EXPERIMENT NO. 1 (Group A)

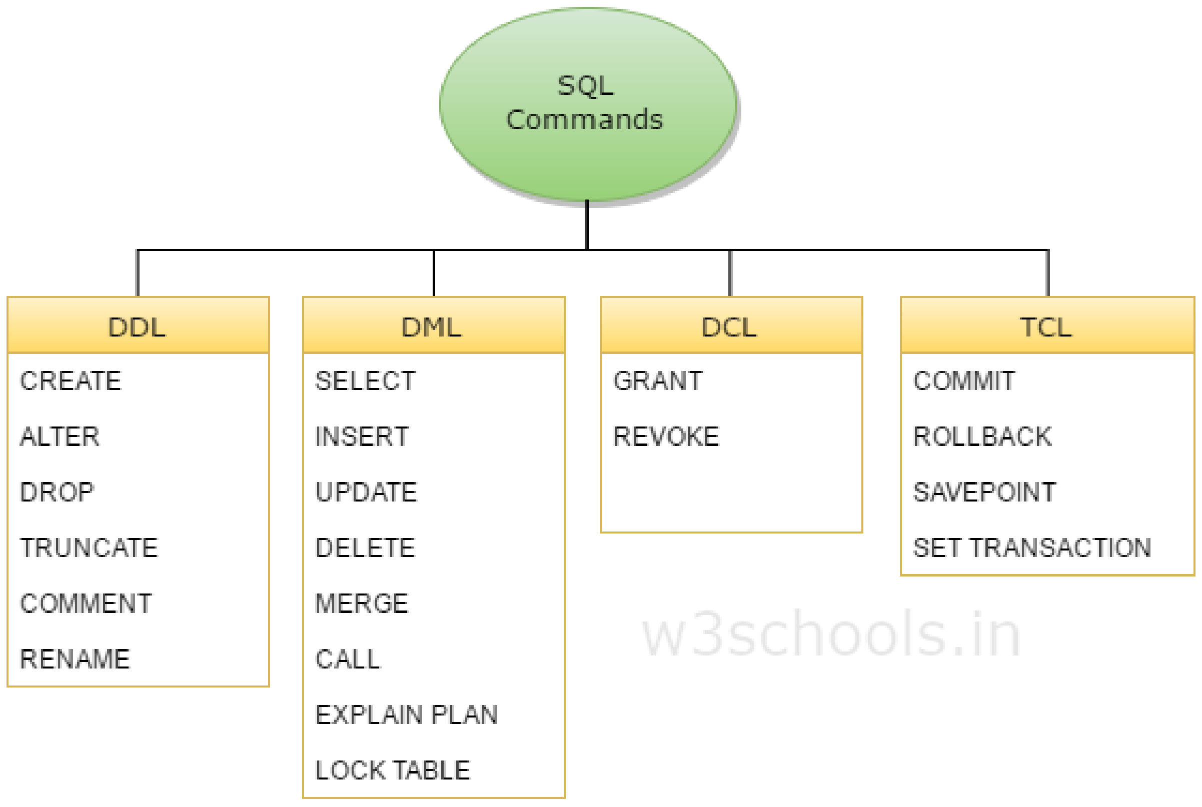
Aim: 1. Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Index, Sequence, Synonym, different constraints etc.

2. Write at least 10 SQL queries on the suitable database application using SQL DML statements.

- **Outcome:** 1) Students will be able to learn and understand various DDL queries like create, drop, truncate.

  2) Students will be able to demonstrate creating and dropping SQL objects like table, view, sequence, index etc.
- **Hardware Requirement:** Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.
- **Software Requirement: MySQL**
- **Theory:**

DATA DEFINITION LANGUAGE (DDL) QUERIES



DDL- Data Definition Language (DDL) statements are used to define the database structure or schema. Data Definition Language understanding with database schemas and describes how the data should consist in the database, therefore language statements like CREATE TABLE or
ALTER TABLE belongs to the DDL. DDL is about "metadata".

DDL includes commands such as CREATE, ALTER and DROP statements. DDL is used to CREATE, ALTER OR DROP the database objects (Table, Views, Users).

Data Definition Language (DDL) are used different statements:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

Create Database: From the MySQL command line, enter the command

CREATE DATABASE <DATABASENAME>;

Replace <DATABASENAMEs> with the name of your database. It cannot include spaces. For example, to create a database of all the US states, you might enter

CREATE DATABASE pune;

- Note: Commands do not have to be entered in upper-case.
  - Note: All MySQL commands must end with ";". If you forgot to include the semicolon, you can enter just ";" on the next line to process the previous ommand.

Select your database: Once the database has been created, you will need to select it in order to begin editing it. Enter the command

- USE pune;

You will see the message Database changed, letting you know that your active database is now pune.

Display a list of your available databases: Enter the command

- SHOW DATABASES;

to list all of the databases you have stored. Besides the database you just created, you will also see a mysql database and a test database.

Create table: We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

- **create table** *department* (*dept name* **varchar** (20), *building* **varchar** (15), *budget* **numeric (12,2), primary key (*dept name*));**

The relation created above has three attributes, *dept name*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, 2 of which are after the decimal point. The **create table** command also specifies that the *dept name* attribute is the primary key of the *department* relation.

Insert values in table: A newly created relation is empty initially. We can use the **insert** command to load data into the relation. For example, if we wish to insert the fact that there is an instructor named Smith in the Biology department with *instructor id* 10211 and a salary of

$66,000, we write:

- **insert into** *instructor* **values** (10211, ’Smith’, ’Biology’, 66000);

Drop table: To remove a relation from an SQL database, we use the **drop table** command. The **drop table** command deletes all information about the dropped relation from the database. The command

- **drop table r**

Alter Table: We use the **alter table** command to add attributes to an existing relation. All tuplesin the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

- **alter table** *r* **add** *AD*;

where *r* is the name of an existing relation, *A* is the name of the attribute to be added, and *D* is the type of the added attribute. We can drop attributes from a relation by the command

- **alter table** *r* **drop** *A*;

where *r* is the name of an existing relation, and *A* is the name of an attribute of the relation.

