Dr. D. Y. Patil Pratishthan's

DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University

Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566

Website : www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

---

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL

Database Management System Laboratory

(Third Year Engineering)

Semester – I

Teaching Faculties : Dr. P. P. Halkarnikar
Dr. Rashmi Deshpande
Ms. Akanksha Kulkarni

Table of Contents

Assignment No - A1

- **Title:** Study and Draw ER Modelling diagram along with normalization using ERDwin/ERD plus for selected problem statement.
- **Objective:** Study and understand modern tools for ER diagrams.
- **Outcome:** Ability to use modern tools like ERD Plus and ER Win for representing logical model of DBMS.
- **Problem Statement:** Draw ER Modelling digram along with normalization using ERDwin/ERD plus for university database.

The Entity-Relationship Model:

- The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database.
- The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema.
- Because of this usefulness, many database-design tools draw on concepts from the E-R model.
- The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes, which we study first.

Entity Sets:

- An entity set is a set of entities of the same type that share the same properties, or attributes.
- The set of all people who are instructors at a given university, for example, can be defined as the entity set instructor.
- Similarly, the entity set student might represent the set of all students in the university.

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Figure 7.1 Entity sets *instructor* and *student*.

Attributes:

- An entity is represented by a set of **attributes**.
- Attributes are descriptive properties possessed by each member of an entity set.
- The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute.
- Possible attributes of the *instructor* entity set are *ID*, *name*, *dept name*, and *salary*.

Simple and composite attributes:

- **Composite** attributes, can be divided into subparts (that is, other attributes).
- For example, an attribute *name* could be structured as a composite attribute consisting of *first name*, *middle initial*, and *last name*.

Single-valued and multivalued:

- The attributes in our examples all have a single value for a particular entity.
- For instance, the student ID attribute for a specific student entity refers to only one student ID.
- Such attributes are said to be single valued.
- There may be instances where an attribute has a set of values for a specific entity.

Derived attribute:

- The value for this type of attribute can be derived from the values of other related attributes or entities.
- For instance, let us say that the *instructor* entity set has an attribute *students advised*, which represents how many students an instructor advises.
- We can derive the value for this attribute by counting the number of *student* entities associated with that instructor.
- As another example, suppose that the *instructor* entity set has an attribute *age* that indicates the instructor's age.
- If the *instructor* entity set also has an attribute *date of birth*, we can calculate *age* from *date of birth* and the current date.
- Thus, *age* is a derived attribute. In this case, *date of birth* may be referred to as a *base* attribute, or a *stored* attribute.
- The value of a derived attribute is not stored but is computed when required.

Relationship Set:

- A **relationship** is an association among several entities. For example, we can define a relationship *advisor* that associates instructor Katz with student Shankar.
- This relationship specifies that Katz is an advisor to student Shankar.
- A **relationship set** is a set of relationships of the same type.
- A relationship may also have attributes called **descriptive attributes**.
- Consider a relationship set *advisor* with entity sets *instructor* and *student*.
- We could associate the attribute *date* with that relationship to specify the date when an instructor became the advisor of a student.
- The *advisor* relationship among the entities corresponding to instructor Katz and student Shankar has the value "10 June 2007" for attribute *date*, which means that Katz became Shankar's advisor on 10 June 2007.

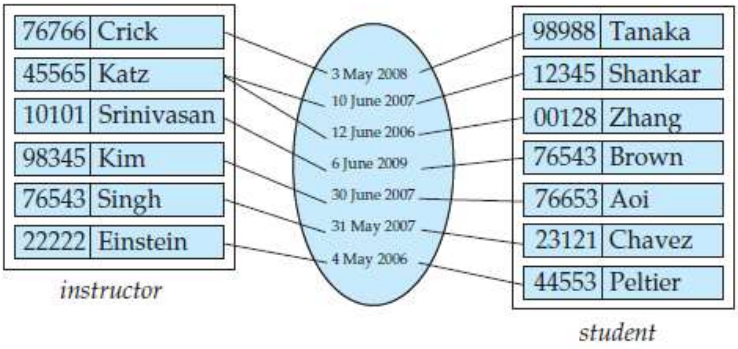


Figure 7.3 *date* as attribute of the *advisor* relationship set.

Mapping Cardinalities:

- Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
- Mapping cardinalities are most useful in describing binary relationship sets, although they can contribute to the description of relationship sets that involve more than two entity sets.
- In this section, we shall concentrate on only binary relationship sets.
- For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:
- The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*.
- If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**.

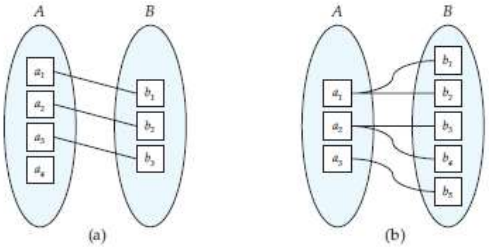


Figure 7.5 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

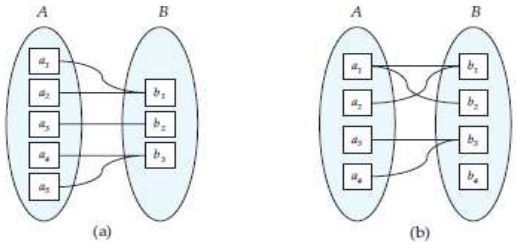


Figure 7.6 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

Keys:

- Keys play an important role in the relational database.
- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.
- For example, ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.

1. Primary key

- It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.
- In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary keys since they are also unique.
- For each entity, the primary key selection is based on requirements and developers.

2. Candidate key

- A candidate key is an attribute or set of attributes that can uniquely identify a tuple.
- Except for the primary key, the remaining attributes are considered a candidate key. The candidate keys are as strong as the primary key.
- For example: In the EMPLOYEE table, id is best suited for the primary key. The rest of the attributes, like SSN, Passport_Number, License_Number, etc., are considered a candidate key.

3. Super Key

- Super key is an attribute set that can uniquely identify a tuple. A super key is a superset of a candidate key.
- **For example:** In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME), the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.
- The super key would be EMPLOYEE-ID (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

4. Foreign key

- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.

5. Alternate key

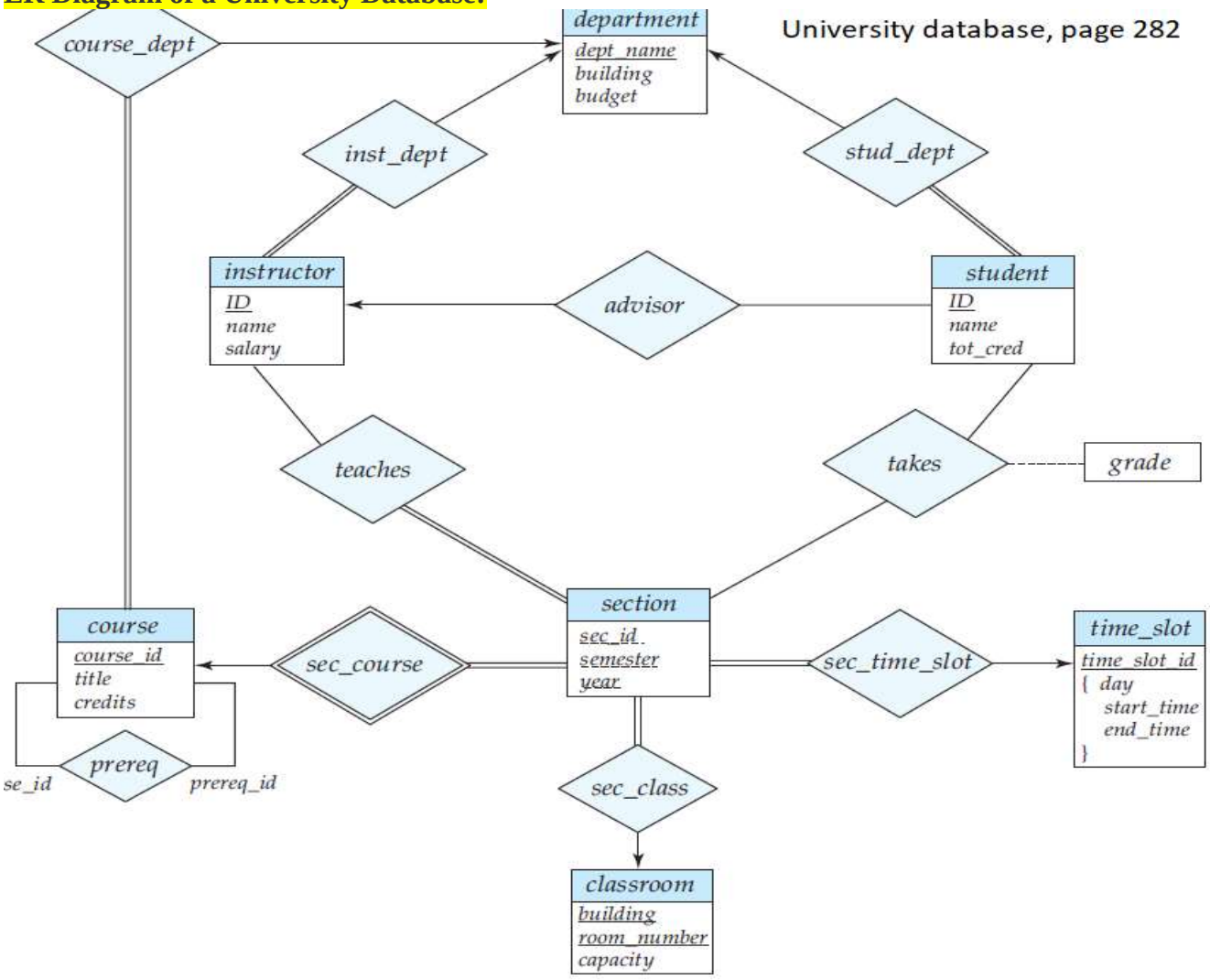
- There may be one or more attributes or a combination of attributes that uniquely identify each tuple in a relation. These attributes or combinations of the attributes are called the candidate keys. One key is chosen as the primary key from these candidate keys, and the remaining candidate key, if it exists, is termed the alternate key. In other words, the total number of the alternate keys is the total number of candidate keys minus the primary key. The alternate key may or may not exist. If there is only one candidate key in a relation, it does not have an alternate key.

