I²IT INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

Innovation & Leadership

P-14, Rajiv Gandhi Infotech Park, Phase – 1, Hinjawadi, Pune – 411057, India

Department of Information Technology

Academic Year 2017-18

TE (IT) SEM I

Lab Manual

414459: Software Laboratory - I

GROUP A: INTRODUCTION TO DATABASES

ASSIGNMENT NO.1

TITLE: Study and design a database with suitable example using following database systems:

- **Relational: SQL / PostgreSQL / MySQL**
- **Key-value: Riak / Redis**
- **Columnar: Hbase**
- **Document: MongoDB / CouchDB**
- **Graph: Neo4J**

Compare the different database systems based on points like efficiency, scalability, characteristics and performance.

OBJECTIVE:

1. To understand the different type of databases.
2. To understand the difference between the different database systems considering efficiency, scalability etc.

THEORY:

Relational

MySQL: MySQL, the most popular Open Source SQL database management system, is developed, distributed, and supported by Oracle Corporation.

- **MySQL is a database management system.**

A database is a structured collection of data. It may be anything from a simple shopping list to a picture gallery or the vast amounts of information in a corporate network. To add, access, and process data stored in a computer database, you need a database management system such as MySQL Server. Since computers are very good at handling large amounts of data, database management systems play a central role in computing, as standalone utilities, or as parts of other applications.

- **MySQL databases are relational.**

A relational database stores data in separate tables rather than putting all the data in one big storeroom. The database structures are organized into physical files optimized for speed. The logical model, with objects such as databases, tables, views, rows, and columns, offers a flexible programming environment

- **MySQL software is Open Source.**

Open Source means that it is possible for anyone to use and modify the software. Anybody can download the MySQL software from the Internet and use it without paying anything. If you wish, you may study the source code and change it to suit your needs. The MySQL software uses the GPL (GNU General Public License).

- **The MySQL Database Server is very fast, reliable, scalable, and easy to use.**

If that is what you are looking for, you should give it a try. MySQL Server can run comfortably on a desktop or laptop, alongside your other applications, web servers, and so

On, requiring little or no attention. If you dedicate an entire machine to MySQL, you can adjust the settings to take advantage of all the memory, CPU power, and I/O capacity available. MySQL can also scale up to clusters of machines, networked together.

MySQL Commands:
Create a database on the sql server.
mysql> create database [databasename];

List all databases on the sql server.
mysql> show databases;

Switch to a database.
mysql> use [db name];

To see all the tables in the db.
mysql> show tables;

To see database's field formats.
mysql> describe [table name];

To delete a db.
mysql> drop database [database name];

To delete a table.
mysql> drop table [table name];

Show all data in a table.
mysql> SELECT * FROM [table name];
Returns the columns and column information pertaining to the designated table.

Key-value:

Riak: Riak is a distributed NoSQL key-value data store that offers high availability, fault tolerance, operational simplicity, and scalability.[4] In addition to the open-source version, it comes in a supported enterprise version and a cloud storage version.
Riak KV :With a key/value design that delivers powerful – yet simple – data models for storing massive amounts of unstructured data, Riak KV is built to handle a variety of challenges facing Big Data applications that include tracking user or session information, storing connected device data and replicating data across the globe.

Riak KV automates data distribution across the cluster to achieve fast performance and robust business continuity with a masterless architecture that ensures high availability, and scales near linearly using commodity hardware so you can easily add capacity without a large operational burden.

Riak TS is the only enterprise-grade NoSQL time series database optimized specifically for IoT and Time Series data. It ingests, transforms, stores, and analyzes massive amounts of time series data.