View: SQL allows a “virtual relation” to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used. Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

Create View: We define a view in SQL by using the create view command. To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is:

- create view *v* as <query expression>;

where <query expression> is any legal query expression. The view name is represented by *v*.

Consider again the clerk who needs to access all data in the *instructor* relation, except *salary*. The clerk should not be authorized to access the *instructor* relation. Instead, a view relation *faculty* can be made available to the clerk, with the view defined as follows:

- create view *faculty* as select *ID*, *name*, *dept name* from *instructor*;

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *physics fall 2009*, we can find all Physics courses offered in the Fall 2009 semester in the Watson building by writing:

- select *course id* from *physics fall 2009* where *building*= 'Watson';

Alter View: The CREATE VIEW statement creates a new view, or replaces an existing view if the OR REPLACE clause is given. If the view does not exist, CREATE OR REPLACE VIEW is the same as CREATE VIEW.

Drop view: Use the DROP VIEW statement to remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it.

Drop view viewname;

Create Index:

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 table student using column rno

- CREATE UNIQUE INDEX RNO_INDEX ON student (rno);

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.

- CREATE UNIQUE INDEX RNO_INDEX ON student (rno DESC);

Alter 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.

- ALTER TABLE student ADD INDEX (id);

Drop Index:

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.

- ALTER TABLE student DROP INDEX (id);

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

Sequence: To create a sequence value in SQL query, we can use AUTO_INCREMENT with optional starting value.

1. create table sonali(id int primary key auto_increment, name varchar(20));
2. create table sonali(id int primary key auto_increment = 10, name varchar(20));

Summary of commands :

Creating View, replacing or updating view and dropping view

1. Create view view1 as select * from emp;
2. Create or replace view1 as select * from emp1;

3. Drop view view1;

Crating sequence and inserting values

3. create table sonali(id int primary key auto_increment, name varchar(20));
4. create table sonali(id int primary key auto_increment = 10, name varchar(20));
5. insert into sonali(name) values('a'),('b'),('c');
6. insert into sonali values(NULL,'sonali');
7. insert into sonali(name) values('sonali');

Creating index

8. create index id1 on sonali(id)
9. create index id1 on sonali(id desc)

Conclusion: In this assignment, we have studied and demonstrated various DDL statements in SQL.

EXPERIMENT NO. 2 (Group A)

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

- **Outcome:** At end of this experiment, student will be able to study of all types of Join, Sub-Query and View.

- **Hardware Requirement:** Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.

- **Software Requirement:** MySQL

- **Theory:**

Introduction to Joins: An **SQL JOIN** clause is used to combine rows from two or more tables, based on a common field between them. Before we continue with examples, we will list the types of JOIN you can use, and the differences between them.

- **JOIN**: Return rows when there is at least one match in both tables
- **LEFT JOIN**: Return all rows from the left table, even if there are no matches in the right table
- **RIGHT JOIN**: Return all rows from the right table, even if there are no matches in the left table
- **FULL JOIN**: Return rows when there is a match in one of the tables

SQL JOIN Syntax

```
SELECT column_name(s)
FROM table_name1,table_name2
WHERE table_name1.column_name=table_name2.column_name
```

SQLJOIN EXAMPLE

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the persons with any orders.

We use the following SELECT statement:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM
Persons, Orders
WHERE Persons.P_Id=Orders.P_Id
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678

SQLLEFT JOIN KEYWORD

The LEFT JOIN keyword returns all rows from the left table (table_name1), even if there are no matches in the right table (table_name2).

SQL LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table_name1   LEFT
JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

SQLLEFT JOIN EXAMPLE

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM
Persons
LEFT JOIN Orders
ON Persons.P_Id=Orders.P_Id
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	

The LEFT JOIN keyword returns all the rows from the left table (Persons), even if there are no matches in the right table (Orders).

SQLRIGHT JOINKEYWORD

The RIGHT JOIN keyword Return all rows from the right table (table_name2), even if there are no matches in the left table (table_name1).

SQL RIGHT JOIN Syntax

```
SELECT column_name(s) FROM
table_name1
```

```
RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

SQL RIGHT JOIN EXAMPLE: Now we want to list all the orders with containing persons - if any, from the tables above.

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM
Persons
RIGHT JOIN Orders
ON Persons.P_Id=Orders.P_Id
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
		34764

The RIGHT JOIN keyword returns all the rows from the right table (Orders), even if there are no matches in the left table (Persons).

SQLFULL JOIN KEYWORD

The FULL JOIN keyword return rows when there is a match in one of the tables.

SQL FULL JOIN Syntax