- For example, employee relation has two attributes, Employee_Id and PAN_No, that act as candidate keys. In this relation, Employee_Id is chosen as the primary key, so the other candidate key, PAN_No, acts as the Alternate key.

6. Composite key

- Whenever a primary key consists of more than one attribute, it is known as a composite key. This key is also known as Concatenated Key.
- **For example,** in employee relations, we assume that an employee may be assigned multiple roles, and an employee may work on multiple projects simultaneously. So the primary key will be composed of all three attributes, namely Emp_ID, Emp_role, and Proj_ID in combination. So these attributes act as a composite key since the primary key comprises more than one attribute.

ER Diagram of a University Database:



Conclusion:

We have successfully studied and understand the basic theoretical concepts of ER diagrams.

Assignment No - A2 (a)

- **Title:** Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym
- **Objective:** Use DDL statements with all constraints, synonym, sequence to design the database and use Insert, Select, Update, Delete, operators, functions, set operators, to solve queries
- **Outcome:** Ability to develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym
- **Problem Statement:** Write a SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym

- **Hardware and Software requirement:**
  1. 64-bit Dell Machine
  2. Linux OS
  3. MySQL
  4.

Related Theory:

To begin with, 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 **TUTORIALS** database.

```
tutorials_tbl (

  tutorial_id INT NOT NULL AUTO_INCREMENT,

  tutorial_title VARCHAR(100) NOT NULL,

  tutorial_author VARCHAR(40) NOT NULL,

  submission_date DATE,
```

```
   PRIMARY KEY ( tutorial_id )

);
```

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.

- **Drop MySQL Tables-** It is very easy to drop an existing MySQL table, but you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.
  - Syntax

    ### Here is a generic SQL syntax to drop a MySQL table –

    DROP TABLE table_name ;

    Dropping Tables from the Command Prompt
  - To drop tables from the command prompt, we need to execute the DROP TABLE SQL command at the mysql> prompt.
  - Example
  - The following program is an example which deletes the **tutorials_tbl** –

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

Enter password:*******

mysql> use TUTORIALS;

Database changed

mysql> DROP TABLE tutorials_tbl

Query OK, 0 rows affected (0.8 sec)
```

```
mysql>
```

MySQL - INDEXES

- A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

- While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.
- Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.
- The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.
- The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.

Simple and Unique Index

You can create a unique index on a table. A unique index means that two rows cannot have the same index value. Here is the syntax to create an Index on a table.

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

You can use one or more columns to create an index.

For example, we can create an index on **tutorials_tbl** using **tutorial_author**.

```
CREATE UNIQUE INDEX AUTHOR_INDEX
ON tutorials_tbl (tutorial_author)
```

You can create a simple index on a table. Just omit the **UNIQUE** keyword from the query to create a simple index. A Simple index allows duplicate values in a table.

If you want to index the values in a column in a descending order, you can add the reserved word DESC after the column name.

```
mysql> CREATE UNIQUE INDEX AUTHOR_INDEX
ON tutorials_tbl (tutorial_author DESC)
```

ALTER command to add and drop INDEX

There are four types of statements for adding indexes to a table −

- **ALTER TABLE tbl_name ADD PRIMARY KEY (column_list)** − This statement adds a **PRIMARY KEY**, which means that the indexed values must be unique and cannot be NULL.
- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list)** − This statement creates an index for which the values must be unique (except for the NULL values, which may appear multiple times).
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list)**− This adds an ordinary index in which any value may appear more than once.
- **ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list)** − This creates a special FULLTEXT index that is used for text-searching purposes.

The following code block is an example to add index in an existing table.

```
mysql> ALTER TABLE testalter_tbl ADD INDEX (c);
```

You can drop any INDEX by using the **DROP** clause along with the ALTER command.

Try out the following example to drop the above-created index.

```
mysql> ALTER TABLE testalter_tbl DROP INDEX (c);
```

You can drop any INDEX by using the DROP clause along with the ALTER command.

ALTER Command to add and drop the PRIMARY KEY

You can add a primary key as well in the same way. But make sure the Primary Key works on columns, which are NOT NULL.

The following code block is an example to add the primary key in an existing table. This will make a column NOT NULL first and then add it as a primary key.

```
mysql> ALTER TABLE testalter_tbl MODIFY i INT NOT NULL;

mysql> ALTER TABLE testalter_tbl ADD PRIMARY KEY (i);
```

You can use the ALTER command to drop a primary key as follows –

```
mysql> ALTER TABLE testalter_tbl DROP PRIMARY KEY;
```

To drop an index that is not a PRIMARY KEY, you must specify the index name.

Displaying INDEX Information

You can use the **SHOW INDEX** command to list out all the indexes associated with a table. The vertical-format output (specified by \G) often is useful with this statement, to avoid a long line wraparound –

Try out the following example –

```
mysql> SHOW INDEX FROM table_name\G

........
```

- **Conclusion :**

We have successfully implemented Control Structures & Exception Handling in Fine Calculation on Book Issue.

Sample Code and Output

1. An employee management system needs to record following data about employees – ID, Name, Age, Department, Salary, Experience, AreaOfExperties. Identify columns, their data types and write create statement. Define primary key.
2. Create a view that will display all details of the employee except Salary and AreaOfExperties.

3. Create a sequence to generate employee id.
4. Create an index for the column ID.
5. Create a synonym for the generated table as “EMP” and demonstrate its use.

```
INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'AAA','10','COMP','10000','3','TRAINER');
 INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'BBB','11','CIVIL','20000','4','MANAGER')
;
 INSERT INTO EMPLOYEE  VALUES(SEQ.NEXTVAL,'CCC','11','IT','30000','5','TEAMLEAD');

----CREATE A VIEW THAT WILL DISPLAY ALL DETAILS OF EMPLOYEE EXCEPT SALARY AND
AREAOFEXPERTISE----
CREATE VIEW EMPLOYEE1 AS
SELECT EMPLOYEE_ID,NAME,AGE,DEPARTMENT,EXPERIENCE FROM EMPLOYEE;

----CREATE A SEQUENCE TO GENERATE EMPLOYEE ID----
CREATE SEQUENCE SEQ
START WITH 1
INCREMENT BY 1;

-----CRAETE AN INDEX FOR COLUMN_ID-----
CREATE INDEX INDEX_ID ON EMPLOYEE(EMPLOYEE_ID);

-----CREATE SYNONYM FOR GENERATED TABLE AS 'EMP' AND DEMONSTRATE ITS USE------
CREATE SYNONYM EMP FOR EMPLOYEE;
SELECT * FROM EMP;
```

Assignment No – A2 (b)

Problem statement: Design at least 10 SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, Delete with operators, functions, and set operator.

Objective: To implement Insert, Select, Update, Delete with operators, functions, and set operator

Outcome: Ability to implement SQL DML Queries.

Problem Statement: Write a SQL query involving Insert, Select, Update, Delete with operators, functions, and set operator.

Related theory

AGGREGATION:
MySQL aggregate functions retrieve a single value after performing a calculation on a set of values. In general, aggregate functions ignore null values.

COUNT:
The COUNT(column name) function returns the number of values (NULL values will not be counted) of the specified column:SQL COUNT(column name)

Syntax
SELECT COUNT(column name) FROM table name;

ORDER BY CLAUSE:
The ORDER BY keyword is used to sort the result-set by one or more columns. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword.

SQL ORDER BY Syntax:
SELECT * FROM table name
ORDER BY column name ASC or DESC;

DISTINCT KEYWORD:
In a table, a column may contain many duplicate values; and sometimes you only want to list the dfferent (distinct) values. The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax:
SELECT DISTINCT column name,column name FROM table name;

UNION:
The UNION operator is used to combine the result-set of two or more SELECT statements. Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

SQL UNION Syntax
SELECT column name(s) FROM table1
UNION
SELECT column name(s) FROM table2;

INTERSECTION:
The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the _rst SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
SQL INTERSECTION Syntax
SELECT column name(s) FROM table1
INTERSECTION
SELECT column name(s) FROM table2;

IN:
The IN operator allows you to specify multiple values in a WHERE clause.
SQL IN Syntax
SELECT column name(s)
FROM table name
WHERE column name IN (value1,value2,...);

NOTIN:
MySQL NOT IN() makes sure that the expression proceeded does not have any of the values present in the arguments.

SQL NOTIN Syntax:
SELECT column name(s)
FROM table name
WHERE column name NOT IN (value1,value2,...);

Functions performed in banking application are:

1. CREATE BRANCH:

To create a new branch in branch table we use the command:
INSERT into branch values('branch name','branch city',branch assets);
eg: If we want to create a branch named London, having branch city Lawrence and assets of 6000000, we use the command:
INSERT into branch values('London','Lawrence',6000000);

1. CREATE CUSTOMER:

To create a new customer in customer table we use the command:
INSERT into customer values('cust name',cust street','cust city');
eg: If we want to create a branch named London, having branch city Lawrence and assets of 6000000, we use the command:
INSERT into customer values('John',Main street','Glassgow');

2. SHOW:

To display the details of branch table we use the command:
SELECT * FROM branch;

To display the details of customer table we use the command
SELECT * FROM customer;

3. DEPOSIT:

To deposit amout in a particular account we use the command:
UPDATE account set balance = balance + amount WHERE acnt no = account no;
eg: If we want to deposit amount of 25000 in account no 25 we write
the command as:
UPDATE account set balance = balance + 25000
WHERE acnt no = 25;

4. DELETE:

To delete the records of certain customer from customer table we write the command as:
DELETE FROM customer
WHERE cust name='customer name'
eg: If we want to delete the records for the customer named John, we write the command as:
DELETE FROM customer WHERE cust name='John'

Conclusion

10 SQL queries for suitable database application using SQL DML statements: Insert, Select, Update, Delete with operators, functions, and set operator.

Sample Code and Output

For the following relation schema:
Account(Acc_no, branch_name,balance)
branch(branch_id, branch_name,branch_city,assets)
customer(cust_id, cust_name, cust_street,cust_city)
Depositor(cust_id, acc_no)
Loan(loan_no, branch_id, amount)
Borrower(cust_id, loan_no)
Create above tables and insert few rows in each table. Solve following query:

1. Find the branches where average account balance > 12000.
2. Find all customers who have an account or loan or both at bank.
3. Find all customers who have both account but not loan at bank.
4. Delete all tuples at every branch located in ‘Nigdi’.
5. Find Maximum loan amount in branch ‘Nigdi’
6. Find no. of depositors at each branch.
7. For all accounts in Akurdi branch increase the balance by 10%.

