is,

*SELECT **COUNT**(name) from Emp where salary = 8000;*

#### **3) FIRST()**

First function returns first value of a selected column

Syntax for FIRST function is,

*SELECT **FIRST**(column\_name) from table-name*

SQL query

*SELECT **FIRST**(salary) from Emp;*

#### **4) LAST()**

LAST return the return last value from selected column

Syntax of LAST function is,

**SELECT LAST(column\_name) from *table-name***

SQL query will be,

*SELECT LAST(salary) from emp;*

### **5) MAX()**

MAX function returns maximum value from selected column of the table.

Syntax of MAX function is,

**SELECT MAX(column\_name) from *table-name***

SQL query to find Maximum salary is,

*SELECT MAX(salary) from emp;*

### **6) MIN()**

MIN function returns minimum value from a selected column of the table. Syntax for MIN function is,

**SELECT MIN(column\_name) from *table-name***

SQL query to find minimum salary is,

*SELECT MIN(salary) from emp;*

### **7) SUM()**

SUM function returns total sum of a selected columns numeric values.

Syntax for SUM is,

*SELECT SUM(column\_name) from table-name* SQL

query to find sum of salaries will be, *SELECT*

*SUM(salary) from emp;*

### **Scalar Functions**

Scalar functions return a single value from an input value. Following are some frequently used Scalar Functions.

#### **1) UCASE()**

UCASE function is used to convert value of string column to Uppercase character.

Syntax of UCASE,

**SELECT UCASE(column\_name) from *table-name***

#### **Example of UCASE()**

SQL query for using UCASE is,

**SELECT UCASE(name) from emp;**

#### **2) LCASE()**

LCASE function is used to convert value of string column to Lowecase character.



### Syntax for LCASE is:

SELECT **LCASE**(column\_name) from *table-name*

### 3) **MID()**

MID function is used to extract substrings from column values of string type in a table.

### Syntax for MID function is:

SELECT **MID**(column\_name, start, length) from *table-name*

### 4) **ROUND()**

ROUND function is used to round a numeric field to number of nearest integer. It is used on Decimal point values. Syntax of Round function is,

SELECT **ROUND**(column\_name, decimals) from *table-name*

### Operators:

**AND** and **OR** operators are used with **Where** clause to make more precise conditions for fetching data from database by combining more than one condition together.

#### 1) **AND operator**

AND operator is used to set multiple conditions with *Where* clause.

#### **Example of AND**

SELECT \* from Emp WHERE salary < 10000 **AND** age > 25

#### 2) **OR operator**

OR operator is also used to combine multiple conditions with *Where* clause. The only difference between AND and OR is their behavior. When we use AND to combine two or more than two conditions, records satisfying all the condition will be in the result. But in case of OR, atleast one condition from the conditions specified must be satisfied by any record to be in the result.

#### **Example of OR**

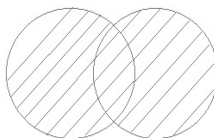
SELECT \* from Emp WHERE salary > 10000 **OR** age > 25

### Set Operation in SQL

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions.

#### 3) **Union**

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



### ***Example of UNION***

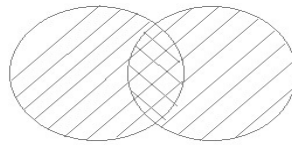
select \* from First

### **UNION**

select \* from second

#### ***4) Union All***

This operation is similar to Union. But it also shows the duplicate rows.



Union All query will be like,

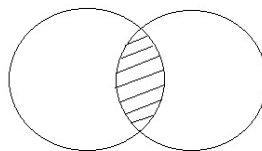
select \* from First

### **UNION ALL**

select \* from second

#### ***5) Intersect***

Intersect operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.



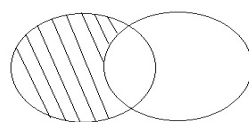
Intersect query will be, select

\* from First **INTERSECT**

select \* from second

#### ***6) Minus***

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



Minus query will be, select  
\* from First MINUS  
select \* from second

**Conclusion:** Hence we studied the DML commands and SQL set operators

**OUTCOMES:**

- Students will be able to write queries for given requirements, using SQL DML Commands
- Students will be able to write queries for given requirements, using SQL Set Operations and functions

## **Experiment No. 4**

**Title:**

**SQL Queries – all types of Join, Sub-Query and View:**

Write at least 10 SQL queries for suitable database application using SQL DML statements.

Note: Instructor will design the queries which demonstrate the use of concepts like all types of Join, Sub-Query and View

**Objective:**

Implement SQL queries for given requirements, using different SQL concepts

**Software Required:** - Mysql

**Theory:** -

**Join in SQL**

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables. **Join** Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

Types of Join:-

The following are the types of JOIN that we can use in SQL.

Inner

Outer

Left

Right

### **Cross JOIN or Cartesian Product**

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is, SELECT column-name-list from *table-name1*

### **CROSS JOIN**

*table-name2*;

### **INNER Join or EQUI Join**

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query.

**Inner Join Syntax is,**

SELECT column-name-list from *table-name1*

### **INNER JOIN**

*table-name2*

WHERE *table-name1*.column-name = *table-name2*.column-name;

### **Natural JOIN**

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

Natural Join Syntax is,

SELECT \*

from *table-name1*

## NATURAL JOIN

*table-name2*;

## Outer JOIN

Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

- Left Outer Join
- Right Outer Join
- Full Outer Join

### Left Outer Join

The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left** table and null for the **right** table's column.

Left Outer Join syntax is, SELECT column-name-list from *table-name1*

### LEFT OUTER JOIN

*table-name2*

on table-name1.column-name = table-name2.column-name; Left outer Join Syntax for **Oracle** is,

select column-name-list from *table-name1*, *table-name2*

on table-name1.column-name = table-name2.column-name(+);

**Left Outer Join** query will be,

SELECT \* FROM class LEFT OUTER JOIN class\_info ON (class.id=class\_info.id); The result table will look like,

ID	NAME	ID	ADDRESS
1	Abhi	1	DELHI
2	Adam	2	MUMBAI
3	Alex	3	CHENNAI
4	Anu	Null	Null
5	Ashish	Null	Null

### Right Outer Join

The right outer join returns a result table with the **matched data** of two tables then remaining rows of the **right table** and null for the **left** table's columns.

select column-name-list from *table-name1*

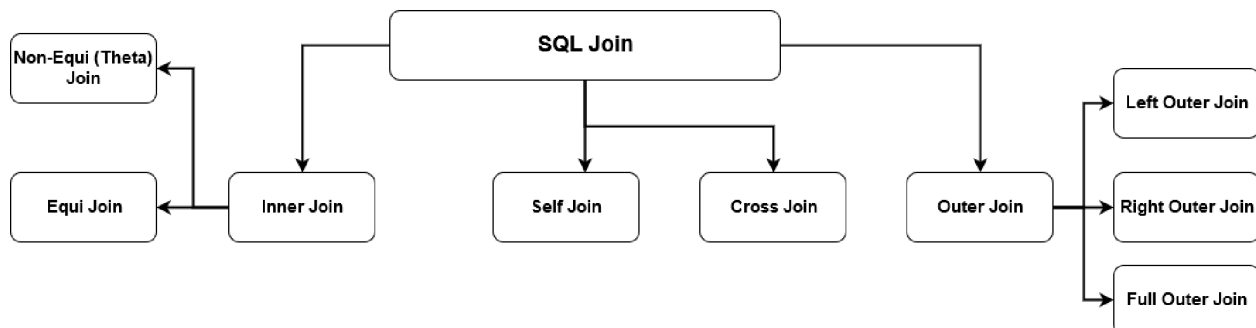
## RIGHT OUTER JOIN

*table-name2*

on table-name1.column-name = table-name2.column-name; Right outer Join Syntax for **Oracle** is,

select column-name-list from *table-name1*, *table-name2*

on table-name1.column-name(+) = table-name2.column-name;



## SQL Subquery

**Subquery** or **Inner query** or **Nested query** is a query in a query. SQL subquery is usually added in the WHERE Clause of the SQL statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database.

**Subqueries** are an alternate way of returning data from multiple tables.

Subqueries can be used with the following SQL statements along with the comparisons operators like =, <, >, >=, <= etc.