```
SELECT column_name(s)
FROM table_name1   FULL
JOIN table_name2
ON table  name1.column  name=table  name2.column  name
```

SQLFULL JOIN EXAMPLE

Now we want to list all the persons and their orders, and all the orders with their persons.

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM
Persons
FULL JOIN Orders
ON Persons.P_Id=Orders.P_Id
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	
		34764

The FULL JOIN keyword returns all the rows from the left table (Persons), and all the rows from the right table (Orders). If there are rows in "Persons" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Persons", those rows will be listed as well.

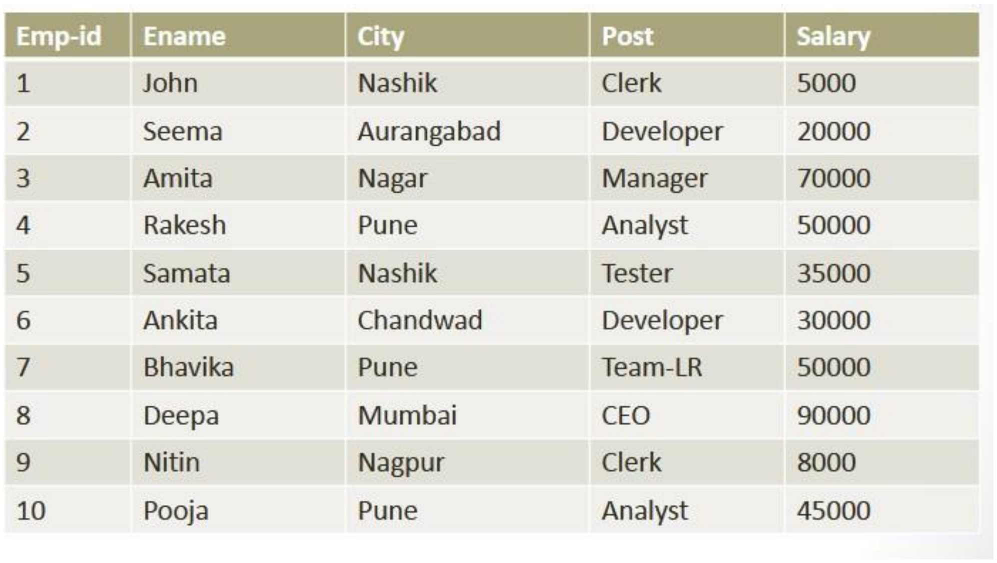
Types of Subqueries

Single Row Sub Query: Sub query which returns single row output. They mark the usage of single row comparison operators, when used in WHERE conditions.

Multiple row sub query: Sub query returning multiple row output. They make use of multiple row comparison operators like IN, ANY, ALL. There can be sub queries returning multiple columns also.

Correlated Sub Query: Correlated subqueries depend on data provided by the outer query.Thistype of subquery also includes subqueries that use the EXISTS operator to test the existence of data rows satisfying specified criteria.

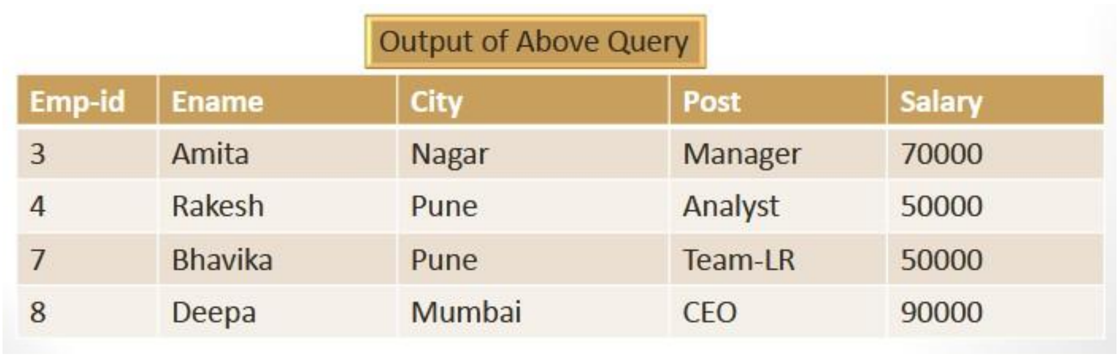
Example:



Emp-id	Ename	City	Post	Salary
1	John	Nashik	Clerk	5000
2	Seema	Aurangabad	Developer	20000
3	Amita	Nagar	Manager	70000
4	Rakesh	Pune	Analyst	50000
5	Samata	Nashik	Tester	35000
6	Ankita	Chandwad	Developer	30000
7	Bhavika	Pune	Team-LR	50000
8	Deepa	Mumbai	CEO	90000
9	Nitin	Nagpur	Clerk	8000
10	Pooja	Pune	Analyst	45000

Display the information of employees, paid more than ‘pooja' from emptable

Select *from emp where salary > (select salary from emp where name= ‘Pooja’);



Output of Above Query

Emp-id	Ename	City	Post	Salary
3	Amita	Nagar	Manager	70000
4	Rakesh	Pune	Analyst	50000
7	Bhavika	Pune	Team-LR	50000
8	Deepa	Mumbai	CEO	90000

MySQL Subqueries -Multiple rows with ALL, ANY, IN operator

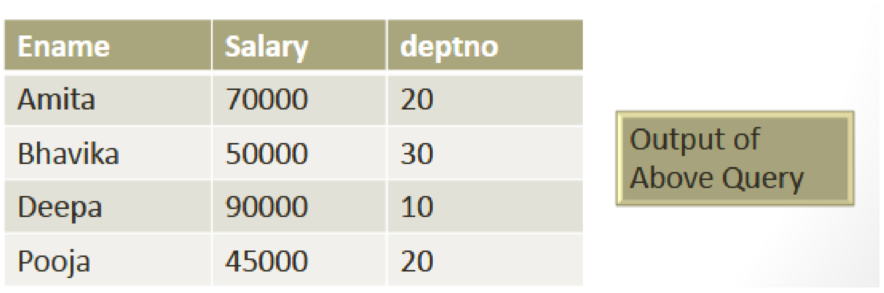
- [> ALL] More than the highest value returned by the subquery
- [< ALL] Less than the lowest value returned by the subquery
- [< ANY] Less than the highest value returned by the subquery
- [> ANY] More than the lowest value returned by the subquery

- [= ANY] Equal to any value returned by the subquery (same as IN)

>All Example-

Display the employee name, salary and department no of those employees whose salary is higher than all developers' salary.

SELECT Ename, salary, deptno FROM EMPWHERE salary > All (SELECT salary FROM emp Where post='Developer')



Ename	Salary	deptno
Amita	70000	20
Bhavika	50000	30
Deepa	90000	10
Pooja	45000	20

Output of Above Query

Conclusion: Hence, in this way we have executed all types of joins, subquery and views using DML commands.

EXPERIMENT NO. 3 (Group A)

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

- **Outcome:** Students will be able to learn and understand various MongoDB queries.
- **Hardware Requirement:** Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.
- **Software Requirement: MySQL**

Theory:

What is NoSQL?

NoSQL is a non-relational DBMS, that does not require a fixed schema, avoids joins, and is easy to scale. The purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook, Google collect terabytes of user data every single day.

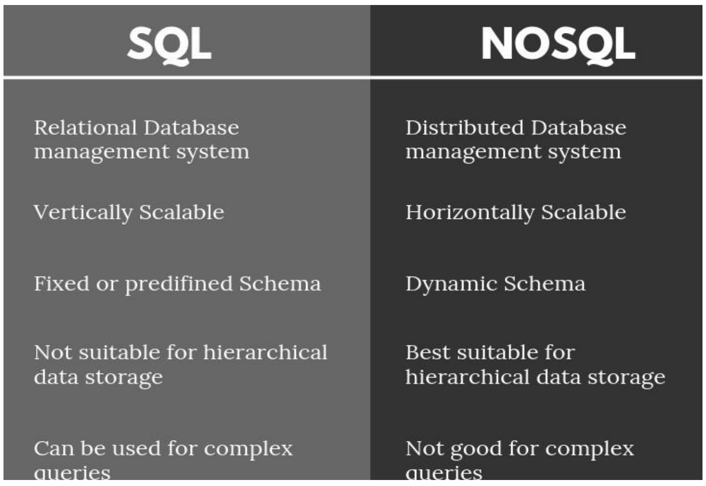
NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Why NoSQL?

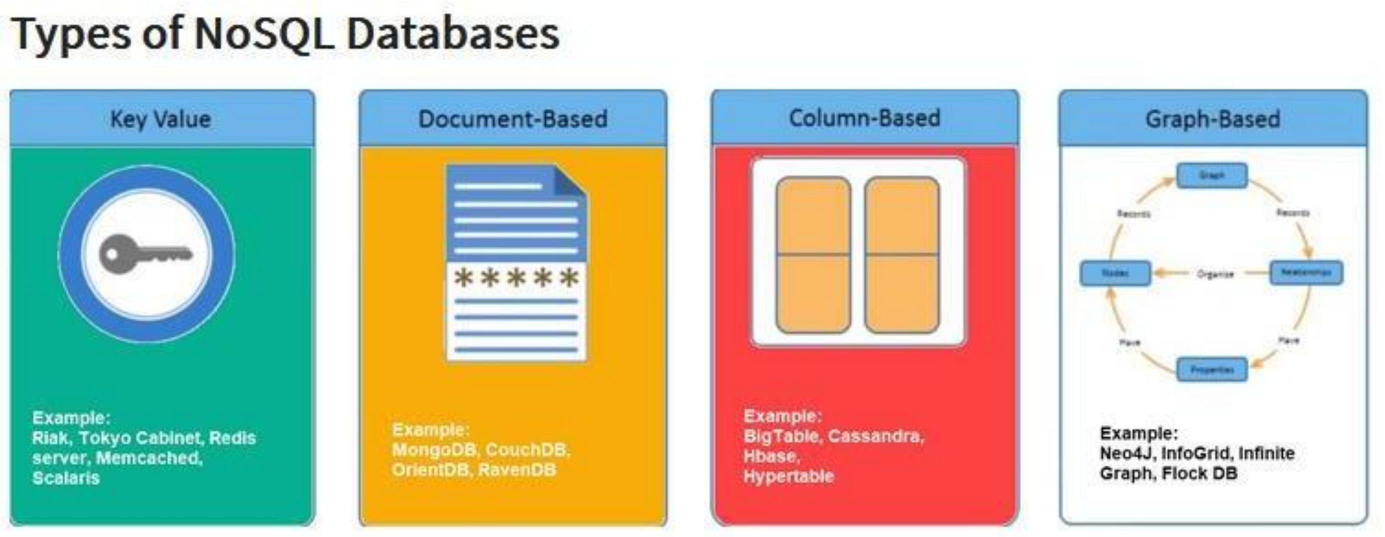
The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