```
CREATE TABLE ACCOUNT
(
  ACC_NO INTEGER,
  BRANCH_NAME VARCHAR(30),
  BALANCE INTEGER
);
INSERT INTO ACCOUNT VALUES('10','AKURDI','1000');
INSERT INTO ACCOUNT VALUES('11','RAVET','2000');
INSERT INTO ACCOUNT VALUES('12','CHINCHWAD','3000');
```

```
CREATE TABLE BRANCH
(
   BRANCH_ID INTEGER,
  BRANCH_NAME VARCHAR(30),
  BRANCH_CITY VARCHAR(20),
  ASSETS VARCHAR(10)
);
INSERT INTO BRANCH VALUES('1','AKURDI','PUNE','HOUSE');
INSERT INTO BRANCH VALUES('2','RAVET','NASHIK','JEWELLERY');
INSERT INTO BRANCH VALUES('3','CHINCHWAD','AMRAVATI','FLAT');
INSERT INTO BRANCH VALUES('4','AKURDI','AMRAVAT','LAT');
INSERT INTO BRANCH VALUES('5','AKURDI','AMRAVA','AT');
INSERT INTO BRANCH VALUES('6','NIGDI','AMRAV','T');

CREATE TABLE CUSTOMER
(
   CUST_ID INTEGER,
  CUST_NAME VARCHAR(30),
  CUST_STREET VARCHAR(20),
  CUST_CITY VARCHAR(10)
);
INSERT INTO CUSTOMER VALUES('20','ABC','LINK ROAD','PUNE');
INSERT INTO CUSTOMER VALUES('21','BCD','LPRO ROAD','NASHIK');
INSERT INTO CUSTOMER VALUES('22','CDE','SHAGUN ROAD','AMRAVATI');

CREATE TABLE DEPOSITOR
(
   CUST_ID INTEGER,
   ACC_NO INTEGER
  );
INSERT INTO DEPOSITOR VALUES('20','10');
INSERT INTO DEPOSITOR VALUES('21','11');
INSERT INTO DEPOSITOR VALUES('22','12');

  CREATE TABLE LOAN
(
   LOAN_NO INTEGER,
   BRANCH_ID INTEGER,
   AMOUNT INTEGER
  );
INSERT INTO LOAN VALUES('100','31','10000');
INSERT INTO LOAN VALUES('101','32','20000');
INSERT INTO LOAN VALUES('102','33','30000');
INSERT INTO LOAN VALUES('103','6','90000');
   CREATE TABLE BORROWERR
(
   CUST_ID INTEGER,
   LOAN_NO INTEGER
  );

 INSERT INTO BORROWERR VALUES('41','1');
 INSERT INTO BORROWERR VALUES('42','2');
 INSERT INTO BORROWERR VALUES('43','3');

--LIST ALL CUSTOMERS IN ALPHABETICAL ORDER WHO HAVE LOAN IN AKURDI BRANCH--
SELECT CUST_NAME
```

```
FROM CUSTOMER,BRANCH
WHERE BRANCH_NAME='AKURDI'
ORDER BY CUST_NAME;

---FIND ALL CUSTOMERS WHO HAVE ACCOUNT OR LOAN OR BOTH AT BANK ---
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID OR BORROWERR.CUST_ID=CUSTOMER.CUST_ID;

---FIND ALL CUSTOMERS WHO HAVE BOTH ACCOUNT AND LOAN AT BAMK----
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID AND BORROWERR.CUST_ID=CUSTOMER.CUST_ID;

-----FIND ALL ACCOUNTS IN AKURDI BRANCH INCREASE THE BALANCE BY10%----
UPDATE ACCOUNT
SET BALANCE=BALANCE*1.1
WHERE BRANCH_NAME='AKURDI';

-----FIND AVERAGE ACCOUNT BALANCE AT AKURDI BRANCH-------
SELECT AVG(BALANCE) AS "AVERAGE BALANCE" FROM ACCOUNT
WHERE BRANCH_NAME='AKURDI';

-----FIND MINIMUM ACCOUNT BALANCE AT EACH BRANCH------
SELECT MIN(BALANCE) AS "MINIMUM BALANCE",BRANCH_NAME FROM ACCOUNT
GROUP BY BRANCH_NAME;

----DELETE ALL LOAN WITH LOAN AMOUNT BETWEEN 30000 AND 50000-----
DELETE FROM LOAN
WHERE AMOUNT<50000 AND AMOUNT>30000;


---A3(2)---

----FIND ALL BRANCHES WHERE AVERAGE BALANCE IS GREATER THAN 12000---
select BRANCH_NAME, avg (balance) from account
group by branch_name
having avg (balance) > 12000;

------FIND ALL CUSTOMERS WHO HAVE ACCOUNT BUT NOT LOAN ----
SELECT CUST_NAME FROM CUSTOMER,DEPOSITOR,BORROWERR
WHERE CUSTOMER.CUST_ID=DEPOSITOR.CUST_ID AND BORROWERR.CUST_ID!=CUSTOMER.CUST_ID;

----DELETE ALL TUPLES AT EVERY BRANCH LOCATED IN NIGDI-----
DELETE FROM ACCOUNT
WHERE BRANCH_NAME='NIGDI';

------FIND MAX LOAN AMOUNT IN NIGDI BRANCH-----
SELECT MAX(LOAN.AMOUNT) AS "MAXIMUM AMOUNT" FROM LOAN,BRANCH
WHERE LOAN.BRANCH_ID = BRANCH.BRANCH_ID AND BRANCH.BRANCH_NAME='NIGDI';

-------FIND NO. OF DEPOSITORS AT EACH BRANCH----
SELECT COUNT(DEPOSITOR.CUST_ID) AS "NO OF CUSTOMERS",ACCOUNT.BRANCH_NAME FROM
DEPOSITOR,ACCOUNT
WHERE DEPOSITOR.ACC_NO=ACCOUNT.ACC_NO
GROUP BY ACCOUNT.BRANCH_NAME;
```

Assignment No - A3

Problem statement: Design at least 10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View.

Objective: To implement : SQL DML statements: all types of Join, Sub-Query and View

Related theory

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.

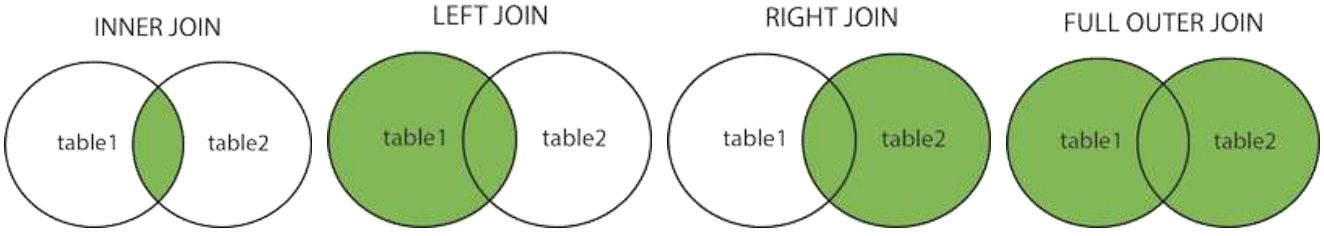
Subqueries can be used with the following SQL statements along with the comparision operators like =, <, >, >=, <= etc.

- SELECT
- INSERT
- UPDATE
- DELETE

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

1. (INNER) JOIN: Returns records that have matching values in both tables
2. LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table
3. RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table
4. FULL (OUTER) JOIN: Return all records when there is a match in either left or right table



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

10 SQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query and View are designed.

Sample Code and Output

For the following relation schema:
employee(employee-name, street, city)
works(employee-name, company-name, salary)
company(company-name, city)
manages(employee-name, manager-name)
Create above tables and insert 5 rows in each table. Give an expression in SQL for each of the following queries:

1. Find the names, street address, and cities of residence for all employees who work for 'First Bank Corporation' and earn more than $10,000.
2. Find the names of all employees in the database who live in the same cities as the companies for which they work.
3. Display employee details that live in cities Pune, Mumbai, and Nasik
4. List employees from ‘First Bank Corporation’ that earn salary more than all employees of ‘Small Bank Corporation’
5. Create a view that will display employee details along with name of his/her manager.
6. Find average salary of employees of ‘First Bank Corporation’.

Give employees of ‘First Bank Corporation’ 15% rise if salary is less than 20000.