🎬 SELECT

🎬 INSERT

🎬 UPDATE

🎬 DELETE

## SQL Subquery Example:

1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators LIKE IN, NOT IN in the where clause. The query syntax would be like,

```
SELECT first_name, last_name, subject FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

Subquery output would be similar to:

first_name	last_name	subject
Shekar	Gowda	Badminton
Priya	Chandra	Chess

## SQL Subquery; INSERT Statement

3) Subquery can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths\_group'. INSERT INTO maths\_group(id, name)

```
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

## SQL Subquery; SELECT Statement

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order\_items table defined in the sql\_joins section.

```
select p.product_name, p.supplier_name, (select order_id from order_items where product_id = 101) as
order_id from product p where p.product_id = 101
```

product_name	supplier_name	order_id
Television	Onida	5103

## Correlated Subquery

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o WHERE o.product_id =
p.product_id);
```

### **Subquery Notes Nested Subquery**

1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle

### **Non-Correlated Subquery**

2) If a subquery is not dependent on the outer query it is called a non-correlated subquery

### **Subquery Errors**

3) Minimize subquery errors: Use drag and drop, copy and paste to avoid running subqueries with spelling and database typos. Watch your multiple field SELECT comma use, extra or too few getting SQL error message "Incorrect syntax".

### **SQL Subquery Comments**

Adding SQL Subquery comments are good habit (/\* your command comment \*/) which can save you time, clarify your previous work. results in less SQL headaches.

### **SQL Views**

A VIEW is a virtual table, through which a selective portion of the data from one or more tables can be seen. Views do not contain data of their own. They are used to restrict access to the database or to hide data complexity. A view is stored as a SELECT statement in the database.

DML operations on a view like INSERT, UPDATE, DELETE affects the data in the original table upon which the view is based.

### **The Syntax to create a sql view is**

```
CREATE VIEW view_name AS
SELECT column_list
FROM table_name [WHERE condition];
```



■ **view\_name** is the name of the VIEW.

■ The SELECT statement is used to define the columns and rows that you want to display in the view.

■ **For Example:** to create a view on the product table the sql query would be like, CREATE VIEW  
view\_product  
AS SELECT product\_id, product\_name FROM product;

### **Conclusion:**

Students are able to implement SQL queries all types of Join, Sub-Query and View for given

### **Outcome:**

**C310246.3** Implement SQL queries for given requirements, using different SQL concepts  
requirements, using different SQL concepts

## **Experiment No. 5**

**Title:** Unnamed PL/SQL code block: Use of Control structure and Exception handling is mandatory. Write a PL/SQL block of code for the following requirements: -

### **Schema:**

Borrower(Rollin, Name, DateofIssue, NameofBook, Status)

Fine(Roll\_no,Date,Amt)

- Accept roll\_no & name of book from user.
- Check the number of days (from date of issue), if days are between 15 to 30 then fine amount will be Rs 5per day.
- If no. of days>30, per day fine will be Rs 50 per day & for days less than 30, Rs. 5 per day.
- After submitting the book, status will change from I to R.
- If condition of fine is true, then details will be stored into fine table.

**Ojective:** Upon Completion of this assignment students should able to learn

1. Concept of PL/SQL block

2. Operation using PL/SQL block like Control structure and Exception Handling.

3. How to Write PL/SQL Block.

4. Understand the difference between PL/SQL and SQL

### Theory:

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

- PL/SQL is a completely portable, high-performance transaction-processing language.
  - PL/SQL provides a built-in, interpreted and OS independent programming environment.
  - PL/SQL can also directly be called from the command-line **SQL\*Plus interface**.
  - Direct call can also be made from external programming language calls to database.
  - PL/SQL's general syntax is based on that of ADA and Pascal programming language.
  - Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.
- Features of PL/SQL

PL/SQL has the following features –

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and

server pages.

The PL/SQL programs are divided and written in logical blocks of code. Each block consists of

three sub-parts –

S.N	Sections & Description
1	<b>Declarations</b> This section starts with the keyword <b>DECLARE</b> . It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	<b>Executable Commands</b> This section is enclosed between the keywords <b>BEGIN</b> and <b>END</b> and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a <b>NULL command</b> to indicate that nothing should be executed.
3	<b>Exception Handling</b> This section starts with the keyword <b>EXCEPTION</b> . This optional section contains <b>exception(s)</b> that handle errors in the program.

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

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>

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

- The PL/SQL Identifiers

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

characters.

- o By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.
- The PL/SQL Delimiters
- A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL

```

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

```

The other two data types will be covered in other chapters.

S.N	Category & Description
0	
1	<b>Scalar</b> Single values with no internal components, such as a <b>NUMBER</b> , <b>DATE</b> , or <b>BOOLEAN</b> .
2	<b>Large Object (LOB)</b> Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	<b>Composite</b> Data items that have internal components that can be accessed individually. For example, collections and records.
4	<b>Reference</b> Pointers to other data items.

#### PL/SQL Scalar Data Types and Subtypes

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

1	Numeric values on which arithmetic operations are performed.

2	<b>Character</b> Alphanumeric values that represent single characters or strings of characters.
3	<b>Boolean</b> Logical values on which logical operations are performed.
4	<b>Datetime</b> Dates and times.

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

#### PL/SQL Numeric Data Types and Subtypes

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

S.N	Data Type & Description
0	
1	<b>PLS_INTEGER</b> Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	<b>BINARY_INTEGER</b> Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	<b>BINARY_FLOAT</b> Single-precision IEEE 754-format floating-point number
4	<b>BINARY_DOUBLE</b> Double-precision IEEE 754-format floating-point number
5	<b>NUMBER(prec, scale)</b> Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	<b>DEC(prec, scale)</b> ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	<b>DECIMAL(prec, scale)</b> IBM specific fixed-point type with maximum precision of 38 decimal digits
8	<b>NUMERIC(pre, secale)</b> Floating type with maximum precision of 38 decimal digits

9	<b>DOUBLE PRECISION</b> ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	<b>FLOAT</b> ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	<b>INT</b> ANSI specific integer type with maximum precision of 38 decimal digits
12	<b>INTEGER</b> ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	<b>SMALLINT</b> ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	<b>REAL</b> Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration –

<pre> DECLARE num1 INTEGER; num2 REAL; PL/SQL procedure successfully completed PRECISION; BEGIN null ; END;</pre>	
S.N	Data Type & Description
0	
1	<b>CHAR</b> Fixed-length character string with maximum size of 32,767 bytes
2	<b>VARCHAR2</b> Variable-length character string with maximum size of 32,767 bytes
3	<b>RAW</b> Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL

4	<b>NCHAR</b> Fixed-length national character string with maximum size of 32,767 bytes
5	<b>NVARCHAR2</b> Variable-length national character string with maximum size of 32,767 bytes
6	<b>LONG</b> Variable-length character string with maximum size of 32,760 bytes
7	<b>LONG RAW</b> Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	<b>ROWID</b> Physical row identifier, the address of a row in an ordinary table
9	<b>UROWID</b> Universal row identifier (physical, logical, or foreign row identifier)

#### PL/SQL Boolean Data Types

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

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

–

SQL statements

Built-in SQL functions (such as **TO\_CHAR**)

PL/SQL functions invoked from SQL

statements Variable Declaration in PL/SQL

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

The syntax for declaring a variable is –

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

Where, *variable\_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any



user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below –

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

example

–

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

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

The **DEFAULT** keyword

The **assignment**

operator For example –

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

constraint. If

you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

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

```
DECLARE
a integer := 10; b
Value of c: 30
Value of f: 23.3333333333333333
```

PL/SQL procedure successfully completed.

```
N
c := a + b;
dbms_output.put_line('Value of c: ' || c);
f := 70.0/3.0;
dbms_output.put_line('Value of f: ' ||
```

**Local variables** – variables declared in an inner block and not accessible to outer blocks.

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

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

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

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

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

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

(For SQL statements, please refer to the [SQL tutorial](#))

```
CREATE TABLE
CUSTOMERS( ID INT NOT
NULL,
```

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

```
INSERT INTO CUSTOMERS
```

```
DECLARE
```

```
c_id customers.id%type := 1;
```

```
/
```

```
customers.id%type,
c_addr customers.address%type;
```

```
Customer Ramesh from Ahmedabad earns 2000
```

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

```
WHERE id = c_id;
```

```
dbms_output.put_li
```

```
25, 'Mumbai', 6500.00 );
('Customer ' || c_name || ' from ' || c_addr || ' earns ' ||
c_sal); END;
```

```
INSERT INTO CUSTOMERS
```

```
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik',
```

```
LOOP
```

```
Sequence of
statements;
```

```
INSERT INTO CUSTOMERS
```

```
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (6, 'Komal',
```

```
22, 'Mumbai', 1500.00);
```

```
DECLARE
```

```
x number :=
```

```
10; BEGIN
```

```
LOOP
```

```
dbms_output.put_line(x); x
```

```
:= x + 10;
```

```
IF x > 50 THEN
```

```
exit;
```

```

10
20
30
DECLARE
END LOOP;
-- after exit, control resumes here
-- dbms_output.put_line('After Exit x is: ');

```

```

10
20
30
40
50
After Exit x is: 60

```

```

WHILE condition LOOP
sequence_of_statements
END LOOP;

```

```

DECLARE
a number(2) :=
10; BEGIN

```

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17

```

```

FOR counter IN initial_value .. final_value LOOP

```

```

sequence_of_statements
; END LOOP;

```

PL/SQL procedure successfully completed.

The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

Next, the condition, i.e., *initial\_value .. final\_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.

After the body of the for loop executes, the value of the counter variable is increased or decreased.

The condition is now evaluated again. If it is TRUE, the loop executes and the process

repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

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

The *initial\_value* and *final\_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR.

The *initial\_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.