To resolve this problem, we could "scale up" our systems by upgrading our existing hardware. This process is expensive.

Difference Between SQL and NoSQL



SQL	NOSQL
Relational Database management system	Distributed Database management system
Vertically Scalable	Horizontally Scalable
Fixed or predifined Schema	Dynamic Schema
Not suitable for hierarchical data storage	Best suitable for hierarchical data storage
Can be used for complex queries	Not good for complex queries


Types of NoSQL Databases
Key Value
Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris
Document-Based
Example:
MongoDB, CouchDB, OrientDB, RavenDB
Column-Based
Example:
BigTable, Cassandra, Hbase, Hypertable
Graph-Based
Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

MongoDB

Scalable High-Performance Open-source, Document-orientated database.

• Built for Speed

• Rich Document based queries for Easy readability.

• Full Index Support for High Performance.

• Replication and Failover for High Availability.

• Auto Sharding for Easy Scalability.

• Map / Reduce for Aggregation.

Advantages of MongoDB

- Schema less : Number of fields, content and size of the document can be differ from one document to another.
- No complex joins
- Data is stored as JSON style
- Index on any attribute
- Replication and High availability

Mongo DB Terminologies for RDBMS concepts

RDBMS	MongoDB
Database	**Database**
Table, View	**Collection**
Row	**Document (JSON, BSON)**
Column	**Field**
Index	**Index**
Join	**Embedded Document**
Foreign Key	**Reference**
Partition	**Shard**

Data Types of MongoDB

- String : This is most commonly used datatype to store the data. String in mongodb must be UTF-8 valid.
- Integer : This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- Boolean : This type is used to store a boolean (true/ false) value.
- Double : This type is used to store floating point values.
- Min/ Max keys : This type is used to compare a value against the lowest and highest BSON elements.
- Arrays : This type is used to store arrays or list or multiple values into one key.
- Timestamp : ctimestamp. This can be handy for recording when a document has been modified or added.
- Object : This datatype is used for embedded documents.
- Null : This type is used to store a Null value.
- Symbol : This datatype is used identically to a string however, it's generally reserved for languages that use a specific symbol type.
- Date : This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- Object ID : This datatype is used to store the document’s ID.
- Binary data : This datatype is used to store binay data.
- Code : This datatype is used to store javascript code into document.
- Regular expression : This datatype is used to store regular expression

Basic Database Operations

- use *<database name>*
  switched to database provided with command
- db
  To check currently selected database use the command db
- show dbs
  Displays the list of databases
- db.dropDatabase()

  To Drop the database
- db.createCollection (name)
- Ex:- db.createCollection(Stud)
  - To create collection
- >show collections
  - List out all names of collection in current database
- db.*databasename*.insert
- ({Key : Value})
- Ex:- db.Stud.insert({{Name:”Jiya”})
  - In mongodb you don't need to create collection. MongoDB creates collection automatically, when you insert some document.
- db.collection.drop() Example:- db.Stud.drop()

MongoDB's db.collection.drop() is used to drop a collection from the database.

CRUD Operations:

- Insert
- Find
- Update
- Delete

CRUD Operations – Insert

The insert() Method:- To insert data into MongoDB collection, you need to use MongoDB's insert() or save()method.

Syntax