```
create table employee(emp_name VARCHAR(100),street VARCHAR(100) ,city VARCHAR(100));
create table work(name VARCHAR(100),company VARCHAR(100),salary int);
create table company(cname VARCHAR(100),city VARCHAR(100));
create table manages(name VARCHAR(100),manager VARCHAR(100));

insert into employee values('Rohit','Pimpri','Pune');
insert into work values('Rohit','SKF',20000);
INSERT INTO COMPANY VALUES('SKF','Pune');
insert into manages values('Rohit','Tejas');
```

```
insert into employee values('Rahul','akurdi','Mumbai');
insert into work values('Rahul','tata',20500);
INSERT INTO COMPANY VALUES('tata','Mumbai');
insert into manages values('Rahul','Rohit');
insert into employee values('Pittu','AKURDI','Pune');
insert into work values('Pittu','RKF',5000);
INSERT INTO COMPANY VALUES('RKF','Pune');
insert into manages values('Pittu','Raj');

--A4(1)
--1
select * from employee where city='Pune';

--2
SELECT a.name,a.street,a.city FROM EMPLOYEE a,work b WHERE a.name=b.name and
b.company='SKF' AND b.salary>30000;

--3
select distinct a.name from employee a,company b where a.city=b.city;

--4
select name from work where company!='SKF';

--5
select name from manages where manager='Tejas';

--6
select avg(salary) from work where company='RKF';

--7
create view mysql as select employee.name,street,city,manager from employee FULL join manages on
employee.name = manages.name;
SELECT * FROM MYsql;

--A4(2)

--1
SELECT a.name,a.street,a.city FROM EMPLOYEE a,work b WHERE a.name=b.name and
b.company='SKF' AND b.salary>10000;

--2
select distinct a.name from employee a,company b where a.city=b.city;

--3
select * from employee where city='Pune' or city='Mumbai' or city='Nashik';

--4
select name from work where COMPANY='tata' and salary > (select max(salary) from work where
company='RKF');
```

```
--5
create view my as select employee.name,street,city,manager from employee FULL join manages on
employee.name = manages.name;
SELECT * FROM MY;

--6
select avg(salary) from work where company='RKF';

--7
update work SET salary=(1.15*salary) where company='SKF' and salary<20000
```

Assignment No. A4

Problem statement: 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:
1. Borrower(Rollin, Name, DateofIssue, NameofBook, Status)
2. 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.

Frame the problem statement for writing PL/SQL block inline with above statement.

Objective:
Design a PL/SQL Ananymous block with exception handling

Related theory

PL/SQL is a procedural language that Oracle developed as an extension to standard SQL to provide a way to execute procedural logic on the database. A. Control Structures: - PL/SQL allows the use of an IF statement to control the execution of a block
of code. In PL/SQL, the IF -THEN - ELSIF - ELSE - END IF construct in code blocks allow specifying certain conditions under which a specific block of code should be execute PL/SQL Control Structures are used to control flow of execution. PL/SQL provides different kinds of statements to provide such type of procedural capabilities. These statements are almost same as that of provided by other languages.
The flow of control statements can be classified into the following categories:
 Conditional Control
 Iterative Control
 Sequential Control
Conditional Control:
PL/SQL allows the use of an IF statement to control the execution of a block of code.
In PL/SQL, the IF -THEN - ELSIF - ELSE - END IF construct in code blocks allow specifying certain conditions under which a specific block of code

PL/SQL Is Block Structured

The Syntax for a PL/SQL Block

```
DECLARE
 variable_declarations
BEGIN
 program_code
EXCEPTION
 exception_handlers
END;
```

In this syntax, *variable_declarations* are any variables that you might want to define. Cursor definitions and nested PL/SQL procedures and functions are also defined here. *program_code* refers to the PL/SQL statements that make up the block. *exception_handlers* refers to program code that gets triggered in the event of a runtime error or exception.

PL/SQL blocks can be *nested*. One block can contain another block as in the following example:

```
DECLARE
 variable declarations go here
BEGIN
 some program code
 BEGIN
  code in a nested block
 EXCEPTION
  exception_handling_code
 END;
 more program code
END;
```

Conclusion

Designed the PL SQL Block.

Sample Code and Output

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:
1. Borrower(Rollin, Name, DateofIssue, NameofBook, Status)
2. Fine(Roll_no,Date,Amt)
    a. Accept roll_no & name of book from user.
    b. Check the number of days (from date of issue), if days are between 15 to 30 then fine amount will be Rs 5per day.
    c. If no. of days>30, per day fine will be Rs 50 per day & for days less than 30, Rs. 5 per day.
    d. After submitting the book, status will change from I to R.
    e. If condition of fine is true, then details will be stored into fine table.

```
SET SERVEROUT ON
SET VERIFY OFF
/*
CREATE TABLE  borrower(roll_no NUMBER , name VARCHAR2(25), dateofissue DATE,name_of_book
VARCHAR2(25), status VARCHAR2(20));
CREATE TABLE fine(roll_no NUMBER,date_of_return DATE,amt NUMBER);
INSERT INTO borrower VALUES(45,'ASHUTOSH',TO_DATE('01-08-2022','DD-MM-YYYY'),'HARRY
POTTER','PENDING');
INSERT INTO borrower VALUES(46,'ARYAN',TO_DATE('15-08-2022','DD-MM-YYYY'),'DARK MATTER','PENDING');
INSERT INTO borrower VALUES(47,'ROHAN',TO_DATE('24-08-2022','DD-MM-YYYY'),'SILENT HILL','PENDING');
INSERT INTO borrower VALUES(48,'SANKET',TO_DATE('26-08-2022','DD-MM-YYYY'),'GOD OF WAR','PENDING');
INSERT INTO borrower VALUES(49,'SARTHAK',TO_DATE('09-09-2022','DD-MM-YYYY'),'SPIDER-MAN','PENDING');
*/
```

```
DECLARE
        i_roll_no NUMBER;
        name_of_book VARCHAR2(25);
        no_of_days NUMBER;
        return_date DATE := TO_DATE(SYSDATE,'DD-MM-YYYY');
        temp NUMBER;
        doi DATE;
        fine NUMBER;
BEGIN
        i_roll_no := &i_roll_no;
        name_of_book := '&nameofbook';
        --dbms_output.put_line(return_date);
        SELECT to_date(borrower.dateofissue,'DD-MM-YYYY') INTO doi FROM borrower WHERE
borrower.roll_no = i_roll_no AND borrower.name_of_book = name_of_book;
        no_of_days := return_date-doi;
        dbms_output.put_line(no_of_days);
        IF (no_of_days >15 AND no_of_days <=30) THEN
                fine := 5*no_of_days;

        ELSIF (no_of_days>30 ) THEN
                temp := no_of_days-30;
                fine := 150 + temp*50;
        END IF;
        dbms_output.put_line(fine);
        INSERT INTO fine VALUES(i_roll_no,return_date,fine);
        UPDATE borrower SET status = 'RETURNED' WHERE borrower.roll_no = i_roll_no;


END;
/
```

Assignment No. A5

Problem statement: 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 <=1500 and marks>=990 then student will be placed in distinction category if marks scored are between 989 and900 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. Stud_Marks(name, total_marks) Result(Roll,Name, Class)
Frame the separate problem statement for writing PL/SQL Stored Procedure and function, inline with above statement. The problem statement should clearly state the requirements.

Objective:

To Create a PL SQL stored procedure and function.

Related Theory

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

PL/SQL Create Procedure

Syntax for creating procedure:

```
1. CREATE [OR REPLACE] PROCEDURE procedure_name
2.     [ (parameter [,parameter]) ]
3. IS
```

```
4.    [declaration_section]
5.  BEGIN
6.    executable_section
7.  [EXCEPTION
8.    exception_section]
9.  END [procedure_name];
```

PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

```
1. CREATE [OR REPLACE] FUNCTION function_name [parameters]
2. [(parameter_name [IN | OUT | IN OUT] type [, ...])]
3. RETURN return_datatype
4. {IS | AS}
5. BEGIN
6.    < function_body >
7. END [function_name];
```

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Conclusion

Created PL/SQL function and procedure.

Sample Code and Output

Write a Stored Procedure namely proc_Grade for the categorization of student. If marks scored by students in examination is <=1500 and marks>=990 then student will be placed in distinction category if marks scored are between 989 and900 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.

Stud_Marks(name, total_marks) and Result(Roll,Name, Class)

```
create or replace procedure proc_grade
                    (temp in number,
                      p_roll_no out stud_marks.roll_no%type,
                      p_name out stud_marks.name%type,
                      p_total out stud_marks.total_marks%type)
 as
 begin
 select name,total_marks,roll_no into p_name,p_total,p_roll_no from stud_marks where roll_no=temp;

 if p_total <=1500 and p_total >= 990 then
         insert into result values(p_roll_no,p_name,'distinction');
 else if p_total <=989 and p_total >= 900 then
         insert into result values(p_roll_no,p_name,'first class');
 else if p_total <=899 and p_total >= 825 then
         insert into result values(p_roll_no,p_name,'HSC');
else
         insert into result values(p_roll_no,p_name,'fail');
 end if;
 end if;
 end if;
exception
when no_data_found then
dbms_output.put_line('Roll no ' || temp ||'  not found');
 end;
 /
```

Execution

procedure created.

SQL> select * from result;

no rows selected

Calling Procedure

```
Declare

temp number(20);
 p_roll_no stud_marks.roll_no%type;
 p_name stud_marks.name%type;
 p_total stud_marks.total_marks%type;
 Begin
temp:=&temp;
 Proc_grade(temp,p_roll_no,p_name,p_total);
 End;
 /
```

Enter value for temp: 1

old 7: temp:=&temp;
new 7: temp:=1;
PL/SQL procedure successfully completed.

SQL> select * from result;

ROLL_NO	NAME	CLASS
1	ABC	distinction

SQL> select * from stud_marks;

ROLL_NO	NAME	TOTAL_MARKS
1	ABC	1000
2	XYZ	960
3	PQR	850
4	LMN	820

----------Same Code but using Function(Just for Reference)-------

Write a function namely func_Grade for the categorization of student. If marks scored by students in examination is <=1500 and marks>=990 then student will be placed in distinction category if marks scored are between 989 and900 category is first class, if marks 899 and 825 category is Higher Second Class.

Write a PL/SQL block for using function created with above requirement.
Stud_Marks(name, total_marks)
Result(Roll,Name, Class)

```
create or replace function fun_grade(temp in number)
                        return number
as
        p_roll_no stud_marks.roll_no%type;
        p_name stud_marks.name%type;
        p_total stud_marks.total_marks%type;
begin
select name,total_marks,roll_no into p_name,p_total,p_roll_no from stud_marks where roll_no=temp;

if p_total <=1500 and p_total >= 990 then
        insert into result values(p_roll_no,p_name,'distinction');
else if p_total <=989 and p_total >= 900 then
        insert into result values(p_roll_no,p_name,'first class');
else if p_total <=899 and p_total >= 825 then
        insert into result values(p_roll_no,p_name,'HSC');
else
        insert into result values(p_roll_no,p_name,'fail');
end if;
end if;
end if;
```

```
 return p_roll_no;
 exception
          when no_data_found then
                    dbms_output.put_line('Roll no ' || temp ||'  not found');
          end;
```

Function created.

SQL> select * from stud_marks;

```
  ROLL_NO NAME               TOTAL_MARKS
---------- -------------------- -----------
         1 ABC                  1000
         2 XYZ                  960
         3 PQR                   850
         4 LMN                   820
```

SQL> select * from result;

```
 ROLL_NO NAME               CLASS
-------- -------------------- --------------------
       1 ABC                 distinction
       2 XYZ                first class
       3 PQR                 HSC
```

Calling Function:

```
Declare
          temp number(20):=&temp;
          p_roll_no varchar2(20);
Begin
          p_roll_no :=fun_grade(temp);
End;
/

Enter value for temp: 4
old  2: temp number(20):=&temp;
new  2: temp number(20):=4;
```

PL/SQL procedure successfully completed.