PL/SQL allows the determination of the loop range dynamically at run time. Example

```
DECLARE
  a number(2);
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
```

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

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

```
WHILE condition1
  LOOP
sequence_of_statements
1 WHILE condition2
  LOOP
sequence_of_statements
```

```
s2 END LOOP;
END LOOP;
```

PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

S.N	Control Statement & Description
1	<p><b><u>EXIT statement</u></b></p> <p>The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.</p>
2	<p><b><u>CONTINUE statement</u></b></p> <p>Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.</p>
3	<p><b><u>GOTO statement</u></b></p> <p>Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.</p>

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

System-defined exceptions

User-defined

exceptions Syntax for

Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many 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 >
DECLARE
c_id customers.id%type := &c_name;
END; /

BEGIN
SELECT name, address INTO c_name, c_addr FROM customers
WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
  WHEN no_data_found THEN dbms_output.put_line('No such
customer!');
  WHEN others THEN

```

```
No such customer!
```

ere is

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

customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in the **EXCEPTION block**.

## Raising Exceptions

```
DECLARE
    exception_name EXCEPTION;
BEGIN
    IF condition THEN
        RAISE exception_name;
    END IF;
EXCEPTION
WHEN exception_name THEN
    statement;
END;
```

l  
d  
option.  
e

## 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;
```

ser

```
DECLARE
c_id customers.id%type := &cc_id;
c_name customers.c_name%type;
WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
DBMS_OUTPUT.PUT_LINE ('Address: '||
```

```
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!
```

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

```
/
```

CurrentDate DATE;

```

NoOfDays Number(2);
Fine Number;
BEGIN
DBMS_OUTPUT.PUT_LINE('Enter Student Roll
Number'); Roll_No := &rollno;
DBMS_OUTPUT.PUT_LINE('Enter Book
Name'); BookName := &bookname;
CurrentDate := trunc(SYSDATE);
SELECT DateOfIssue into IssueDate FROM Borrower WHERE RollNo = Roll_No AND NameOfBook
=BookName;
SELECT trunc(SYSDATE) - IssueDate INTO NoOfDays from
dual; DBMS_OUTPUT.PUT_LINE('No of Days' || NoOfDays);
IF (NoOfDays > 30) THEN
Fine := NoOfDays * 50;
ELSIF (NoOfDays >= 15 AND NoOfDays <=30) THEN
Fine := NoOfDays * 5;
END IF;
IF FINE > 0 THEN
INSERT INTO Fine values (Roll_No, CurrentDate,
Fine); END IF;
UPDATE Borrower SET Status = 'R' WHERE
RollNo=Roll_No; END;

```

**Conclusion:** Thus we have studied how to write pl/sql code block with the help of control structures and exception handling.

**Outcome:**

- Students will be able define, declare, initialize and manage variable values.
- Students will be able to write and execute PLSQL unnamed blocks and use control structures.
- Students will be able to manage errors with exception handling

## Experiment No. 6

### Title of Assignment:

#### Named PL/SQL Block: PL/SQL Stored Procedure and Stored Function.

Write a Stored Procedure namely proc\_Grade for the categorization of student. If marks scored by students in examination is  $\leq 1500$  and marks  $\geq 990$  then student will be placed in distinction category if marks scored are between 989 and 900 category is first class, if marks  $\geq 899$  and  $\leq 825$  category is Higher Second Class.

Write a PL/SQL block to use procedure created with above requirement.

Stud\_Marks(name, total\_marks)

Result (Roll, Name, Class)

Note: Instructor will frame the problem statement for writing stored procedure and Function in line with above statement.

### Course Objective:

Implement PL/SQL Code block for given requirements

### Software Required: - Mysql

### Theory: -

Stored Procedures: A stored procedure or in simple a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists of declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.

- 1) IN-parameters
- 2) OUT-parameters
- 3) IN OUT-parameters

A procedure may or may not return any value. General Syntax to create a procedure is:



```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters] IS
```

```
Declaration section BEGIN
```

```
Execution section
```

```
EXCEPTION
```

```
Exception section END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedures: Example

The below example creates a procedure 'employer\_details' which gives the details of the employee.

```
CREATE OR REPLACE PROCEDURE employer_details
```

```
IS
```

```
CURSOR emp_cur IS
```

```
SELECT first_name, last_name, salary FROM emp_tbl; 5> emp_rec emp_cur%rowtype;
```

```
BEGIN
```

```
FOR emp_rec in sales_cur 8> LOOP
```

```
dbms_output.put_line(emp_cur.first_name || ' ' || emp_cur.last_name 10> || ' ' || emp_cur.salary);
```

```
END LOOP;
```

```
END;
```

There are two ways to execute a procedure.

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name. procedure\_name;

## **PL/SQL Functions**

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

**General Syntax to create a function is**

```
CREATE [OR REPLACE] FUNCTION function_name [parameters] RETURN return_datatype;  
IS  
Declaration_section  
BEGIN  
Execution_section  
Return return_variable;  
EXCEPTION
```

exception section Return return\_variable; END;

- 1) Return Type: The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
- 2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a function called "employer\_details\_func" similar to the one created in stored proc

```
CREATE OR REPLACE FUNCTION employer_details_func  
RETURN VARCHAR(20);  
IS  
emp_name VARCHAR(20);  
BEGIN  
SELECT first_name INTO emp_name 8> FROM emp_tbl WHERE empID = '100'; 9> RETURN  
emp_name;  
END;
```

In the example we are retrieving the 'first\_name' of employee with empID 100 to variable 'emp\_name'. The return type of the function is VARCHAR which is declared in line no 2.

The function returns the 'emp\_name' which is of type VARCHAR as the return value in line no 9. A function can be executed in the following ways.

1) Since a function returns a value we can assign it to a variable. employee\_name := employer\_details\_func;

If 'employee\_name' is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

3) In a PL/SQL Statements like,

```
dbms_output.put_line(employer_details_func);
```

This line displays the value returned by the function.

Program

```
-- Initially create table for student marks
```

```
create table Stud_Marks(  
    STUD_NAME varchar2(20),  
    TOTAL_MARKS number(5)  
);
```

```
-- Result table
```

```
create table Result(  
    STUD_NAME varchar2(20),  
    ROLL_NO number(5),  
    CLASS varchar2(20)  
);
```

```
--Creating a stored procedure
```

```
create or replace PROCEDURE PROC_GRADE1 AS
```

```
BEGIN
```

```
    FOR i IN (SELECT * FROM Stud_Marks)
```

```

        LOOP
            DBMS_OUTPUT.PUT_LINE('Student Name: ' || i.Stud_Name || ' Student Marks: ' ||
i.Total_Marks);
            IF i.Total_Marks <=1500 AND i.Total_Marks >=990 THEN
                INSERT INTO Result (STUD_NAME,CLASS) VALUES
(i.Stud_Name,'Distinction');
            ELSIF i.Total_Marks <=989 AND i.Total_Marks >=900 THEN
                INSERT INTO Result (STUD_NAME,CLASS) VALUES (i.Stud_Name,'First
Class');
            ELSIF i.Total_Marks <=825 AND i.Total_Marks >=899 THEN
                INSERT INTO Result (STUD_NAME,CLASS) VALUES (i.Stud_Name,'Higher
Second Class');
            END IF;
        END LOOP;
        COMMIT;
    END;

```

### **Course Outcome:**

C306.4 Implement PL/SQL Code block for given requirements

### **Conclusion:**

Students are able to PL/SQL Stored Procedure and Stored Function

## Experiment No. 7

**Title : Cursors :( All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor)**

Write a PL/SQL block of code using parameterized Cursor that will merge the data available in the newly created table N\_Roll Call with the data available in the table O\_RollCall. If the data in the first table already exist in the second table, then that data should be skipped.

### **Objective:**

Implement PL/SQL Code block for given requirements

**Software Required:** - Mysql

### **Theory: -**

#### **PL/SQL Cursor**

**Summary:** in this tutorial, we will introduce you to **PL/SQL cursor**. You will learn step by step how to use a cursor to loop through a set of rows and process each row individually.

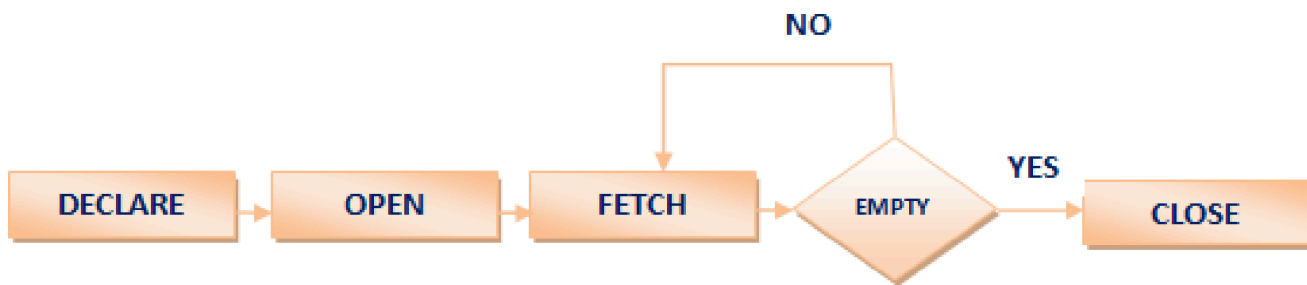
Introducing to PL/SQL Cursor

When you work with Oracle database, you work with a complete set of rows returned from an SQL SELECT statement. However the application in some cases cannot work effectively with the entire result set, therefore, the database server needs to provide a mechanism for the application to work with one row or a subset of the result set at a time. As the result, Oracle created PL/SQL cursor to provide these extensions.

A PL/SQL cursor is a pointer that points to the result set of an SQL query against database tables.

Working with PL/SQL Cursor

The following picture describes steps that you need to follow when you work with a PL/SQL cursor:



### ***The DECLARE Statement***

Using the DECLARE statement you can declare a cursor and associate it with the SELECT statement which fetches the desired records from a table. This SELECT statement associated with a cursor does not allow INTO clause.

Once you declare a cursor you can retrieve records from it using the FETCH statement. You need to make sure the cursor declaration precedes handler declarations. You can create use cursors in a single stored program.

### **Syntax**

Following is the syntax of the MySQL Cursor DECLARE Statement –

**DECLARE cursor\_name CURSOR FOR select\_statement;**

### **PL/SQL Cursor Attributes**

These are the main attributes of a PL/SQL cursor and their descriptions.

Attribute	Description
cursor_name%FOUND	returns <code>TRUE</code> if record was fetched successfully by cursor <code>cursor_name</code>
cursor_name%NOTFOUND	returns <code>TRUE</code> if record was not fetched successfully by cursor <code>cursor_name</code>
cursor_name%ROWCOUNT	returns the number of records fetched from the cursor <code>cursor_name</code> at the time we test <code>%ROWCOUNT</code> attribute
cursor_name%ISOPEN	returns <code>TRUE</code> if the cursor <code>cursor_name</code> is open

### **Explicit Cursors**

An **explicit cursor** is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.

**General Syntax for creating a cursor is as given below:**

***CURSOR cursor\_name IS select\_statement;***

- *cursor\_name* – A suitable name for the cursor.
- *select\_statement* – A select query which returns multiple rows.

**General Syntax to open a cursor is:**

- *OPEN cursor\_name;*

**General Syntax to fetch records from a cursor is:**

- *FETCH cursor\_name INTO record\_name;*

**OR**

- *FETCH cursor\_name INTO variable\_list;*

**General Syntax to close a cursor is:**

- *CLOSE cursor\_name;*

When a cursor is opened, the first row becomes the current row. When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row. On every fetch statement, the pointer moves to the next row. If you want to fetch after the last row, the program will throw an error. When there is more than one row in a cursor we can use loops along with explicit cursor attributes to fetch all the records.

## **FOR LOOP**

### **Cursor Syntax**

The syntax for the CURSOR FOR LOOP in Oracle/PLSQL is:

FOR record\_index in cursor\_name LOOP

{...statements...}

END LOOP;

Parameters or Arguments record\_index:The index of the record.

cursor\_name:The name of the cursor that you wish to fetch records from.

statements:The statements of code to execute each pass through the CURSOR FOR LOOP.

### Parameterized cursor:

PL/SQL Parameterized cursor pass the parameters into a cursor and use them in to query. PL/SQL

Parameterized cursor define only datatype of parameter and not need to define its length. Default values is assigned to the Cursor parameters. and scope of the parameters are locally.

Parameterized cursors are also saying static cursors that can passed parameter value when cursor are opened.Following example introduce the parameterized cursor. following emp\_information table,

EMP_NO	EMP_NAME
1	Forbs ross
2	marks jems
3	Saulin
4	Zenia Sroll

Cursor display employee information from emp\_information table whose emp\_no four (4). DECLARE

cursor c(no number) is select \* from emp\_information where emp\_no = no;

tmp emp\_information%rowtype; BEGIN

OPEN c(4);

FOR tmp IN c(4) LOOP

dbms\_output.put\_line('EMP\_No: '||tmp.emp\_no); dbms\_output.put\_line('EMP\_Name:

'||tmp.emp\_name); END Loop;

CLOSE c; END;

/



**SQL>@parameter\_cursor\_demo**

EMP\_No: 4

EMP\_Name: Zenia

**Program**