```
>db.COLLECTION_NAME.insert(document)
```

Example

```
>db.stud.insert({name: “Jiya”, age:15})
```

_id Field

- If the document does not specify an *_id* field, then MongoDB will add the _id field and assign a unique *ObjectId* for the document before inserting.
- The _id value must be unique within the collection to avoid duplicate key error.

Insert a Document without Specifying an _id Field

- db.stud.insert({ Name : “Reena", Rno: 15 })
- db.stud.find()

  { "_id" : "5063114bd386d8fadbd6b004”, “Name” : “Reena", “Rno”: 15 }

Insert a Document Specifying an _id Field

- db.stud.insert({ _id: 10, Name : “Reena", Rno: 15 })
- db.stud.find()

{ "_id" : 10, “Name” : “Reena", “Rno”: 15 }

Insert Single Documents

```
db.stud.insert ( {Name: “Ankit”, Rno:1, Address: “Pune”} )
```

Insert Multiple Documents

```
db.stud.insert ( [
 { Name: “Ankit”, Rno:1, Address: “Pune”} ,
 { Name: “Sagar”, Rno:2},
 { Name: “Neha”, Rno:3}
 ] )
```

Insert Multicolumn attribute

```
db.stud.insert( {
   Name: “Ritu",
   Address: { City: “Pune", State: “MH” },
  Rno: 6
 })
```

Insert Multivalued attribute

```
db.stud.insert( { Name
   : “Sneha",
   Hobbies: [“Singing”, “Dancing” , “Cricket”] ,
   Rno:8
 })
```

Insert Multivalued with Multicolumn attribute

```
db.stud.insert(  {
    Name : “Sneha",
         Awards: [ { Award : “Dancing”, Rank: “1st”, Year: 2008 },
                   {Award : “Drawing”, Rank: “3rd”, Year: 2010 } ,
                   {Award : “Singing”, Rank: “1st”, Year: 2015 } ],
   Rno: 9  })
```

CRUD Operations – **Find**

The find() Method- To display data from MongoDB collection. Displays all the documents in a non structured way.

Syntax> db.COLLECTION_NAME.find()

The pretty() Method- **To display the results in a formatted way, you can use** pretty() **method.**

Syntax

```
>db. COLLECTION_NAME.find().pretty()
```

Specify Equality Condition

use the query document { <field>: <value> }

Examples:

- db.stud.find(name: "Jiya" })
- db.stud.find({ _id: 5 })

Comparison Operators

Operator	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	values that are greater than or equal to a specified value.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

Find Examples with comparison operators

- db.stud.find({ rno: { $gt:5} }) *Shows all documents whose rno>5*
- db.stud.find({ rno: { $gt: 0, $lt: 5} }) *Shows all documents whose rno greater than 0 and less than 5*

Examples to show only particular columns

- db.stud.find({name: "Jiya"},{Rno:1}) *To show the rollno of student whose name is equal to Jiya (by default _id is also shown)*
- db.stud.find({name: "jiya"},{_id:0,Rno:1}) *show the rollno of student whose name is equal to Jiya (_id is not shown)*

Examples for Sort function

- db.stud.find().sort({ Rno: 1 })
  Sort on age field in Ascending order (1)
- db.stud.find().sort({ Rno: -1 })
  Sort on age field in Ascending order(-1)

Examples of Count functions

- db.stud.find().count()
  Returns no of documents in the collection

Examples of limit and skip

- db.stud.find().limit(2)
  Returns only first 2 documents
- db.stud.find().skip(5)
  Returns all documents except first 5 documents

CRUD Operations – Update

Syntax db.*CollectionName*.update

```
(
  <query/Condition>,
  <update with $set or $unset>,
  {
    upsert: <boolean>,
    multi: <boolean>,
  } )
```

upsert

- If set to *True*, creates new document if no matches found.

multi

- If set to *True*, updates multiple documents that matches the query criteria

CRUD Operations – Update Examples

1**>** Set age = 25 where id is 100, First Whole document is replaced where condition is matched and only one field is remained as age:25

```
db.stud.update(
{ _id: 100 },
{ age: 25})
```

2> Set age = 25 where id is 100, Only the age field of one document is updated where condition is matched .