```
SQL> select * from result;
  ROLL_NO NAME               CLASS
---------- -------------------- --------------------
         1 ABC                 distinction
         2 XYZ
         4 LMN                fail first class

         3 PQR                 HSC
```

Assignment No. A6

Problem statement: 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_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.
Frame the separate problem statement for writing PL/SQL block to implement all types of Cursors inline with above statement. The problem statement should clearly state the requirements.

Objective:

Learn and Implement the cursors

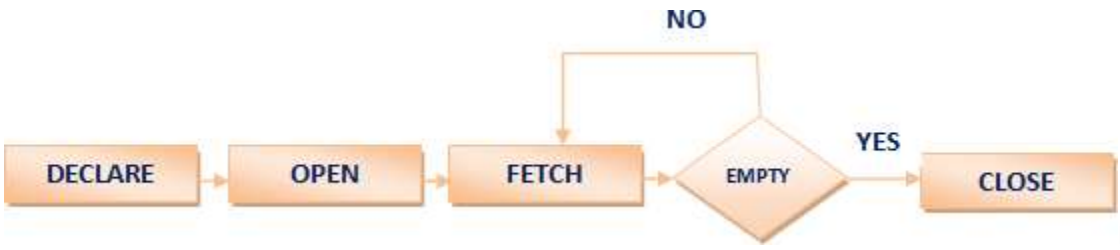
Related theory

hen 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:



PL/SQL Cursor

Let's examine each step in greater detail.

Declaring PL/SQL Cursor

To use PL/SQL cursor, first you must declare it in the declaration section of PL/SQL block or in a package as follows:

```
CURSOR cursor_name [ ( [ parameter_1 [, parameter_2 ...] ) ]
    [ RETURN return_specification ]
IS sql_select_statements
```

[FOR UPDATE [OF [column_list]];

```
1 CURSOR cursor_name [ ( [ parameter_1 [, parameter_2 ...] ) ]
2       [ RETURN return_specification ]
3 IS sql_select_statements
4 [FOR UPDATE [OF [column_list]];
```

First, you declare the name of the cursor cursor_name after the CURSOR keyword. The name of the cursor can have up to 30 characters in length and follows the naming rules of identifiers in PL/SQL. It is important to note that cursor's name is not a variable so you cannot use it as a variable such as assigning it to other cursor or using it in an expression. The parameter1, parameter2… are optional elements in the cursor declaration. These parameters allow you to pass arguments into the cursor. The RETURN return_specification is also an optional part.

Second, you specify a valid SQL statement that returns a result set where the cursor points to.

Third, you can indicate a list of columns that you want to update after the FOR UPDATE OF. This part is optional so you can omit it in the CURSOR declaration.

Here is an example of declaring a cursor:

```
CURSOR cur_chief IS
    SELECT first_name,
        last_name,
        department_name
    FROM employees e
    INNER JOIN departments d ON d.manager_id = e.employee_id;
```

We retrieved data from employees and departments tables using the SELECTwith the INNER JOIN clause and set the cur_chief cursor to this result set.

Opening a PL/SQL Cursor

After declaring a cursor, you can open it by using the

```
1 OPEN cursor_name [ ( argument_1 [, argument_2 ...] ) ];
```

You have to specify the cursor's name cursor_name after the keyword OPEN. If the cursor was defined with a parameter list, you need to pass corresponding arguments to the cursor.

When you OPEN the cursor, PL/SQL executes the SQL SELECT statement and identifies the active result set. Notice that the OPEN action does not actually retrieve records from the database. It happens in the FETCH step. If the cursor was declared with the FOR UPDATE clause, PL/SQL locks all the records in the result set.

We can open the `cur_chief` cursor as follows:

```
1 OPEN cur_chief;
```

Fetching Records from PL/SQL Cursor

Once the cursor is open, you can fetch data from the cursor into a record that has the same structure as the cursor. Instead of fetching data into a record, you can also fetch data from the cursor to a list of variables.

The fetch action retrieves data and fills the record or the variable list. You can manipulate this data in memory. You can fetch the data until there is no record found in active result set.

The syntax of `FETCH` is as follows:

```
1 FETCH cursor_name INTO record or variables
```

You can test the cursor's attribute %FOUND or %NOTFOUND to check if the fetch against the cursor is succeeded. The cursor has more attributes that we will cover in the next section.

We can use PL/SQL LOOP statement together with the FETCH to loop through all records in active result set as follows:

```
LOOP
   -- fetch information from cursor into record
   FETCH cur_chief INTO r_chief;


   EXIT WHEN cur_chief%NOTFOUND;


   -- print department - chief
```

```
  DBMS_OUTPUT.PUT_LINE(r_chief.department_name || ' - ' ||
                r_chief.first_name || ',' ||
                r_chief.last_name);
 END LOOP;
```

Closing PL/SQL Cursor

You should always close the cursor when it is no longer used. Otherwise, you will have a memory leak in your program, which is not expected.

To close a cursor, you use `CLOSE` statement as follows:

```
1 CLOSE cursor_name;
```

And here is an example of closing the `cur_chief` cursor:

```
1 CLOSE cur_chief;
```

PL/SQL Cursor Attributes

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

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

Conclusion

PL/SQL Cursors are studies.

Sample Code and Output

----------Program for Assignment 6(1)- Explicit Cursor----------

Write PL/SQL block using explicit cursor for following requirements:
College has decided to mark all those students detained (D) who are having attendance less than 75%. Whenever such update takes place, a record for the same is maintained in the d_stud table.
create table stud21(roll number(4), att number(4), status varchar(1));
create table d_stud(roll number(4), att number(4));

```
Declare

Cursor crsr_att is select roll, att,status from stud21 where att<75;

mroll stud21.roll%type;

matt stud21.att%type;

mstatus stud21.status%type;

Begin

open crsr_att;

if crsr_att%isopen then

loop

fetch crsr_att into mroll,matt,mstatus;

exit when crsr_att%notfound;

if crsr_att%found then

update stud21 set status='D' where roll=mroll;

insert into d_stud values(mroll,matt);

end if;

end loop;

end if;

end;
```

----------Program for Assignment 6(2)-Parameterized Cursor------

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

Create table new_class(roll number(10), name varchar(10));
Create table old_class(roll number(10), name varchar(10));

```
Declare

cursor crsr_class is select * from old_class;

cursor crsr_chk(str_name varchar) is select roll from new_class where name =
str_name; str_roll new_class.roll%type;

str_name new_class.name%type;

v varchar(10);

Begin

Open crsr_class;

Loop

fetch crsr_class into str_roll,str_name;

Exit When crsr_class%NOTFOUND;

Open crsr_chk(str_name);

Fetch crsr_chk into v;

if crsr_chk%FOUND Then

dbms_output.put_line('brach'|| str_name || 'exist');

Else

dbms_output.put_line('brach'|| str_name || ' not exist. Inserting in
New_class table');

insert into new_class values(str_roll,str_name); End if;

Close crsr_chk;

End loop;
```

```
Close crsr_class;

End;
```

------Program for Assignment 6(3)-explicit cursor FOR LOOP--------

An explicit cursor FOR LOOP statement prints the name and job ID of every employee with designation CLERK and whose manager has an ID greater than 120.

```
Create table employees (job_id varchar(10), name varchar(10), manager_id number(3));
insert into employees values('CLERK','ABC',111);

insert into employees values('PEON','XYZ',110);

insert into employees values('CLERK','PQR',121);

insert into employees values('CLERK','ZXC',121);

select * from employees;

SET SERVEROUT ON;

DECLARE

  CURSOR c1 IS

    SELECT name, job_id FROM employees

    WHERE job_id LIKE '%CLERK%' AND manager_id > 120

    ORDER BY name;

BEGIN

  FOR item IN c1

  LOOP

  DBMS_OUTPUT.PUT_LINE('Name = ' || item.name || ', Job = ' || item.job_id);

  END LOOP;
```

```
END;
```

Assignment No. A7

Problem statement: 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.
Frame the problem statement for writing Database Triggers of all types, in-line with above statement. The problem statement should clearly state the requirements.

Objective:

To learn and implement the PL SQL Triggers

Related theory

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −
• A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
• A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
• A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers
Triggers can be written for the following purposes −
• Generating some derived column values automatically
• Enforcing referential integrity
• Event logging and storing information on table access
• Auditing
• Synchronous replication of tables
• Imposing security authorizations
• Preventing invalid transactions

Creating Triggers
The syntax for creating a trigger is −

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

```
Executable-statements
EXCEPTION
Exception-handling-statements
END;
```

Where,

• CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.

• {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.

• {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.

• [OF col_name] − This specifies the column name that will be updated.

• [ON table_name] − This specifies the name of the table associated with the trigger.

• [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

• [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

• WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters

Select * from customers;

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

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

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

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

The following points need to be considered here −

• OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

• If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

• The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table –

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null.

Let us now perform one more DML operation on the CUSTOMERS table.
The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

Conclusion

Data base triggers has been studied

Sample Code and Output

Implementation:

1. create table borrower (rollin int, name varchar(30), dateofissue date, nameofbook varchar(30), status char(10));

2. insert borrower values (101, 'abc', '2017-07-16', 'dbms', 'r'), (102, 'abc1', '2017-07-16', 'cn', 'i'), (103,'abc3', '2017-07-18', 'toc', 'i'), (104, 'abc4', '2017-07-20', 'ds', 'i'), (105, 'abc5', '2017-07-23', 'daa', 'r'), (106, 'nisha', '2017-08-10', 'splm', 'r');

3. create table library_audit(rollin int, name char(10), dateofissue date, nameofbook char(10), status char, ts timestamp);

4. Create trigger for insert:

```
create trigger after_insert after insert on borrower for each row
begin
insert into library_audit values(new.rollin, new.name, new.dateofissue, new.nameofbook, new.status, current_timestamp);
end;
/
```

```
select * from borrower
+--------+-------+-------------+------------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+-------------+------------+--------+
| 101 | abc | 2017-07-16 | dbms | r |
| 102 | abc1 | 2017-07-16 | cn | i |
| 103 | abc3 | 2017-07-18 | toc | i |
| 104 | abc4 | 2017-07-20 | ds | i |
| 105 | abc5 | 2017-07-23 | daa | r |
| 106 | nisha | 2017-08-10 | splm | r |
```

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

```
select * from library_audit
insert into borrower values(107,'ada','2017-08-10','dbms','i')
select * from borrower
```

```
+--------+-------+-------------+------------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+-------------+------------+--------+
| 101 | abc | 2017-07-16 | dbms | r |
| 102 | abc1 | 2017-07-16 | cn | i |
| 103 | abc3 | 2017-07-18 | toc | i |
| 104 | abc4 | 2017-07-20 | ds | i |
| 105 | abc5 | 2017-07-23 | daa | r |
| 106 | nisha | 2017-08-10 | splm | r |
| 107 | ada | 2017-08-10 | dbms | i |
+--------+-------+-------------+------------+--------+
```

```
select * from library_audit
+--------+------+-------------+------------+--------+--------------------+
| rollin | name | dateofissue | nameofbook | status | ts |
+--------+------+-------------+------------+--------+--------------------+
| 107 | ada | 2017-08-10 | dbms | i | 2017-09-19 05:24:57 |
+--------+------+-------------+------------+--------+--------------------+
```

5. After DELETE Trigger

```
create trigger after_delete after delete on borrower for each row
begin
insert into library_audit values(old.rollin, old.name, old.dateofissue, old.nameofbook,
old.status, current_timestamp());
end ;
/
```

```
select * from borrower //
+--------+-------+-------------+------------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+-------------+------------+--------+
| 101 | abc | 2017-07-16 | dbms | r |
| 102 | abc1 | 2017-07-16 | cn | i |
| 103 | abc3 | 2017-07-18 | toc | i |
| 104 | abc4 | 2017-07-20 | ds | i |
| 105 | abc5 | 2017-07-23 | daa | r |
| 106 | nisha | 2017-08-10 | splm | r |
| 107 | ada | 2017-08-10 | dbms | i |
+--------+-------+-------------+------------+--------+
```

```
select * from library_audit //
+--------+------+------------+-----------+--------+--------------------+
| rollin | name | dateofissue | nameofbook | status | ts |
+--------+------+------------+-----------+--------+--------------------+

| 107 | ada | 2017-08-10 | dbms | i | 2017-09-19 05:24:57 |
+--------+------+------------+-----------+--------+--------------------+

delete from borrower where rollin = 105
select * from borrower

+--------+-------+------------+-----------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+------------+-----------+--------+
| 101 | abc | 2017-07-16 | dbms | r |
| 102 | abc1 | 2017-07-16 | cn | i |
| 103 | abc3 | 2017-07-18 | toc | i |
| 104 | abc4 | 2017-07-20 | ds | i |
| 106 | nisha | 2017-08-10 | splm | r |
| 107 | ada | 2017-08-10 | dbms | i |
+--------+-------+------------+-----------+--------+

select * from library_audit

+--------+------+------------+-----------+--------+--------------------+
| rollin | name | dateofissue | nameofbook | status | ts |
+--------+------+------------+-----------+--------+--------------------+
| 107 | ada | 2017-08-10 | dbms | i | 2017-09-19 05:24:57 |
| 105 | abc5 | 2017-07-23 | daa | r | 2017-09-19 05:26:51 |
+--------+------+------------+-----------+--------+--------------------+
```

6. After UPDATE Trigger

```
create trigger after_update after update on borrower for each row
begin
insert into library_audit values(old.rollin, old.name, old.dateofissue, old.nameofbook,
old.status, current_timestamp());
end ;
/

select * from borrower //
+--------+-------+------------+-----------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+------------+-----------+--------+
| 101 | abc | 2017-07-16 | dbms | r |
| 102 | abc1 | 2017-07-16 | cn | i |
```

103	abc3	2017-07-18	toc	i
104	abc4	2017-07-20	ds	i
106	nisha	2017-08-10	splm	r
107	ada	2017-08-10	dbms	i
+--------+-------+-------------+------------+--------+

select * from library_audit
+--------+------+-------------+------------+--------+---------------------+
| rollin | name | dateofissue | nameofbook | status | ts |
+--------+------+-------------+------------+--------+---------------------+
| 107 | ada | 2017-08-10 | dbms | i | 2017-09-19 05:24:57 |
| 105 | abc5 | 2017-07-23 | daa | r | 2017-09-19 05:26:51 |
+--------+------+-------------+------------+--------+---------------------+

update borrower set status='r' where borrower.rollin=104
select * from borrower
+--------+-------+-------------+------------+--------+
| rollin | name | dateofissue | nameofbook | status |
+--------+-------+-------------+------------+--------+
101	abc	2017-07-16	dbms	r
102	abc1	2017-07-16	cn	i
103	abc3	2017-07-18	toc	i
104	abc4	2017-07-20	ds	r
106	nisha	2017-08-10	splm	r
107	ada	2017-08-10	dbms	i
+--------+-------+-------------+------------+--------+
select * from library_audit
+--------+------+-------------+------------+--------+---------------------+
| rollin | name | dateofissue | nameofbook | status | ts |
+--------+------+-------------+------------+--------+---------------------+
107	ada	2017-08-10	dbms	i	2017-09-19 05:24:57
105	abc5	2017-07-23	daa	r	2017-09-19 05:26:51
104	abc4	2017-07-20	ds	i	2017-09-19 05:28:10
+--------+------+-------------+------------+--------+---------------------+

Assignment No. A8
MYSQL/Oracle Database Connectivity

Title:
JDBC Connectivity

Objective:
Is to implement database navigation operations using JDBC/ODBC.

Problem Statement:
Implement MYSQL/ORACLE database connectivity with PHP/PYTHON/JAVA implement database navigation operations using JDBC/ODBC.

Software or Hardware Requirement: Eclipse, Oracle database/MYSQL Database.

Theory:

- Java is very standardized, but there are many versions of SQL.
- JDBC is a means of accessing SQL databases from Java JDBC is a standardized API for use by Java programs.
- JDBC is also a specification for how third-party vendors should write database drivers to access specific SQL versions.
- JDBC: establishes a connection with a database sends SQL statements processes the results.
- The JDBC API contains methods to communicate with DBMS or RDBMS The JDBC API uses the JDBC driver to carry out it tasks Java Program JDBC API JDBC Driver Database.

JDBC Steps
1. Instantiate proper driver
2. Open connection to database
3. Connect to database
4. Query database (or insert/update/delete)
5. Process the result
6. Close connection to database.

1. Instantiate Driver
Class.forName("driver class")
Driver class is vendor dependent, e.g., sun.jdbc.odbc.JdbcOdbcDriver JDBC-ODBC bridge used to access ODBC Sources oracle.jdbc.driver.OracleDriver driver to access Oracle Database com.mysql.jdbc.Driver driver to access MySQL database .

2. Open Connection

DriverManager.getConnection(url)

Or

DriverManager.getConnection(url, user, pwd)

URL is jdbc:: e.g., jdbc:oracle:someDB

jdbc:://:/ jdbc:mysql://localhost:3306/testDB

3. Connect to database

Load JDBC driver Class.forName("com.mysql.jdbc.Driver").newInstance();

or

Class.forName("oracle.jdbc.driver.OracleDriver")

Make connection Connection conn = DriverManager.getConnection(url);

URL Format: jdbc:://:/ jdbc:mysql://localhost:3306/testDB

4. Query database

Create statement Statement stmt = conn.createStatement();

stmt object sends SQL commands to database

Methods

executeQuery() for SELECT statements

executeUpdate() for INSERT, UPDATE, DELETE, statements Send SQL statements stmt.executeQuery("SELECT …"); stmt.executeUpdate("INSERT …")

Conclusion:

Implemented database navigation operations using JDBC/ODBC.

Input:

```
package session;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Connect {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		 try {

		Class.forName("oracle.jdbc.driver.OracleDriver");//Instantiation of the driver
```

```
Connection con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "xe", "SYSTEM123"); //creation of statement for execution

Statement st = con.createStatement();       //opening the connection with db

//ResultSet rs = st.executeQuery("create table conndemo(name varchar(25), roll int ,marks int)"); // create table

ResultSet rs1 = st.executeQuery("insert into conndemo values('lmn',2,21)"); //inserted to table
ResultSet rs2 = st.executeQuery("insert into conndemo values('abc',3,25)");
ResultSet rs3 = st.executeQuery("insert into conndemo values('xyz',4,22)");
ResultSet rs4 = st.executeQuery("insert into conndemo values('def',5,27)");

ResultSet rs5 = st.executeQuery("select * from conndemo"); //fetching data
System.out.println("executed successfully");

while (rs5.next()) {
   System.out.println(rs5.getString("name") + " " + rs5.getInt("roll") + " " + rs5.getInt(3));
}

int rs6 = st.executeUpdate("update studjdbcdemo set name='tina' where roll=3");

ResultSet rs7 = st.executeQuery("drop table studjdbcdemo"); //Deleting the database table view

ResultSet rs8 = st.executeQuery("create view viewJDBCNamee as select name from studjdbcdemo"); //creating view
ResultSet rs9 = st.executeQuery("select * from viewJDBCNamee");
while (rs9.next()) {
   System.out.println("" + rs9.getString(1));
}

ResultSet rs10 = st.executeQuery("alter table studjdbcdemo add admissionDate Date"); //alter table structure