```
create database assi7;
```

```
use assi7;
```

```
show tables;
```

```
create table old_roll(roll int,name varchar(10));
```

```
create table new_roll(roll int,name varchar(10));
```

```
insert into old_roll values(4,'d');
```

```
insert into old_roll values(3,'bcd');
```

```
insert into old_roll values(1,'bc');
```

```
insert into old_roll values(5,'bch');
```

```
insert into new_roll values(2,'b');
```

```
insert into new_roll values(5,'bch');
```

```
insert into new_roll values(1,'bc');
```

```
select * from old_roll;
```

```
select * from new_roll;
```

```
delimiter $
```

```
create procedure roll_list()
```

```
begin
```

```
declare oldrollnumber int;
```

```
declare oldname varchar(10);
```

```
declare newrollnumber int;
```

```
declare newname varchar(10);
```

```
declare done int default false;
```

```
declare c1 cursor for select roll,name from old_roll;
```

```
declare c2 cursor for select roll,name from new_roll;
```

```
declare continue handler for not found set done=true;
```

```
open c1;
```

```
loop1:loop
```

```

fetch c1 into oldrollnumber,oldname;
if done then
leave loop1;
end if;
open c2;

loop2:loop
fetch c2 into newrollnumber,newname;
if done then
insert into new_roll values(oldrollnumber,oldname);
set done=false;
close c2;
leave loop2;
end if;
if oldrollnumber=newrollnumber then
leave loop2;
end if;
end loop;
end loop;
close c1;
end $
delimiter ;
call roll_list();
select * from new_roll;

```

### **Outcome:**

C306.4 Implement PL/SQL Code block for given requirements

**Conclusion:** We have implemented all types of Cursors successfully.

### **Activity to be Submitted by Students**

Write PL/SQL code to display Employee details using Explicit Cursors

Write PL/SQL code in Cursor to display employee names and salary.

## Experiment: 08

**Title:** Database Trigger (All Types: Row level and Statement level triggers, Before and After Triggers). Write a database trigger on Library table. The System should keep track of the records that are being updated or deleted. The old value of updated or deleted records should be added in Library\_Audit table.

**Objective:** Study of All Types: Row level and Statement level triggers, before and After Triggers

**Theory:**

### What is a Trigger?

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

### Syntax of Triggers

#### Syntax for Creating a Trigger

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW
AS n] [FOR EACH ROW]
WHEN
(condition)
BEGIN
```

--- sql  
statements END;

- *CREATE [OR REPLACE ] TRIGGER trigger\_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col\_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE ] TRIGGER trigger\_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table\_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column\_name or :new.column\_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected ( i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

**For Example:** The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product\_price\_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and

```
'product_price_history' table CREATE TABLE
product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

```
CREATE TABLE product
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );
```

2) Create the price\_history\_trigger and execute it.

```
CREATE or REPLACE TRIGGER
price_history_trigger BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
:old.product_name,
:old.supplier_name,
:old.unit_price);
END;
/
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product\_price\_history' table.

4) If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

### Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.

- 1) **Row level trigger** - An event is triggered for each row updated, inserted or deleted.
- 2) **Statement level trigger** - An event is triggered for each sql statement executed.

### PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

- 1) BEFORE statement trigger fires first.
- 2) Next BEFORE row level trigger fires, once for each row affected.
- 3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
- 4) Finally the AFTER statement level trigger fires.

**For Example:** Let's create a table 'product\_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product  
(Message varchar2(50),  
Current_Date number(32)  
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

- 1) **BEFORE UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER
Before_Update_Stat_product BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

- 2) **BEFORE UPDATE, Row Level:** This trigger will insert a record into the table 'product\_check' before each row is updated.

```
CREATE or REPLACE TRIGGER
Before_Upddate_Row_product BEFORE
UPDATE ON product
FOR EACH
ROW BEGIN
INSERT INTO product_check
Values('Before update row
level',sysdate); END;
/
```

- 3) **AFTER UPDATE, Statement Level:** This trigger will insert a record into the table 'product\_check' after a sql update statement is executed, at the statement level.

```

CREATE or REPLACE TRIGGER
After_Update_Stat_product AFTER
UPDATE ON
product BEGIN
INSERT INTO product_check
Values('After update, statement level',
sysdate); End;
/

```

4) **AFTER UPDATE, Row Level:** This trigger will insert a record into the table 'product\_check' after each row is updated.

```

CREATE or REPLACE TRIGGER
After_Update_Row_product AFTER
insert On
product FOR
EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row
level',sysdate); END;
/

```

Now lets execute a update statement on table product.

```

UPDATE PRODUCT SET unit_price =
800 WHERE product_id in (100,101);

```

Lets check the data in 'product\_check' table to see the order in which the trigger is fired. SELECT \* FROM product\_check;

**Output:**



Message

Current\_Date

-----

Before update, statement level

26-Nov-2008 Before update, row level

26-Nov-2008

After update, Row level

26-Nov-2008

Before update, row level

26-Nov-2008

After update, Row level

26-Nov-2008 After update, statement level

26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

### **Program:**

```
SQL> create table LIBRARY_AUDIT_STMT_LVL (STMT_TYPE
varchar(20), UPDATE_TS timestamp);
```

Table created.

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_BEFORE_DELETE
BEFORE DELETE ON LIBRARY_AUDIT_STMT_LVL
2 BEGIN
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL (STMT_TYPE, UPDATE_TS)
VALUES ('BEFORE DELETE', CURRENT_TIMESTAMP);
```

```
4      E
ND; 5 /
```

Trigger created.

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE
AFTER DELETE ON LIBRARY_AUDIT_STMT_LVL
2 BEGIN
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL(STMT_TYPE, UPDATE_TS)
VALUES('AFTER DELETE', CURRENT_TIMESTAMP);
4      E
ND; 5 /
```

Trigger created.

```
SQL> select * from
LIBRARY_AUDIT_STMT_LVL; no rows selected
SQL> create table LIBRARY(BOOK_STATUS_ID NUMBER,BOOKNAME VARCHAR(20));
```

Table created.

```
SQL> INSERT INTO LIBRARY VALUES(11,'DBMS');
```

1 row created.

```
SQL> INSERT INTO LIBRARY VALUES(22,'JAVA');
```

1 row created.

```
SQL> SELECT * FROM LIBRARY;
BOOK_STATUS_ID BOOKNAME
-----11 DBMS-----
                22 JAVA
```

```
SQL> DROP TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE;
```

Trigger dropped.