Features:

- Fault-tolerant availability:
  Riak replicates key/value stores across a cluster of nodes with a default n_val of three. In the case of node outages due to network partition or hardware failures, data can still be written to a neighboring node beyond the initial three, and read-back due to its "masterless" peer-to-peer architecture.
- Queries:
  Riak provides a REST-ful API through HTTP and Protocol Buffers for basic PUT, GET, POST, and DELETE functions. More complex queries are also possible, including secondary indexes, search (via Apache Solr), and MapReduce. MapReduce has native support for both JavaScript (using the SpiderMonkey runtime) and Erlang.
- Predictable latency:
  Riak distributes data across nodes with hashing and can provide latency profile, even in the case of multiple node failures.
- Storage options:
  Keys/values can be stored in memory, disk, or both.
- Multi-datacenter replication
  In multi-datacenter replication, one cluster acts as a "primary cluster." The primary cluster handles replication requests from one or more "secondary clusters" (generally located in other regions or countries). If the datacenter with the primary cluster goes down, a second cluster can take over as the primary cluster.

- Tunable consistency
  Option to choose between eventual and strong consistency for each bucket.

Redis:

Redis is an open source, advanced key-value store and an apt solution for building high-performance, scalable web applications.

Redis has three main peculiarities that sets it apart.

- Redis holds its database entirely in the memory, using the disk only for persistence.

- Redis has a relatively rich set of data types when compared to many key-value data stores.

- Redis can replicate data to any number of slaves.

Redis Advantages

Following are certain advantages of Redis.

- **Exceptionally fast** − Redis is very fast and can perform about 110000 SETs per second, about 81000 GETs per second.

- **Supports rich data types** − Redis natively supports most of the datatypes that developers already know such as list, set, sorted set, and hashes. This makes it easy to solve a variety of problems as we know which problem can be handled better by which data type.

- **Operations are atomic** − All Redis operations are atomic, which ensures that if two clients concurrently access, Redis server will receive the updated value.

- **Multi-utility tool** − Redis is a multi-utility tool and can be used in a number of use cases such as caching, messaging-queues (Redis natively supports Publish/Subscribe), any short-lived data in your application, such as web application sessions, web page hit counts, etc.

Redis Versus Other Key-value Stores

- Redis is a different evolution path in the key-value DBs, where values can contain more complex data types, with atomic operations defined on those data types.

- Redis is an in-memory database but persistent on disk database, hence it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than the memory.

- Another advantage of in-memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk. Thus, Redis can do a lot with little internal complexity.

Columnar:

Hbase: HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.HBase is a data model that is similar to Google’s big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.

Storage Mechanism in HBase

HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:

- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Rowid	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

Features of HBase

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters.

Applications of HBase

- It is used whenever there is a need to write heavy applications.
- HBase is used whenever we need to provide fast random access to available data.
  - Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

Document:

MONGODB:

MongoDB is an open-source document database, and leading NoSQL database. MongoDB is written in c++. MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Below given table shows the relationship of RDBMS terminology with MongoDB

RDBMS	**MongoDB**
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	Mongod
mysql/sqlplus	Mongo