ResultSet rs11 = st.executeQuery("alter table studjdbcdemo drop column admis-
sionDate");
```

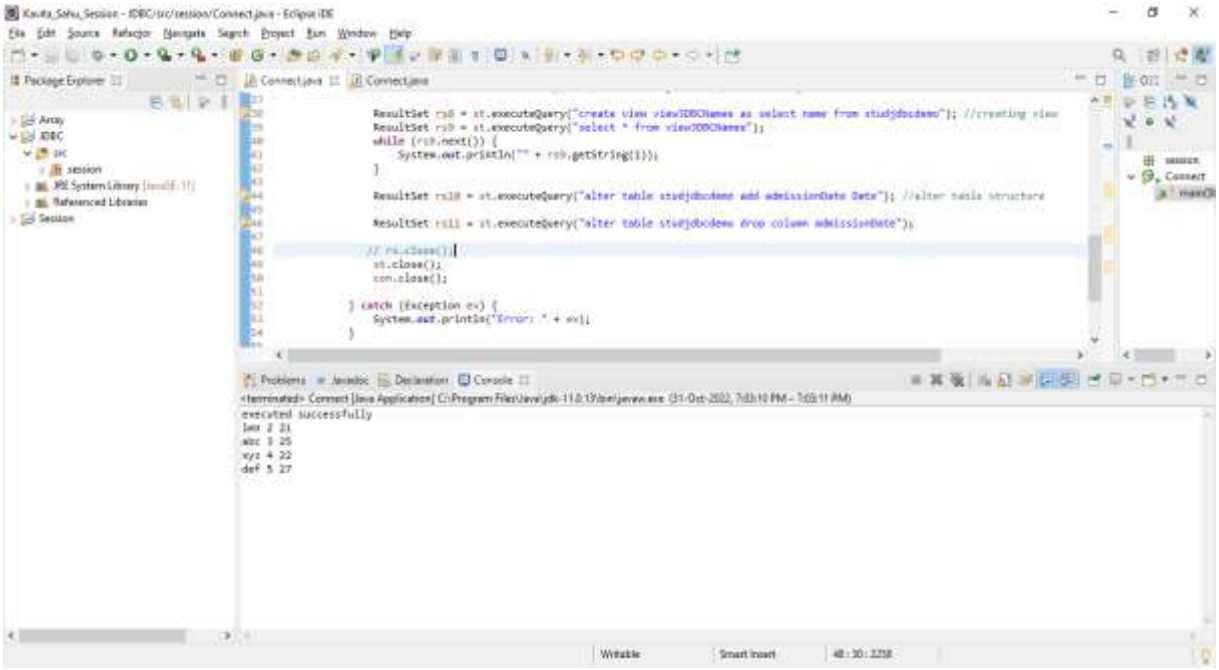
```
            // rs.close();
             st.close();
             con.close();

          } catch (Exception ex) {
             System.out.println("Error: " + ex);
          }

      }

}
```

Output:



Assignment No: 9 – B1

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

- **Objectives**: To implement SAVE and logical operators of MongoDB.

- **Problem Statement** :To implement SAVE method and logical AND AND logical OR operators using MongoDB.

- **Hardware and Software Requirement**:

  1. Computer System with Linux/ Open Source Operating System
  2. MongoDB

- **Theory**:

MongoDB's update() and save() methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method :

The update() method updates the values in the existing document.

Syntax
The basic syntax of update() method is as follows −
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)

MongoDB Save() Method :

The save() method replaces the existing document with the new document passed in the save() method.

Syntax
The basic syntax of MongoDB save() method is shown below −
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

Logical Operators in MongoDB:
AND in MongoDB

In the find() method, if you pass multiple keys by separating them by ',' then MongoDB treats it as AND condition.
Following is the basic syntax of AND −

```
>db.mycol.find(
{
$and: [
{key1: value1}, {key2:value2}
]
}
).pretty()
```

OR in MongoDB

To query documents based on the OR condition, you need to use $or keyword
Following is the basic syntax of OR −

```
>db.mycol.find(
{
$or: [
{key1: value1}, {key2:value2}
]
}
).pretty()
```

- **Coding:**

1.Example
Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
>db.mycol.find()


{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
```

{$set:{'title':'New MongoDB Tutorial'}},{multi:true})

2.Example :

```
>db.mycol.save( {
"_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point New Topic",
"by":"Tutorials Point" } )
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"Tutorials Point New Topic",
"by":"Tutorials Point"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Logical Operators:

Logical AND

```
>db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty() {
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": "tutorials point",
"url": "http://www.tutorialspoint.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}
```

For the above given example, equivalent where clause will be ' where by = 'tutorials point' AND title = 'MongoDB Overview' '.
You can pass any number of key, value pairs in find clause.

Logical OR

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": "tutorials point",
"url": "http://www.tutorialspoint.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
}
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is

'tutorials point'. Equivalent SQL where clause is 'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
{"title": "MongoDB Overview"}]}).pretty()
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": "tutorials point",
"url": "http://www.tutorialspoint.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": "100"
```

- **Test Cases :**

Test_id	Description	Expected O/P	Actual O/P	Result
1	To SAVE a new document	Existing document should be replaced by new one	Existing document is replaced by new one	Pass
2	AND	db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).	Any one of the two field is inserted	Pass
3	OR	db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).	Both the fields are inserted	pass

- **Conclusion**:

We have implemented UPDATE method, SAVE method and logical operators using MongoDB.

Sample Code

Create a collection **student** in mongodb and insert few documents with fields (stud_id, stud_name, dept, marks)

1. Find students having marks greater than 50
2. Find students having marks between 50 and 80

3. Find students having marks more than 60 from ‘Computer’ department
4. Update marks of all students from ‘Civil’ department. Set marks to 30
5. Delete students from ‘Chemical’ department having marks less than 30

```
db.stud.insert({"stud_id":"2","name":"saroj","dept":"comp","marks":999})
db.stud.insert([{"stud_id":"4","name":"Tej","dept":"comp","marks":100},{"stud_id":"
44","name":"hero","dept":"mech","marks":1000},{"stud_id":"55","name":"rider","dept"
:"comp","marks":1010}])

1. marks greater than 900:-
db.stud.find({"marks":{$gt:900}})

2.marks between 900 to 1000
db.stud.find({"marks":{$gte:900,$lte:1000}})

3.marks greater than 900 and dept=comp:-
db.stud.find({$and:[{"marks":{$gt:900}},{"dept":"comp"}]})

4. update marks=100 where branch=mech
db.stud.update({"dept":"mech"},{$set:{"marks":"100"}})

5.delete stud from mech with marks less than 950
db.stud.remove({$and:[{"dept":"mech"},{"marks":{$lt:950}}]})
```

Create a collection **employee** in mongodb and insert few documents with fields (emp_id, emp_name, dept, salary)

1. Find employees having salary greater than 50000
2. Find employees having salary between 50000 and 80000
3. Find employees having salary more than 60000 from ‘HR’ department
4. Update marks of all students from ‘Civil’ department. Set marks to 30

Delete students from ‘Chemical’ department having marks less than 30

```
db.emp1.insert([{"emp_id":"1","name":"Shubh","dept":"comp","salary":10005},{"emp_id
":"2","name":"Shubhash","dept":"hr","salary":10000},
{"emp_id":"3","name":"baji","dept":"stack","salary":10120},
{"emp_id":"4","name":"ram","dept":"comp","salary":10120}])
1.salary grater then 5000
db.emp1.find({"salary":{$gt:5000}})

2.10000 to 10050
db.emp1.find({"salary":{$gte:10000,$lte:10050}})

3.salary>6000 & dept=hr
db.emp1.find({$and:[{"salary":{$gt:6000}},{"dept":"hr"}]})
```

Assignment No. 10 – B2

- **Title**: Implement aggregation and indexing with suitable example using MongoDB.
- **Objectives**: To implement Aggregation and Indexing using MongoDB.
- **Problem Statement** : To perform MongoDB Database programming to aggregate and index a document.
- **Hardware and Software Requirement**:

  1. Computer System with Linux/ Open Source Operating System
  2. MongoDB

- **Theory**:

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 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 the index.

The ensureIndex() Method
To create an index you need to use ensureIndex() method of MongoDB.

Syntax
The basic syntax of ensureIndex() method is as follows().
>db.COLLECTION_NAME.ensureIndex({KEY:1})

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

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.

Basic syntax of aggregate() method is as follows −
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)

Coding:

Indexing
Example
>db.mycol.ensureIndex({"title":1})
>
In ensureIndex() method you can pass multiple fields, to create index on multiple fields.
>db.mycol.ensureIndex({"title":1,"description":-1})

Aggregation:
Example
In the collection you have the following data −
{
_id: ObjectId(7df78ad8902c)
title: 'MongoDB Overview',
tags: description: 'MongoDB is no sql database',
by_user: 'tutorials point',
url: 'http://www.tutorialspoint.com ['mongodb', 'database', 'NoSQL'],
likes: 100
},
{
_id: ObjectId(7df78ad8902d)
title: 'NoSQL Overview',
description: 'No sql database is very fast',
by_user: 'tutorials point',
url: 'http://www.tutorialspoint.com',
tags: ['mongodb', 'database', 'NoSQL'],
likes: 10
},
{
_id: ObjectId(7df78ad8902e)
title: 'Neo4j Overview',
description: 'Neo4j is no sql database',
by_user: 'Neo4j',
url: 'http://www.neo4j.com',
tags: ['neo4j', 'database', 'NoSQL'],
likes: 750
},

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following aggregate() method −

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])
{
"result" : [
{
"_id" : "tutorials point",
"num_tutorial" : 2
},
{
"_id" : "Neo4j",
"num_tutorial" : 1
}
],
```

Output:

"ok" : 1

1.$sum :Sums up the defined value from all documents in the collection.
db.mycol.aggregate([{$group : {_id : "$by_user",num_tutorial : {$sum : "$likes"}}}])

2.$avg : Calculates the average of all given values from all documents in the collection.
db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])

3. $min : Gets the minimum of the corresponding values from all documents in the collection.
db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])

4. $max :Gets the maximum of the corresponding values from all documents in the collection.
db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])

- **Test Cases :**

Test_id	Description	Expected O/P	Actual O/P	Result
1	Indexing	If (+1) then indexing is done in ascending order. If (-1) then indexing is done in descending order.	Indexing is done	Pass
2	Aggregation	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum	{ "_id" : "tutorials	Pass

		: 1}}}])	point",
			"num_tutorial" : 2
			},
			{
			"_id" : "Neo4j",
			"num_tutorial" : 1
			}
			],	

- **Conclusion**:

We have successfully implemented indexing and aggregation method using MongoDB.

Assignment No. 11 – B3

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

- **Objectives**: To implement map reduce using MongoDB.

- **Problem Statement** :To perform mapReduce operation on records to reduce database size using MongoDB.

- **Hardware and Software Requirement**:

  1. Computer System with Linux/ Open Source Operating System
  2. MongoDB

- **Theory**:
  As per the MongoDB documentation, 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.

  MapReduce Command

  Following is the 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.

- **Coding:**:

  Using MapReduce
  Consider the following document structure storing user posts. The document stores user_name of the user and the status of post.