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE  
AFTER DELETE ON LIBRARY
```

```
2 BEGIN
```

```
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL(STMT_TYPE, UPDATE_TS)  
VALUES('AFTER DELETE', CURRENT_TIMESTAMP);
```

```
4     E
```

```
ND; 5 /
```

Trigger created.

```
SQL> DELETE FROM LIBRARY WHERE BOOK_STATUS_ID=22;
```

1 row deleted.

```
SQL> SELECT * FROM
```

```
LIBRARY_AUDIT_STMT_LVL; STMT_TYPE
```

```
UPDATE_TS
```

```
-----
```

```
-----
```

AFTER DELETE

31-AUG-17 05.13.44.843418 PM

```
SQL> DROP TRIGGER TRIGGER_STMT_LVL_BEFORE_DELETE;
```

Trigger dropped.

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_BEFORE_DELETE  
BEFORE DELETE ON LIBRARY BEGIN
```

```
2 INSERT INTO LIBRARY_AUDIT_STMT_LVL (STMT_TYPE, UPDATE_TS)  
VALUES ('BEFORE DELETE', CURRENT_TIMESTAMP);
```

```
3     E
```

```
ND; 4 /
```

Trigger created.

```
SQL> DELETE FROM LIBRARY WHERE BOOK_STATUS_ID=11;
```

1 row deleted.

```
SQL> SELECT * FROM
```

```
LIBRARY_AUDIT_STMT_LVL; STMT_TYPE
```

-----

```
UPDATE_TS
```

-----

```
AFTER DELETE
```

```
31-AUG-17 05.13.44.843418 PM
```

```
AFTER DELETE
```

```
31-AUG-17 05.20.54.175978 PM
```

```
BEFORE DELETE
```

```
31-AUG-17 05.24.25.852613 PM
```

```
stmt_type
```

-----

```
UPDATE_TS
```

-----

```
AFTER DELETE
```

```
31-AUG-17 05.24.25.852880 PM
```

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_BEFORE_DELETE
```

```
BEFORE DELETE ON LIBRARY_AUDIT_STMT_LVL
```

```

2 BEGIN
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL (STMT_TYPE, UPDATE_TS)
VALUES ('BEFORE DELETE'SQL> , CURRENT_TIMESTAMP);
4      E
ND; 5 /

```

Trigger created.

```

SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE
AFTER DELETE ON LIBRARY_AUDIT_STMT_LVL
2 BEGIN
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL(STMT_TYPE, UPDATE_TS)
VALUES('AFTER DELESQ> TE', CURRENT_TIMESTAMP);
4      E
ND; 5 /

```

Trigger created.

```

SQL> select * from
LIBRARY_AUDIT_STMT_LVL; no rows selected
SQL> create table LIBRARY(BOOK_STATUS_ID NUMBER,BOOKNAME VARCHAR(20));

```

Table created.

```

SQL> INSERT INTO LIBRARY VALUESSQL> S(11,'DBMS');
1 row created.

```

```

SQL> INSERT INTO LIBRARY VALUES(22,'JAVA');

```

1 row created.

```

SQL> SELECT * FROM LIBRARY;
BOOK_STATUS_ID BOOKNAME

```

-----

11 DBMS

22 JAVA

```
SQL> DROP TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE;
```

Trigger dropped.

```
SQL> CREATE OR REPLACE TRIGGER TRIGGER_STMT_LVL_AFTER_DELETE  
AFTER DELETE ON LIBRARY
```

```
2 BEGIN 5
```

```
3 INSERT INTO LIBRARY_AUDIT_STMT_LVL(STMT_TYPE, UPDATE_TS)  
VALUES('AFTER DELETE', CURRENT_TIMESTAMP);
```

```
4      E
```

```
ND; 5 /
```

Trigger created. 6 7 8 9 10 11 12 13

```
SQL> DELETE FROM LIBRARY WHERE BOOK_STATUS_ID=22;
```

1 row deleted.

```
SQL> SELECT * FROM
```

```
LIBRARY_AUDIT_STMT_LVL; STMT_TYPE
```

```
-----
```

```
UPDATE_TS
```

```
-----
```

```
AFTER DELET 14 E
```

```
31-AUG-17 05.13.44.843418 PM
```

```
SQL> DROP TRIGGER TRIGGER_STMT_LVL_BEFORE_DELETE;
```

Trigger dropped.

```
SQL> DELETE 15 FROM LIBRARY WHERE BOOK_STATUS_ID 16 = 17 1 18 1;
```

1 row deleted.

```
SQL> SELECT * FROM  
LIBRARY_AUDIT_STMT_LVL; STMT_TYPE
```

-----

UPDATE\_TS 19

\_\_\_\_\_20

\_\_\_\_\_ AFTER DELETE

31-AUG-17 05.13.44.843418 PM

21

AFTER DEL 22 ETE

31- 23 AUG-17 05.20.54.175978 PM

```
SQL> INSERT INTO LIBRARY VALUES(11,'DBMS');
```

1 row created.

```
SQL> CREAT 24 E OR REPLACE TRIGGER TRIGGER 25 _STMT_LVL_BEFORE_ 26 DELETE  
BEFORE DELETE ON LIBRARY BE 27 G 28 I 29 N
```

```
2 INSERT INTO LIBRARY_AUDIT_STMT_LVL (STMT_TYPE, UPDATE_TS) VALUES  
( 'BEFORE DELETE', CURRENT_TIMESTAMP(3));
```

```
31 3 END;
```

```
4 /
```

Trigger created.

```
SQL> DELETE FROM LIBRARY WHERE BOOK_STATUS_ID=11;
```

1 row deleted.

```
SQL> SE 32 LECT * FRO 33 M LIBRA 34
RY_AUDIT_STMT_LVL; STMT_TYPE
```

```
-----
```

```
UPDATE_TS
```

```
_____35_____
```

```
AFTER DELETE
```

```
31-AUG-17 05.13.44.843418 PM
```

```
AFTER DELETE
```

```
31-AUG 36 -17 05.20.54.1759 37 78 PM
```

```
BEFORE DELETE
```

```
31-AUG-17 05.24.25.8 38 5 39 2 40 613 PM
```

```
STMT_TYPE
```

```
-----
```

```
UPDATE_TS
```

```
_____41_____42
```

```
_____AFTER DELETE
```

```
31-AUG-17 05.24.25.852880 PM
```

**Conclusion:** Thus have successfully studied and implemented All Types: Row level and Statement level triggers, before and After Triggers

#### **OUTCOMES:**

- Students will be able to create and test DML trigger.
- Students will be able to identify the need and when to use triggers.



## Experiment: 9

### Title : Database Connectivity

**Aim:** Implement MYSQL/Oracle database connectivity with PHP/ python/Java Implement Database navigation operations (add, delete, edit,) using ODBC/JDBC.

**Objective:** Java Database Connectivity with MySQL

In Java, we can connect our Java application with the MySQL database through the Java code. JDBC (Java Database Connectivity) is one of the standard APIs for database connectivity, using it we can easily run our query, statement, and also fetch data from the database.

### Prerequisite to understand Java Database Connectivity with MySQL

1. You should have [MySQL](#) on your System.
2. You should have [JDK](#) on your System.
3. To set up the connectivity, the user should have MySQL Connector to the Java (JAR file), the 'JAR' file must be in class path while compiling and running the code of JDBC.

### *Steps to download MySQL Connector*

**Step 1** – Search for MySQL community downloads.

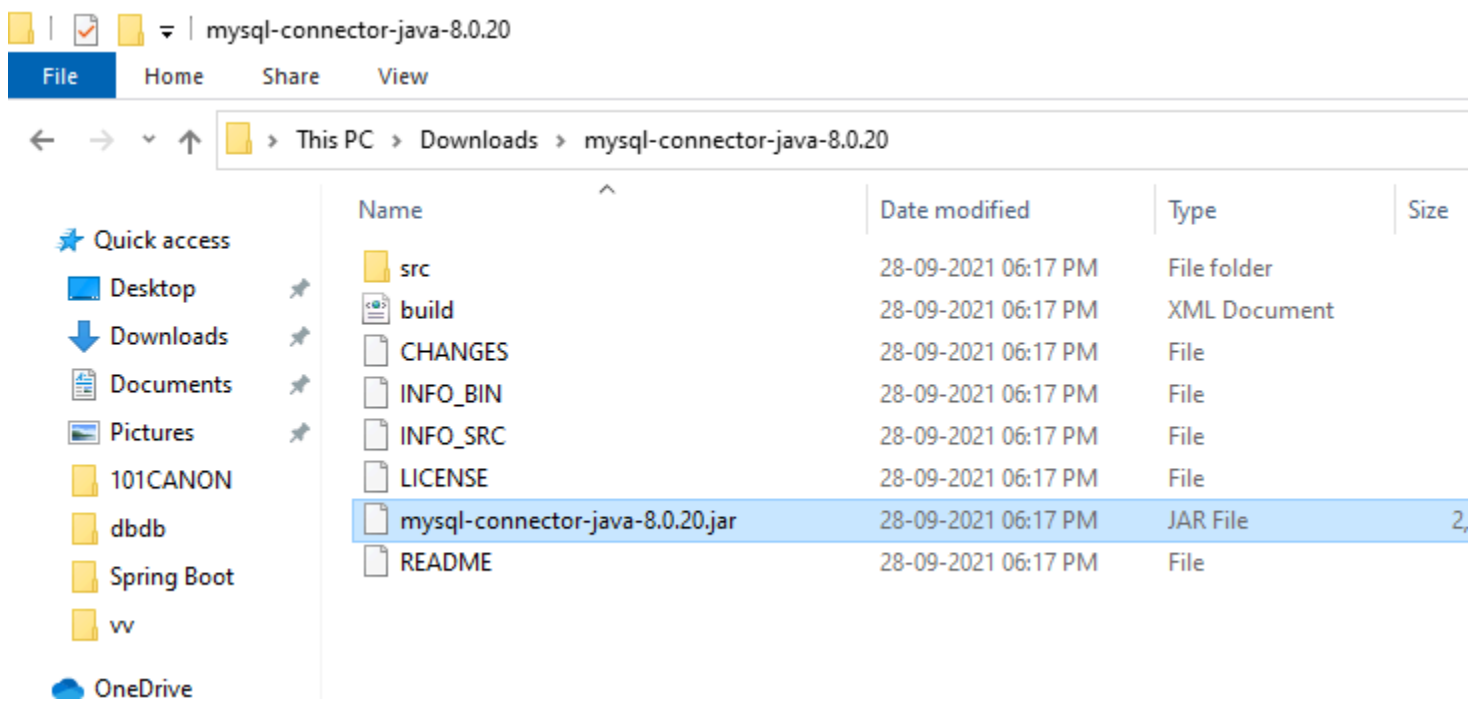
**Step 2** – Go to the **Connector/J**.

**Step 3** – Select the Operating System **platform-independent**.

**Step 4** – Download the zip file **Platform Independent (Architecture Independent), ZIP Archive**.

**Step 5** – Extract the zip file.

**Step 6** – Get the **mysql-connector-java-8.0.20.jar** file from the folder.



### *Setting up Database Connectivity with MySQL using [JDBC](#) code*

Users have to follow the following steps:

**Step 1** – Users have to create a database in MySQL (for example let the name of the database be ‘mydb’).

**Step 2** – Create a table in that database.

**Example:**