Sample document

Below given example shows the document structure of a blog site which is simply a comma separated key value pair.

```
{
   _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,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you didn't provide then MongoDB provide a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of mongodb server and remaining 3 bytes are simple incremental value.

Any relational database has a typical schema design that shows number of tables and the relationship between these tables. While in MongoDB there is no concept of relationship

Advantages of MongoDB over RDBMS

- Schema less : MongoDB is document database in which one collection holds different different documents. Number of fields, content and size of the document can be differ from one document to another.
- Structure of a single object is clear
- No complex joins
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL
- Tuning
- Ease of scale-out: MongoDB is easy to scale
- Conversion / mapping of application objects to database objects not needed
- Uses internal memory for storing the (windowed) working set, enabling faster access of data

Why should use MongoDB

- Document Oriented Storage : Data is stored in the form of JSON style documents
- Index on any attribute
- Replication & High Availability
- Auto-Sharding
- Rich Queries
- Fast In-Place Updates
- Professional Support By MongoDB

Where should use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

COUCH DB:

CouchDB is a JSON document-oriented database written in Erlang.

- It is a highly concurrent database designed to be easily replicable, horizontally, across numerous devices and be fault tolerant.
- It is part of the NoSQL generation of databases.
- It is an open source Apache foundation project.

- It allows applications to store JSON documents via its RESTful interface.
- It makes use of map/reduce to index and query the database.

Major Benefits of CouchDB

- **JSON Documents** - Everything stored in CouchDB boils down to a JSON document.
- **RESTful Interface** - From creation to replication to data insertion, every management and data task in CouchDB can be done via HTTP.
- **N-Master Replication** - You can make use of an unlimited amount of 'masters', making for some very interesting replication topologies.
- **Built for Offline** - CouchDB can replicate to devices (like Android phones) that can go offline and handle data sync for you when the device is back online.
- **Replication Filters** - You can filter precisely the data you wish to replicate to different nodes.

Graph:

Neo4j: It is the world's leading open source Graph Database which is developed using Java technology. It is highly scalable and schema free (NoSQL).

What is a Graph Database?
A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. It is composed of two elements - nodes (vertices) and relationships (edges).

Graph database is a database used to model the data in the form of graph. In here, the nodes of a graph depict the entities while the relationships depict the association of these nodes.

Popular Graph Databases

Neo4j is a popular Graph Database. Other Graph Databases are Oracle NoSQL Database, OrientDB, HypherGraphDB, GraphBase, InfiniteGraph, and AllegroGraph.

Why Graph Databases?

Nowadays, most of the data exists in the form of the relationship between different objects and more often, the relationship between the data is more valuable than the data itself.

Relational databases store highly structured data which have several records storing the same type of data so they can be used to store structured data and, they do not store the relationships between the data.

Unlike other databases, graph databases store relationships and connections as first-class entities.

The data model for graph databases is simpler compared to other databases and, they can be used with OLTP systems. They provide features like transactional integrity and operational availability.

RDBMS Vs Graph Database

Following is the table which compares Relational databases and Graph databases.

Sr.No	RDBMS	Graph Database
1	Tables	Graphs
2	Rows	Nodes
3	Columns and Data	Properties and its values
4	Constraints	Relationships
5	Joins	Traversal

Advantages of Neo4j

- **Flexible data model** − Neo4j provides a flexible simple and yet powerful data model, which can be easily changed according to the applications and industries.
- **Real-time insights** − Neo4j provides results based on real-time data.
- **High availability** − Neo4j is highly available for large enterprise real-time applications with transactional guarantees.
- **Connected and semi structures data** − Using Neo4j, you can easily represent connected and semi-structured data.
- **Easy retrieval** − Using Neo4j, you can not only represent but also easily retrieve (traverse/navigate) connected data faster when compared to other databases.
- **Cypher query language** − Neo4j provides a declarative query language to represent the graph visually, using an ascii-art syntax. The commands of this language are in human readable format and very easy to learn.
- **No joins** − Using Neo4j, it does NOT require complex joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without joins or indexes.