  ```
  {
  "post_text": "tutorialspoint is an awesome website for tutorials",
  "user_name": "mark",
  "status":"active"
  }
  ```

  Now, we will use a mapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code −

  ```
  >db.posts.mapReduce(
  function() { emit(this.user_id,1); },
  function(key, values) {return Array.sum(values)}, {
  query:{status:"active"},
  out:"post_total"
  }
  )
  ```

  The above mapReduce query outputs the following result −

  ```
  {
  "result" : "post_total",
  "timeMillis" : 9,
  "counts" : {
  "input" : 4,
  "emit" : 4,
  "reduce" : 2,
  "output" : 2},

  "ok" : 1,
  }
  ```

  Output:

To see the result of this mapReduce query, use the find operator −

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)}, {
query:{status:"active"},
out:"post_total"
}
).find()
```

The above query gives the following result which indicates that both users tom and mark have two posts in active states −

```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

- **Test Cases** :

Test_id	Description	Expected O/P	Actual O/P	Result
1	MapReduce Operation	MapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user	{ "_id" : "tom", "value" : 2 } { "_id" : "mark", "value" : 2 }	Pass

- **Conclusion**:

We have successfully implemented Mapreduce method using MongoDB.

Sample Code

Create a collection **bank** in mongodb and insert few documents with fields (cust_id, cust_name, branch, balance)

Write a MapReduce function to display balance in each branch of the bank

cust_id, cust_name, branch, balance

```
db.bank.insert([{"cust_id":1,"cust_name":"Tejas","branch":"Akurdi","bal":30000},
{"cust_id":2,"cust_name":"Tejal","branch":"Akurdi","bal":35000},
{"cust_id":3,"cust_name":"Amey","branch":"Pimpri","bal":3000},
```

{"cust_id":4,"cust_name":"Ajay","branch":"Pimpri","bal":10000},
{"cust_id":5,"cust_name":"Ashish","branch":"Pune","bal":50000}])

Mapreduce:-
var Mapfunction = function(){emit(this.branch,this.bal)}
var Reducefunction = function(key,values){return Array.sum(values)}
db.bank.mapReduce(Mapfunction,Reducefunction,{'out':'resultant'})

Answer:-
db.resultant.find()

Create a collection books in mongodb and insert few documents with fields (book_id, title, author, type) Write a MapReduce function to display number of books of each type

book_id, title, author, type
db.books.insert([{book_id:1,title:"My",author:"Rajesh",type:"songs"}])
db.books.insert([{book_id:2,title:"Jack",author:"Raj",type:"Poem"},
{book_id:3,title:"What",author:"John",type:"Story"},
{book_id:4,title:"Real",author:"Warner",type:"Real Stories"}])
db.books.insert([{book_id:5,title:"Ram",author:"Raj",type:"Poem"},
{book_id:6,title:"Temperature",author:"Tejas",type:"Story"}])

Mapreduce:-
var Mapfunction = function(){emit(this.type,1)}
var Reducefunction = function(key,values){return Array.sum(values)}
db.books.mapReduce(Mapfunction,Reducefunction,{'out':'typeofbooks'})

Answer:-
db.typeofbooks.find()

Assignment No. 12 - B4

- **Title**: Write a program to implement MogoDB database connectivity with PHP/ python/Java Implement Database navigation operations (add, delete, edit etc.) using ODBC/JDBC.

- **Objectives**: To establish a connectivity between MongoDB and Java.

- **Problem Statement**: Implement MongoDB database connectivity with Java .
  Implement Database navigation operation (add , delete ,edit) using JDBC.

- **Theory**:
  Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

Solution:

A. Related theory

Before we start using MongoDB in our Java programs, we need to make sure that we have MongoDB JDBC Driver and Java set up on the machine. You can check Java tutorial for Java installation on your machine. Now, let us check how to set up MongoDB JDBC driver.

- You need to download the jar from the path Download mongo.jar. Make sure to download latest release of it.
- You need to include the mongo.jar into your classpath.

Connect to database

To connect database, you need to specify database name, if database doesn't exist then mongodb creates it automatically.

Code snippets to connect to database would be as follows:

```
import com.mongodb.MongoClient;
import com.mongodb.MongoException;
import com.mongodb.WriteConcern;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;
import com.mongodb.ServerAddress;
import java.util.Arrays;
public class MongoDBJDBC{
public static void main( String args[] ){
try{
// To connect to mongodb server
MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
// Now connect to your databases
DB db = mongoClient.getDB( "test" );
System.out.println("Connect to database successfully");
boolean auth = db.authenticate(myUserName, myPassword);
```

```
System.out.println("Authentication: "+auth);
}catch(Exception e){
System.err.println( e.getClass().getName() + ": " + e.getMessage() );
}
}
}
```

Now, let's compile and run above program to create our database test. You can change your path as per your requirement. We are assuming current version of JDBC driver mongo-2.10.1.jar is available in the current path

```
$javac MongoDBJDBC.java
$java -classpath ".:mongo-2.10.1.jar" MongoDBJDBC
Connect to database successfully
Authentication: true
```

If you are going to use Windows machine, then you can compile and run your code as follows:

```
$javac MongoDBJDBC.java
$java -classpath ".;mongo-2.10.1.jar" MongoDBJDBC
Connect to database successfully
Authentication: true
```

Value of **auth** will be true, if the user name and password are valid for the selected database.

Create a collection

To create a collection, **createCollection()** method of **com.mongodb.DB** class is used.

Code snippets to create a collection:

Now connect to your databases

```
DB db = mongoClient.getDB( "test" );
System.out.println("Connect to database successfully");
DBCollection coll = db.createCollection("mycol");
System.out.println("Collection created successfully");
```

Getting/ selecting a collection

To get/select a collection from the database, **getCollection()** method of **com.mongodb.DBCollection** class is used.

```
DBCollection coll = db.getCollection("mycol");
System.out.println("Collection mycol selected successfully");
```

Insert a document

To insert a document into mongodb, **insert()** method of **com.mongodb.DBCollection** class is used.

```
BasicDBObject doc = new BasicDBObject("title", "MongoDB").
append("description", "database").
append("likes", 100).
append("url", "http://www.tutorialspoint.com/mongodb/").
append("by", "tutorials point");
coll.insert(doc);
System.out.println("Document inserted successfully");
```

Retrieve all documents

To select all documents from the collection, **find()** method of **com.mongodb.DBCollection** class is used. This method returns a cursor, so you need to iterate this cursor.

```
DBCursor cursor = coll.find();
int i=1;
while (cursor.hasNext()) {
System.out.println("Inserted Document: "+i);
System.out.println(cursor.next());
i++;
}
```

Update document

To update document from the collection, **update()** method of **com.mongodb.DBCollection** class is used.

```
// find hosting = hostB, and update the clients to 110
BasicDBObject newDocument = new BasicDBObject();
newDocument.put("clients", 110);
BasicDBObject searchQuery = new
BasicDBObject().append("hosting", "hostB");
collection.update(searchQuery, newDocument,false,true);
```

To find perticular document

To find document from the collection, you need to select the documents using **Find or findOne()** method and then pass document to be serached as argument to it.

```
DBObject myDoc = coll.findOne(searchQuery);
```

Delete first document

To delete first document from the collection, you need to first select the documents using **findOne()** method and then **remove** method of **com.mongodb.DBCollection** class.

```
DBObject myDoc = coll.findOne();
col1.remove(myDoc);
```

B. Conclusion

By using Mongo.Jar file we can connect java and mongoDb and can perform all the operation using java connectivity

Sample Code

```
import com.mongodb.*;
public class Mongo {
public static void main( String args[] ) {
      try{
      MongoClient mongoClient = new MongoClient( "localhost" , 27017 );
      DB db = mongoClient.getDB( "test" );
      System.out.println("Connect to database successfully");
      DBCollection col1=db.createCollection("shree1",new BasicDBObject());
      System.out.println("collection created");
      DBCollection col2=db.createCollection("shruti2",new BasicDBObject());
         System.out.println("collection created");
         DBCollection col1=db.getCollection("shree1");
         System.out.println("collection created");
```

```
          BasicDBObject doc1 = new BasicDBObject();
          doc1.put("name", "shraddha");
          doc1.put("website", "google.com");

          BasicDBObject doc2 = new BasicDBObject();
          doc2.put("addressLine1", "Sweet Home");
          doc2.put("addressLine2", "Karol Bagh");
          doc2.put("addressLine3", "New Delhi, India");

          col1.insert(new BasicDBObject[] {doc1,doc2});

          }catch(Exception e){
          System.err.println( e.getClass().getName() + ": " + e.getMessage()
);
       }
    }
}

/*output
> show dbs
PL01   0.078GB
abc    0.078GB
cat    0.078GB
doc    0.078GB
local  0.078GB
maggi  0.078GB
mdb    0.078GB
suj    0.078GB
test   0.078GB
xyz    0.078GB
> use doc
switched to db doc
> show collections
shree1
shruti2
> db.shree1.find()
{ "_id" : ObjectId("57ac2983522ed0bb4f416f15"), "name" : "shraddha",
"website" : "google.com" }
{ "_id" : ObjectId("57ac2983522ed0bb4f416f16"), "addressLine1" : "Sweet
Home", "addressLine2" : "Karol Bagh", "addressLine3" : "New Delhi, India" }
*/
```