```
create table designation
(
  code int primary key auto_increment,
  title char(35) not null unique
);
```

This is MySQL code for creating a table.

**Step 3** – Now, we want to access the data of this table using Java database connectivity.

- Create a directory in your main drive (named gfg).
- Now, inside gfg created two more directories one named as ‘src’ and the other ‘lib’.

```
C:\WINDOWS\system32\cmd.exe

C:\>md gfg

C:\>cd gfg

C:\gfg>md lib

C:\gfg>md src

C:\gfg>dir
Volume in drive C has no label.
Volume Serial Number is DC5F-2A04

Directory of C:\gfg

14-12-2021  08:03    <DIR>          .
14-12-2021  08:03    <DIR>          ..
14-12-2021  08:03    <DIR>          lib
14-12-2021  08:03    <DIR>          src
               0 File(s)                0 bytes
               4 Dir(s)  51,382,362,112 bytes free

C:\gfg>
```

- Put the MySQL connector java jar file in the lib folder.

```
C:\gfg>cd lib

C:\gfg\lib>dir
Volume in drive C has no label.
Volume Serial Number is DC5F-2A04

Directory of C:\gfg\lib

14-12-2021  08:08    <DIR>          .
14-12-2021  08:08    <DIR>          ..
09-03-2020  11:19             2,385,601 mysql-connector-java-8.0.20.jar
               1 File(s)            2,385,601 bytes
               2 Dir(s)  51,388,645,376 bytes free
```

**Step 4 –** We will write connectivity code in the src folder, To write connectivity code user must know the following information:

- **Driver class:-** The driver class for connectivity of MySQL database “*com.mysql.cj.jdbc.Driver*”, after the driver has been registered, we can obtain a Connection instance that is connected to a particular database by calling *DriverManager.getConnection()*., in this method, we need to pass URL for connection and name and password of the database.
- **URL for Connection:-** The connection URL for the mysql database is `jdbc:mysql://localhost:3306/mydb` (‘mydb’ is the name of database).

Specify to the DriverManager which JDBC drivers to try to make Connections use below line:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

To get connection object use below line :

```
Connection connection=DriverManager.getConnection("URL in  
string","username","password");
```

To get more clarification follow the connectivity code below:

**Step 5** – In this src code, we will set up the connection and get all the data from the table. we have created the '*check.java*' file in the *src* folder.

**//Java program to set up connection and get all data from table**

```
import java.sql.*;
```

```
public class GFG {
```

```
    public static void main(String arg[])
```

```
    {
```

```
        Connection connection = null;
```

```
        try {
```

```
            // below two lines are used for connectivity.
```

```
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
            connection = DriverManager.getConnection(
```

```
                "jdbc:mysql://localhost:3306/mydb",
```

```
                "mydbuser", "mydbuser");
```

```
            // mydb is database
```

```
            // mydbuser is name of database
```

```
            // mydbuser is password of database
```

```
            Statement statement;
```

```
            statement = connection.createStatement();
```

```
            ResultSet resultSet;
```

```
            resultSet = statement.executeQuery(
```

```

        "select * from designation");
    int code;
    String title;
    while (resultSet.next()) {
        code = resultSet.getInt("code");
        title = resultSet.getString("title").trim();
        System.out.println("Code : " + code
                            + " Title : " + title);
    }
    resultSet.close();
    statement.close();
    connection.close();
}
catch (Exception exception) {
    System.out.println(exception);
}
} // function ends
} // class ends

```

Output of '*check.java*' file in the *src* folder:

```

C:\gfg\src>javac -classpath ..\lib\mysql-connector-java-8.0.20.jar;. Check.java
C:\gfg\src>java -classpath ..\lib\mysql-connector-java-8.0.20.jar;. Check
Code : 2 Title : CEO
Code : 3 Title : cook
Code : 1 Title : dancer
Code : 5 Title : manager
Code : 31 Title : null
Code : 8 Title : security
Code : 6 Title : waiter
C:\gfg\src>_

```

**Conclusion:** We successfully done the Java Database Connectivity with MySQL

## Experiment No : 10

**Title:** Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators)

**Objective:** Upon Completion of this assignment students should able to learn

1. Execute simple MongoDB Commands.
2. Execute simple MongoDB CRUD operation.
3. Study logical operators in MongoDB.

**Theory:**

### MongoDB Save() Method

The `db.collection.save()` method is used to updates an existing document or inserts a new document, depending on its document parameter.

### Syntax

The basic syntax of MongoDB `save()` method is shown below –

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

### Example

Following example will replace the document with the `_id` '5983548781331adf45ec7'.

```
>db.mycol.save(  
{  
  "_id" : ObjectId(5983548781331adf45ec7), "title":"Test1",
```

```
  "by":"JIT"  
}  
)  
>db.mycol.find()  
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New Topic",  
  "by":"DBMSTutor"  
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"
```

ain  
e,

as verified by the inserted document.

- save() performs an update with `upsert:true` since the document contains an `_id` field.

e.g. `db.invoice.save( { _id: 1001, inv_no: "I00001", inv_date: "10/10/2012", ord_qty: 200 } );`

### Logical Operators in MongoDB:

Name	Description
\$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
\$not	Inverts the effect of a query expression and returns documents that do not match the query expression.
\$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
\$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause

e.g.

1)     **AND** Queries With Multiple Expressions Specifying the Same Field

```
db.inventory.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )
```

This query will select all documents in the inventory collection where:

- the price field value is not equal to 1.99 and
- the price field exists.

2)     **AND** Queries With Multiple Expressions Specifying the Same Operator

```
db.inventory.find( {  
  
  $and : [  
  
    { $or : [ { price : 0.99 }, { price : 1.99 } ] },  
  
    { $or : [ { sale : true }, { qty : { $lt : 20 } } ] }  
  
  ]  
  
} )
```

This query will select all documents where:

- the price field value equals 0.99 or 1.99, and
- the sale field value is equal to true or the qty field value is less than 20

**Conclusion:** NoSQL queries using MongoDB

**Outcome:** Implement NoSQL queries using MongoDB



## Experiment No 11

### TITLE: AGGREGATION, INDEXING USING MONGODB

Implement aggregation and indexing with suitable example using MongoDB.

#### OBJECTIVES:

- To create and manage different types of indexes in MongoDB for query execution
- To learn the indexing and aggregation framework in MongoDB

#### SOFTWARE & HARDWARE REQUIREMENTS:

1. 64-bit Open source Linux or its derivative's
2. MongoDB Server

#### THEORY:

##### AGGREGATION

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In sql count (\*) and with group by is an equivalent of MongoDB aggregation.

##### The aggregate() Method

For the aggregation in mongodb you should use **aggregate()** method.

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

**\$group (aggregation)**

**\$group** :- Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain a `_id` field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the `$group`'s `_id` field.

`$group` does not order its output documents.

The `$group` stage has the following prototype form:

```
{ $group: { _id: <expression>, <field1>: { <accumulator1>: <expression1> }, ... } }
```

#### Aggregate Expressions:

Expression	Description	Example
<code>\$sum</code>	Sums up the defined value from all documents in the collection.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", num_tutorial: {\$sum: "\$likes"}}}])</code>
<code>\$avg</code>	Calculates the average of all given values from all documents in the collection.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", num_tutorial: {\$avg: "\$likes"}}}])</code>
<code>\$min</code>	Gets the minimum of the corresponding values from all documents in the collection.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", num_tutorial: {\$min: "\$likes"}}}])</code>
<code>\$max</code>	Gets the maximum of the corresponding values from all documents in the collection.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", num_tutorial: {\$max: "\$likes"}}}])</code>
<code>\$push</code>	Inserts the value to an array in the resulting document.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", url: {\$push: "\$url"}}}])</code>
<code>\$addToSet</code>	Inserts the value to an array in the resulting document but does not create duplicates.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", url: {\$addToSet: "\$url"}}}])</code>
<code>\$first</code>	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied " <code>\$sort</code> "-stage.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", first_url: {\$first: "\$url"}}}])</code>
<code>\$last</code>	Gets the last document from the source documents according to the grouping.	<code>db.mycol.aggregate([{\$group: {_id: "\$by_user", last_url: {\$last: "\$url"}}}])</code>

	Typically this makes only sense together with some previously applied "\$sort"-stage.	
--	---	--

### Example

**To find the total population of each state.**

```
db.AggreExample.aggregate([{$group:{"_id":"$State",totalpop:{$sum:"$pop"}}}])
```

**To find the Average population of each state**

```
db.AggreExample.aggregate([{$group:{"_id":"$State",avgPop:{$avg:"$pop"}}}])
```

Possible stages in aggregation framework are following:

**\$project:** Used to select some specific fields from a collection.

**\$match:** This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.

**\$group:** This does the actual aggregation as discussed above.

**\$sort:** Sorts the documents.

**\$skip:** With this it is possible to skip forward in the list of documents for a given amount of documents.

**\$limit:** This limits the amount of documents to look at by the given number starting from the current positions

Example:

**To Return States with Populations above 20,000**

```
db.AggreExample.aggregate([  
  { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
```

```
{ $match: { totalPop: { $gte: 20000 } } }  
])
```

## INDEXING

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require the MongoDB to process a large volume of data.

Indexes are special data structures that store a small portion of the data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

### Create an Index on a Single Field:

To create an index, use `ensureIndex()` or a similar method from your driver. The `ensureIndex()` method only creates an index if an index of the same specification does not already exist.

For example, the following operation creates an index on the `userid` field of the `records` collection:

```
db.records.ensureIndex({userid:1})
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

The created index will support queries that select on the field `userid`, such as the following:

### Create a unique index.

MongoDB allows you to specify a unique constraint on an index. These constraints prevent applications from inserting documents that have duplicate values for the inserted fields. MongoDB cannot create a unique index on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index.

To create a unique index, consider the following prototype:

**db.collection.createIndex( { a: 1 }, { unique: true } )**

ensureIndex() method also accepts list of options (which are optional), whose list is given below:

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is <b>false</b> .
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is <b>false</b> .
name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
dropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key. Specify true to create unique index. The default value is <b>false</b> .
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is <b>false</b> .
expireAfterSeconds	integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.
v	index version	The index version number. The default index version depends on the version of MongoDB running when creating the index.
weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.

default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is <b>english</b> .
language_override	string	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

### Unique Compound Index

You can also enforce a unique constraint on compound indexes, as in the following prototype:

```
db.collection.createIndex( { a: 1, b: 1 }, { unique: true } )
```

### Sparse indexes

If a document does not have a value for a field, the index entry for that item will be null in any index that includes it. Thus, in many situations you will want to combine the unique constraint with the sparse option. Sparse indexes skip over any document that is missing the indexed field, rather than storing null for the index entry. Since unique indexes cannot have duplicate values for a field, without the sparse option, MongoDB will reject the second document and all subsequent documents without the indexed field.

Consider the following prototype.

```
db.collection.createIndex( { a: 1 }, { unique: true, sparse: true } )
```

### Create a Hashed Index

Hashed indexes compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections. To create a hashed index, specify hashed as the value of the index key, as in the following example:

EXAMPLE: Specify a hashed index on `_id`

```
db.collection.createIndex( { _id: "hashed" } )
```

### Drop the index.

To modify the index, you must drop the index first.

```
db.orders.dropIndex(  
{"cust_id":1,"ord_date":-1,"items":1}  
)
```

The method returns a document with the status of the operation. Upon successful operation, the ok field in the returned document should specify a 1.

**CONCLUSION:** Thus we studied aggregation and indexing framework in MongoDB

**OUTCOMES:**

Students will be able to

- Create and manage different indexes in MongoDB
- Use aggregation and indexing framework in MongoDB for effective query execution

## Experiment No 12

**TITLE: Implement Map reduces operation with suitable example using MongoDB.**

### **OBJECTIVES:**

- To understand Map reduces operation
- Use MongoDB to implement the Map reduce techniques

### **SOFTWARE & HARDWARE REQUIREMENTS:**

1. 64-bit Open source Linux or its derivative
2. MongoDB Server

### **THEORY:**

**Map-reduce** is a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses **mapReduce** command for map-reduce operations. MapReduce is generally used for processing large data sets. In simple terms, the mapReduce command takes 2 primary inputs, the mapper function and the reducer function .

### **Working of Mapper and Reducer Function :**

MapReduce is a two-step approach to data processing. First you map, and then you reduce. The mapping step transforms the inputted documents and emits a key=>value pair (the key and/or value can be complex). Then, key/value pairs are grouped by key, such that values for the same key end up in an array. The reduce gets a key and the array of values emitted for that key, and produces the final result. The map and reduce functions are written in JavaScript. A Mapper will start off by reading a collection of data and building a Map with only the required fields we wish to process



and group them into one array based on the key. And then this key value pair is fed into a Reducer, which will process the values.

### MapReduce Command:

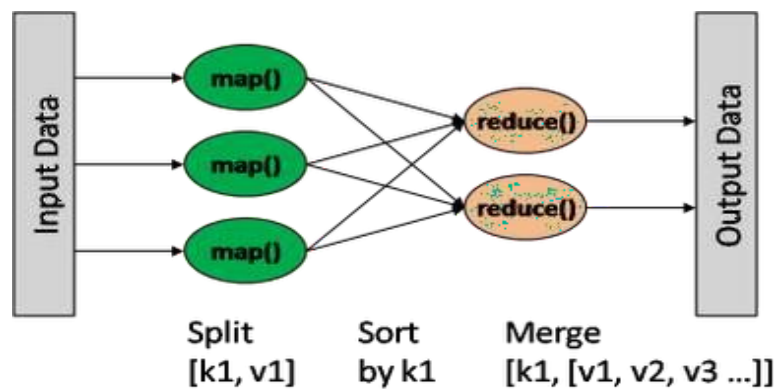
syntax of the basic mapReduce command:

```
db.collection.mapReduce(function() {emit(key,value);},
```

```
//map function function(key,values) {return
```

```
reduceFunction}, //reduce function
```

```
{out: collection, query: document, sort: document, limit: number})
```



The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values. MapReduce Command:

syntax of the basic mapReduce command:

```
db.collection.mapReduce(function(){emit(key,value);},//map function function(key,values)
```

```
{return reduceFunction}, //reduce function
```

```
{out: collection, query: document, sort: document, limit: number})
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.

In the above syntax:

- **map** is a javascript function that maps a value with a key and emits a key-value pair

- **reduce** is a javascript function that reduces or groups all the documents having the same key
- **out** specifies the location of the map-reduce query result
- **query** specifies the optional selection criteria for selecting documents
- **sort** specifies the optional sort criteria
- **limit** specifies the optional maximum number of documents to be returned

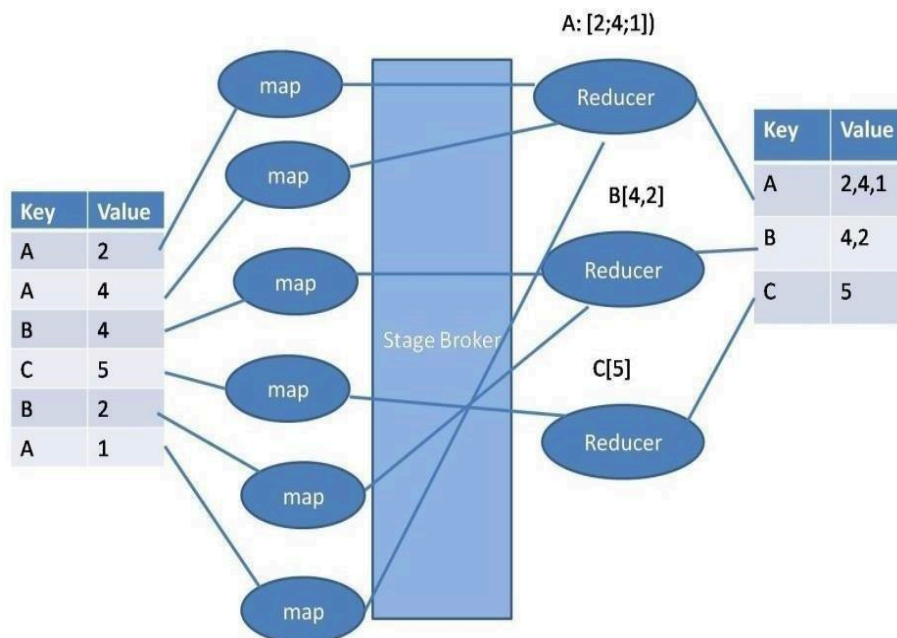
### Map Reduce Example

The below example is to retrieve the sum of total values related to particular key.

1. Insert data in *mapCollection*.