```
db.stud.update(
{ _id: 100 },
{ $set:{age: 25}})
```

3> To remove a age column from single document where id=100 db.stud.update(

```
{ _id: 100 },
{ $unset:{age: 1}})
```

CRUD Operations – Remove

- **Remove All Documents**
  - db.inventory.remove({})
- **Remove All Documents that Match a Condition**

    - db.inventory.remove ({ type : "food" })
- **Remove a Single Document that Matches a Condition**
    - db.inventory.remove ({ type : "food" }, 1)

Conclusion: In this way, we have executed MongoDB queries by using CRUD operations.

EXPERIMENT NO. 4 (Group A)

Aim: 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(Roll_no, Name, Date_of_Issue, NameofBook, Status)
2. Fine(Roll_no,Date,Amt)
   - Accept roll_no &amp; 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&gt;30, per day fine will be Rs 50 per day &amp; 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.

- **Outcome:** At end of this experiment, student will be able to study of PL/SQL procedure.
- **Hardware Requirement:** Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.
- **Software Requirement:** MySQL
- **Theory:**

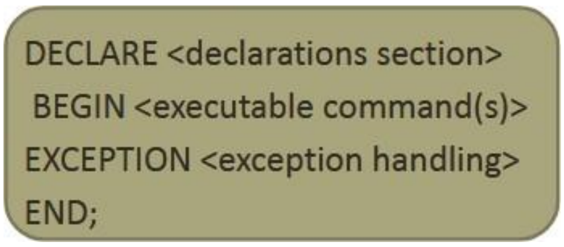
PL/SQL Introduction

•PL/SQL is a combination of SQL along with the procedural features of programming languages

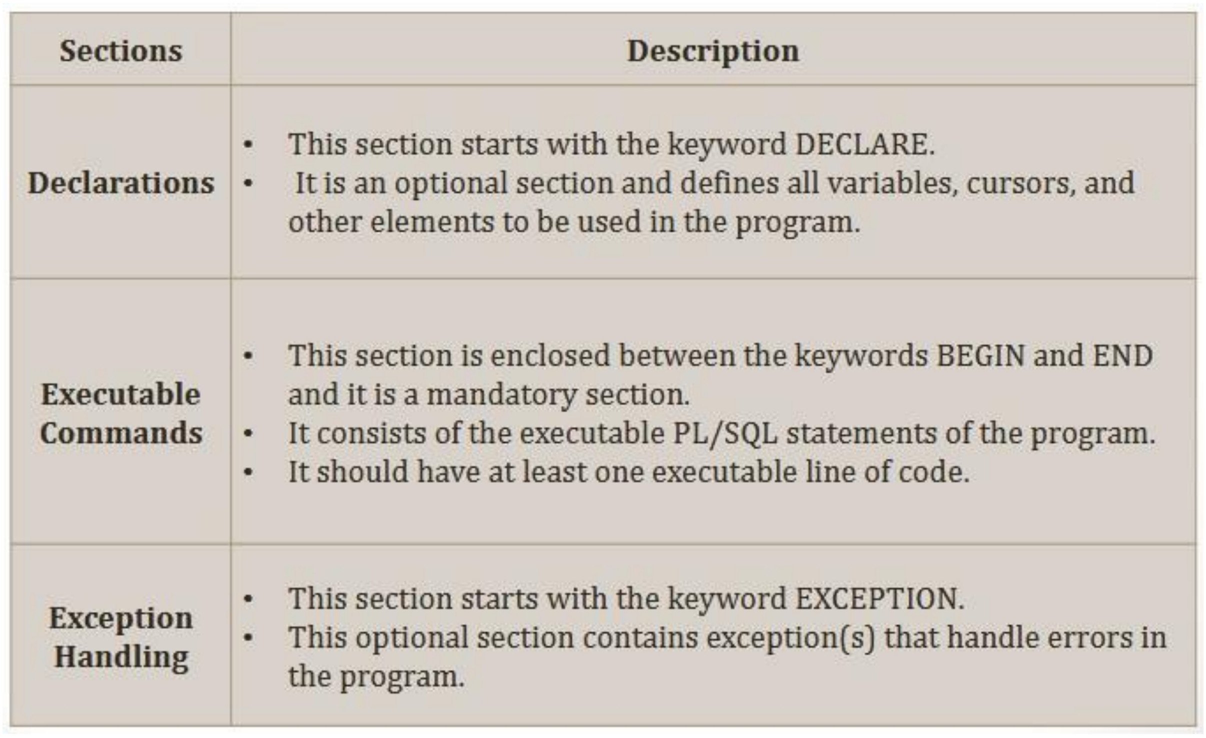
•Basic Syntax of PL/SQL which is a block-structured language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts

•Every PL/SQL statement ends with a semicolon (;)

•Following is the basic structure of a PL/SQL block



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



Sections	Description
Declarations	• This section starts with the keyword DECLARE. • It is an optional section and defines all variables, cursors, and other elements to be used in the program.
Executable Commands	• This section is enclosed between the keywords BEGIN and END and it is a mandatory section. • It consists of the executable PL/SQL statements of the program. • It should have at least one executable line of code.
Exception Handling	• This section starts with the keyword EXCEPTION. • This optional section contains exception(s) that handle errors in the program.

Anonymous blocks: Unnamed

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them.

They need to be created and used in the same session because they will not be stored in the server as a database objects.

Since they need not to store in the database, they need no compilation steps.

They are written and executed directly, and compilation and execution happen in a single process.

Named blocks:

Named blocks are having a specific and unique name for them.They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present in the server.

Named blocks are basically of two types:

1. Procedure
2. Function

Stored Procedure Syntax

CREATE PROCEDURE sp_name([proc_parameter: [IN | OUT | INOUT] param_namedata_type])

Begin<Declare variable_namedata_type;>

<Control Statements/loops> SQL

executable statements; End

Stored Procedure-Parameters

IN –is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure.

OUT –the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program

INOUT –an INOUT parameter is the combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter and pass the new value back to the calling program.

The IN parameter example

```
Mysql> DELIMITER  //
Mysql> CREATE PROCEDURE Allstud(IN SNameVARCHAR(25)) BEGIN
SELECT *FROM stud where Name=SName; END
//
Mysql> DELIMITER ;
Mysql> call Allstud('Reena');
```

The IN and OUT parameter example

```
Mysql> CREATE PROCEDURE Allstud(IN Rno1 int,OUT SName VARCHAR) BEGIN
SELECT Name into SName FROM stud where Rno=RNo1; END
//
Mysql> call Allstud(2,@SName)// Mysql>
select @Sname
```

Date Related Functions required to solve the assignment

CURDATE() –

•The CURDATE() function returns the current date.
Note:This function returns the current date as a "YYYY-MM-DD" format
•Example
•Return current date:
•SELECT CURDATE();

DATEDIFF()

The DATEDIFF() function returns the difference in days between two date values.

•Syntax

•DATEDIFF(date1, date2)

•Example

•Return the difference in days between two date values:

•SELECT DATEDIFF("2017-06-25", "2017-06-15");

Exception handling

Declaring a handler

•To declare a handler, you use the statement as follows:

•DECLARE action HANDLER FOR condition_value statement;

•If a condition whose value matches the condition_value, MySQL will execute the statement and continue or exit the current code block based on the action .

•The action accepts one of the following values:

1. CONTINUE :the execution of the enclosing code block (BEGIN ... END) continues.
2. EXIT : the execution of the enclosing code block, where the handler is declared, terminates.

Conclusion: In this way, we have implemented Pl/SQL procedure.

EXPERIMENT NO. 5 (Group A)

Aim: Design the Database Cursor**.** Implement all types: Implicit, Explicit, Cursor FOR Loop, and parameterized Cursor.

Outcome: At end of this experiment, student will be able to study of database cursor.

Hardware Requirement: Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.

Software Requirement: Oracle SQL

Theory:

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

Types of Cursors:

There are two types of cursor namely as,

1) **Implicit cursor.**
2) **Explicit cursor.**

Why do we need the Cursors?

- **SELECT** statement should return only one row at a time in previous PL/SQL programs. This is too restrictive in many applications. We use the idea of Cursor to handle the above problem.

Implicit Cursor:

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

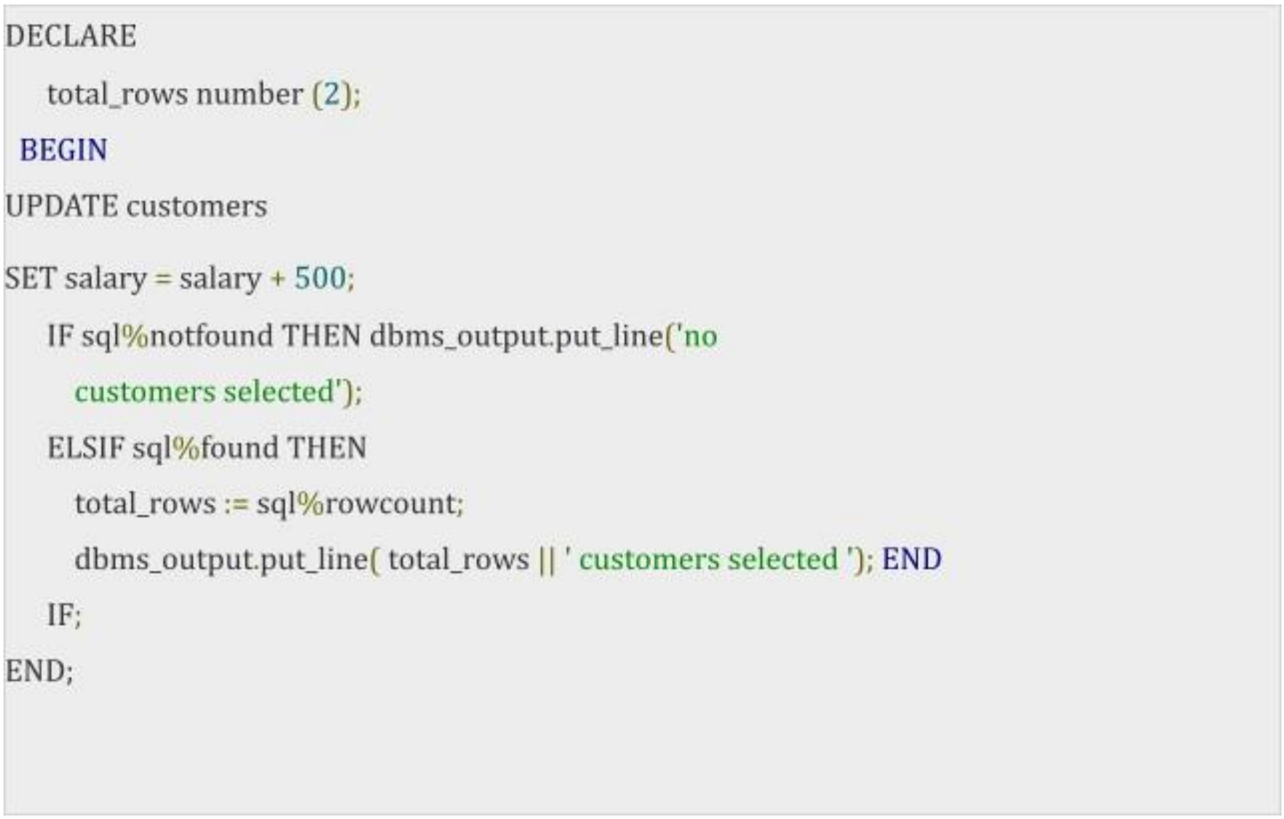
In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes.

Sr No.	Attributes and Description

1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Syntax:



```
DECLARE
   total_rows number (2);
 BEGIN
UPDATE customers
SET salary = salary + 500;
   IF sql%notfound THEN dbms_output.put_line('no
     customers selected');
   ELSIF sql%found THEN
     total_rows := sql%rowcount;
     dbms_output.put_line( total_rows || ' customers selected '); END
   IF;
END;
```

Explicit Cursors:

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is,

```
DECLARE cursor_name CURSOR FOR select_statement;
```

Working with an explicit cursor includes the following 4 steps

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
DECLARE c_customers CURSOR FOR
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time.

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory.