```
db.mapc.insert({key:"a", value:2})
```

```
db.mapc.insert({key:"a", value:4})
```



**CONCLUSION:** Thus we studied how to Implement Map reduces operation with suitable example using MongoDB.

**OUTCOMES:**

- Students will apply the concepts of Map reduce.

Students will be able to write queries for given requirements using Mongodb

## **Experiment No: 12**

### **Title: Database Connectivity**

Write a program to implement MongoDB database connectivity with any front end language to implement Database navigation operations (add, delete, edit etc.)

#### **OBJECTIVES:**

- To understand MongoDB database Connectivity
- Use Front End tool like Java to do MongoDB database Connectivity

#### **SOFTWARE & HARDWARE REQUIREMENTS:**

1. 64-bit Open source Linux or its derivative
2. MongoDB Server

#### **IMPLEMENTATION:**

```
import com.mongodb.client.MongoDatabase;  
  
import com.mongodb.client.model.Updates;  
  
import com.mongodb.client.MongoCollection;  
  
import com.mongodb.client.FindIterable;  
  
import org.bson.Document;  
  
import com.mongodb.MongoClient;
```

```
import java.util.Scanner;

import java.util.Iterator;

public class MongoJava {

    private static Scanner sc;

    public static void main (String args[]) {

        MongoClient<Document> collection=null;

        try {

            // Creating a Mongo client

            MongoClient mongo = new MongoClient( "localhost" , 27017 );

            // Accessing the database

            MongoDB database = mongo.getDatabase("Institute");

            System.out.println("Connected to the database successfully");

            //Creating a collection

            // database.createCollection("Students");

            //System.out.println("Collection Students created successfully");

            // Retrieving a collection

            collection = database.getCollection("Students");

            System.out.println("Collection Students selected successfully");

            sc = new Scanner(System.in);
```

```
int choice;

do {

    System.out.println("\n1. Insert Document \n2. Delete Document \n3. Update
Document \n4. Display All \n5. Display marks>75 \n6. Exit \nEnter your choice :");

    choice = sc.nextInt();

    switch (choice) {

        case 1: insert(collection);

        break;

        case 2: deletedoc(collection);

        break;

        case 3: updatedoc(collection);

        break;

        case 4: displayAll(collection);

        break;

        case 5: display(collection);

        break;

        case 6: System.out.println("Exiting Program");

        System.exit(0);

        break;

        default:
```

System.out.println(choice + " is not a valid Menu Option! Please Select Another.");

```
    }  
    }while(choice != 6 /*Exit loop when choice is 6*/);  
    }  
    catch(Exception ex){  
        ex.printStackTrace();  
    }  
}  
  
public static void insert(MongoCollection<Document> collection) {  
    System.out.println("Inserting document");  
    Document document = new Document();  
    Scanner sc=new Scanner(System.in);  
    System.out.println("Enter roll no:");  
    int rollno = sc.nextInt();  
    System.out.println("Enter name:");  
    String name= sc.next();  
    System.out.println("Enter class:");  
    String classs=sc.next();  
    System.out.println("Enter marks:");
```

```
int marks= sc.nextInt();

System.out.println("Enter technical interests:");

String techint=sc.next();

document.append("roll_no",rollno);

document.append("name",name);

document.append("class",classs);

document.append("marks",marks);

document.append("technical_interests",techint);

collection.insertOne(document);

System.out.println("Document inserted successfully");

}

public static void deletedoc(MongoCollection<Document> collection) {

    System.out.println("Deleting document");

    Document document = new Document();

    System.out.println("Enter roll no:");

    Scanner sc=new Scanner(System.in);

    int rollno= sc.nextInt();

    document.put("roll_no",rollno);

    collection.deleteOne(document);

    System.out.println("Document deleted successfully");

}
```

```
public static void updatedoc(MongoCollection<Document> collection) {  
    System.out.println("Updating document");  
  
    Document newdocument = new Document();  
    System.out.println("Enter roll no:");  
    Scanner sc=new Scanner(System.in);  
    int rollno = sc.nextInt();  
    newdocument.put("roll_no",rollno);  
    System.out.println("Enter New marks:");  
    int marks = sc.nextInt();  
    collection.updateOne(newdocument,Updates.set("marks", marks));  
}
```

```
public static void display(MongoCollection<Document> collection) {  
    System.out.println("Displaying documents in collection having marks  
greater than 75");  
    // Getting the iterable object  
    Document query=new Document();  
    query.append("marks",new Document().append("$gt", 75));  
    FindIterable<Document> iterDoc = collection.find(query);  
    // Getting the ite4rator
```



```
        Iterator it = iterDoc.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }

    public static void displayAll(MongoCollection<Document> collection) {
        System.out.println("Displaying all documents in collection");
        // Getting the iterable object
        FindIterable<Document> iterDoc = collection.find();
        int i = 1;
        // Getting the iterator
        Iterator it = iterDoc.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
            i++;
        }
    }
}
```

### **1.Display all records from Collection Students.**

```
1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
4
Displaying all documents in collection
Document({_id=616a95ff2d485b7e9ba970d7, roll_no=4, name=aaa, class=A, marks=88, technical_interests=zzz})
Document({_id=616a962b2d485b7e9ba970d8, roll_no=7, name=qqqq, class=B, marks=51, technical_interests=yyyy})
Document({_id=616a964f2d485b7e9ba970d9, roll_no=10, name=uuuu, class=D, marks=95, technical_interests=ppp})
Document({_id=616a966b2d485b7e9ba970da, roll_no=12, name=kkkk, class=A, marks=70, technical_interests=dddd})
Document({_id=616a96992d485b7e9ba970db, roll_no=15, name=iiii, class=C, marks=34, technical_interests=vvvv})
Document({_id=616a96d42d485b7e9ba970dc, roll_no=19, name=ssss, class=D, marks=74, technical_interests=hhhh})
Document({_id=616a96f62d485b7e9ba970dd, roll_no=20, name=tttt, class=B, marks=79, technical_interests=nnnn})
```

## 2. Insert records in collection students containing fields Stu\_RollNo,Stu\_Name, Class, Marks, Technical\_Interests.

```
1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
1
Inserting document
Enter roll no:
20
Enter name:
tttt
Enter class:
B
Enter marks:
79
Enter technical interests:
nnnn
Document inserted successfully
```

## 3. Delete the record of student having RollNo 10 from the collection.

```
1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
2
Deleting document
Enter roll no:
10
Document deleted successfully
```

```
1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
4
Displaying all documents in collection
Document({_id=616a95ff2d485b7e9ba970d7, roll_no=4, name=aaa, class=A, marks=88, technical_interests=zzz})
Document({_id=616a962b2d485b7e9ba970d8, roll_no=7, name=qqqq, class=B, marks=51, technical_interests=yyyy})
Document({_id=616a966b2d485b7e9ba970da, roll_no=12, name=kkkk, class=A, marks=70, technical_interests=dddd})
Document({_id=616a96992d485b7e9ba970db, roll_no=15, name=iiii, class=C, marks=34, technical_interests=vvvv})
Document({_id=616a96d42d485b7e9ba970dc, roll_no=19, name=ssss, class=D, marks=74, technical_interests=hhhh})
Document({_id=616a96f62d485b7e9ba970dd, roll_no=20, name=tttt, class=B, marks=79, technical_interests=nnnn})
```

**4. Update the record of student with RollNo 20 by changing Marks from 50 to 60.**

```
1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
3
Updating document
Enter roll no:
20
Enter New marks:
60

1. Insert Document
2. Delete Document
3. Update Document
4. Display All
5. Display marks>75
6. Exit
Enter your choice :
4
Displaying all documents in collection
Document({_id=616a95ff2d485b7e9ba970d7, roll_no=4, name=aaa, class=A, marks=88, technical_interests=zzz})
Document({_id=616a962b2d485b7e9ba970d8, roll_no=7, name=qqqq, class=B, marks=51, technical_interests=yyyy})
Document({_id=616a966b2d485b7e9ba970da, roll_no=12, name=kkkk, class=A, marks=70, technical_interests=dddd})
Document({_id=616a96992d485b7e9ba970db, roll_no=15, name=iiii, class=C, marks=34, technical_interests=vvvv})
Document({_id=616a96d42d485b7e9ba970dc, roll_no=19, name=ssss, class=D, marks=74, technical_interests=hhhh})
Document({_id=616a96f62d485b7e9ba970dd, roll_no=20, name=tttt, class=B, marks=60, technical_interests=nnnn})
```

**CONCLUSION:** Hence we successfully implemented MongoDB database connectivity with Java Frontend

**OUTCOMES:**

- Students will apply the front end Tool like Java
- Students will be able to write program for given requirements using Mongoddb and Java.