```
CLOSE c_customers;
```

Cursors with Parameters:

1) We can pass parameters into a cursor and use them in the query.

2) We can only pass values to the cursor; and cannot pass values out of the cursor through parameters.

3) Only the data type of the parameter is defined, not its length.

4) Optionally, we can also give a default value for the parameter, which will take effect if no value is passed to the cursor.

Cursors with Parameters Example

```
DECLARE
cur_emp (par_dept VARCHAR2) CURSOR FOR SELECT ename,
salary
FROM    emp    WHERE
deptno    =    par_dept;
Open cur_emp(5);
. . .
```

Declaring variables in a Cursor:

In native SQL, the SELECT list may contain both columns and expressions. In PL/SQL, the SELECT list may contain PL/SQL variables, expressions, and even functions as well as host language bind variables (> PL/SQL 2.1).

```
DECLARE
                project_bonus NUMBER := 1000;
```

Conclusion: In this way, we have designed database cursor and implemented all types of cursors.

EXPERIMENT NO. 7(A) (Group A)

Aim: Write a program to implement MySQL/Oracle database connectivity with any frontend language to implement Database navigation operations (add, delete, edit etc.)

Outcome: At end of this experiment, student will be able to implement database connectivity with any frontend language

Hardware Requirement: Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.

Software Requirement: Oracle SQL

Theory:

Step1 :
place the simple.php & config1.php file in the C:\xampp\htdocs\ in htdocs folder

Step2:
In http://localhost/phpmyadmin/ create the any database as per front end (simple.php) form having first_name, last_name ,email_id, address etc. attributes.

Step 3:
now run the http://localhost/simple.php code

EXPERIMENT NO. 7(B) (Group A)

Aim: Write a program to implement MogoDB database connectivity with PHP

Objective: 1. To learn and understand MongoDB connectivity with PHP.

2. To implement select,insert,update,delete operations using MongoDB in php.

Hardware requirements:

- Any CPU with Pentium Processor or similar, 256 MB
- RAM or more, 1 GB Hard Disk or more.

Software requirements:

Ubuntu 14.04 or Windows 7, PHP5.2, Any Javascript enabled browser.

Theory:

To use MongoDB with PHP, we need to install additional MongoDB PHP driver.

Make a Connection and Select a Database

To make a connection, you need to specify the database name, if the database doesn't exist then MongoDB creates it automatically.

Following is the code snippet to connect to the database −

```
<?php
  // connect to mongodb
  $m = new MongoClient();

  echo "Connection to database successfully";
  // select a database
  $db = $m->mydb;

  echo "Database mydb selected";
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
```

Create a Collection

Following is the code snippet to create a collection −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
	
   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->createCollection("mycol");
   echo "Collection created succsessfully";
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection created succsessfully
```

Insert a Document

To insert a document into MongoDB, insert() method is used.

Following is the code snippet to insert a document −

```
<?php
   // connect to mongodb
   $m = new MongoClient();
   echo "Connection to database successfully";
	
   // select a database
   $db = $m->mydb;
   echo "Database mydb selected";
   $collection = $db->mycol;
   echo "Collection selected succsessfully";

        $document =
     array( "title" =>
     "MongoDB",
     "description" =>
     "database", "likes" =>
     100,
     "url" =>
     "http://www.mongo.com/mongodb/", "by",
     "JIT"
   );

   $collection->insert($document);
   echo "Document inserted successfully";
 ?>
```

When the program is executed, it will produce the following result –

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document inserted successfully
```

Find All Documents

To select all documents from the collection, find() method is used.

Following is the code snippet to select all documents –

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";

  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
```

```
    echo "Collection selected succsessfully";


    $cursor = $collection->find();
    // iterate cursor to display title of documents


          foreach ($cursor as
      $document) { echo
      $document["title"] . "\n";
    }
  ?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
    Collection selected
  succsessfully { "title":
  "MongoDB"
}
```

Update a Document

To update a document, you need to use the update() method.

In the following example, we will update the title of inserted document to MongoDB. Following is the code snippet to update a document −

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";


  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succsessfully";


  // now update the document
        $collection->update(array("title"=>"MongoDB"),
    array('$set'=>array("title"=>"MongoDBNew")));
```

```
   echo "Document updated successfully";


   // now display the updated document
   $cursor = $collection->find();


   // iterate cursor to display title of
   documents echo "Updated document";

        foreach ($cursor as
     $document) { echo
     $document["title"] . "\n";
   }
 ?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Document updated successfully
Updated document {
  "title": "MongoDBNew"
}
```

Delete a Document

To delete a document, you need to use remove() method.

In the following example, we will remove the documents that has the title MongoDB Tutorial. Following is the code snippet to delete a document −

```
<?php
  // connect to mongodb
  $m = new MongoClient();
  echo "Connection to database successfully";


  // select a database
  $db = $m->mydb;
  echo "Database mydb selected";
  $collection = $db->mycol;
  echo "Collection selected succsessfully";
```

```
// now remove the document
$collection->remove(array("title"=>"MongoDBNew"),false);
echo "Documents deleted successfully";

// now display the available documents
$cursor = $collection->find();


// iterate cursor to display title of
documents echo "Updated document";

     foreach ($cursor as
  $document) { echo
  $document["title"] . "\n";
}
?>
```

When the program is executed, it will produce the following result −

```
Connection to database successfully
Database mydb selected
Collection selected succsessfully
Documents deleted successfully
```

In the above example, the second parameter is boolean type and used for justOne field of remove() method.

Conclusion: Thus, we learnt and successfully implemented MongoDB database connectivity with MongoDB.