Features of Neo4j

- **Data model (flexible schema)** − Neo4j follows a data model named native property graph model. Here, the graph contains nodes (entities) and these nodes are connected with each other (depicted by relationships). Nodes and relationships store data in key-value pairs known as properties.

  In Neo4j, there is no need to follow a fixed schema. You can add or remove properties as per requirement. It also provides schema constraints.
- **ACID properties** − Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.
- **Scalability and reliability** − You can scale the database by increasing the number of reads/writes, and the volume without effecting the query processing speed and data integrity. Neo4j also provides support for **replication** for data safety and reliability.

- **Cypher Query Language** − Neo4j provides a powerful declarative query language known as Cypher. It uses ASCII-art for depicting graphs. Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like Joins.

- **Built-in web application** − Neo4j provides a built-in **Neo4j Browser** web application. Using this, you can create and query your graph data.

- **Drivers** − Neo4j can work with −

    - REST API to work with programming languages such as Java, Spring, Scala etc.

    - Java Script to work with UI MVC frameworks such as Node JS.

    - It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications. In addition to these, you can also work with other databases such as MongoDB, Cassandra, etc.

- **Indexing** − Neo4j supports Indexes by using Apache Lucence.

CONCLUSION: We have studied Different Type of Database Systems and their comparison on points like efficiency, scalability, characteristics and performance etc.

FAQ:

A batch

1. Write the difference between RDBMS and MongoDB
2. What are the advantages of DBMS?
3. Explain the terms table and record in a database.
4. Compare Raik V Redis Database Systems?

B batch

1. Explain the use of MongoDB.
2. What is RDBMS?
3. Explain different data models with example.
4. Compare Neo4j Vs RDBMS

C batch

1. Name the subsystems of RDBMS.
2. How do you communicate with a RDBMS?
3. What is a database transaction?
4. Compare MongoDB VS CouchDB?

D batch

1. What are the advantages of DBMS over traditional file system?
2. Explain the differences between structured data and unstructured data.
3. What are the major functions of database administrator?
4. Compare HBASE Vs RDBMS?

ASSIGNMENT NO.2

TITLE: Install and configure client and server for MySQL and MongoDB (Show all commands and
necessary steps for installation and configuration).

OBJECTIVE:

1. To understand the Installation and configuration of database in client-server architecture.

Theory:

Steps in installation of MySQL Server and Configuration of Client

[Note:

1. ***I have implemented all following steps on Fedora 19 x86_64 machine.***
2. ***All steps to be implemented as 'root' login.***
3. ***Internet connection is must.***
4. ***Update your machine using command 'yum update'.]***

Steps 1 to 5 to be implemented on both server & client:

1. Set manual IP addresses for both server and client machines.

E.g. Server : 10.5.2.47
Client : 10.5.2.X

2. Enable MySQL services.

# systemctl enable mysqld.service

3. Start MySQL services.

# systemctl start mysqld.service

4. Install 'Community MySQL Server' package.

# yum install community-mysql-server
# mysql_secure_installation

(Set root password, remove anonymous users, disallow remote login. Say 'Yes' to all.)
(Use 'systemctl stop mysqld.service' to stop MySQL service.)

5. Disable firewall. *[MOST IMPORTANT STEP!!]*

# systemctl disable firewalld # systemctl stop firewalld

On server:

1. Login with MySQL.

# mysql –h localhost –u root –p
Password:
(Enter password.)

2. Create new user(s) for client.
3. Grant all privileges on database to user.

> create user 'user001'@'%' identified by 'user001';
*> grant all on *.* to 'user001'@'%';*

4. Exit from MySQL.

```
> exit
```

5. Restart MySQL service.

```
# systemctl stop mysqld.service
# systemctl start mysqld.service
```

On Client:

1. Locate ‘my.cnf’ file.

```
# locate my.cnf
```

(It results in ‘/etc/my.cnf’.)

2. Add following line below ‘[mysqld]’ line using editor.

```
bind-address = 10.5.2.47
```

(Comment ‘skip-networking’ if exists. i.e., ‘#skip-networking’.)

3. Restart MySQL service.

```
# systemctl stop mysqld.service
# systemctl start mysqld.service
```

(This step may give you an error message. Ignore it.)

4. Log in to the remote server.

```
# mysql –h 10.5.2.47 –u user001 –p user001_db
Password:
```

(Enter password.)

Installing MongoDB

Installing MongoDB is a simple process on most platforms. Precompiled binaries are available for Linux, Mac OS X, Windows, and Solaris. This means that, on most platforms, you can download the archive from *http://www.mongodb.org*, inflate it, and run the binary. The MongoDB server requires a directory it can write database files to and a port it can listen for connections on. This section covers the entire install on the two variants of system: Windows and everything else (Linux, Max, Solaris).
When we speak of “installing MongoDB,” generally what we are talking about is setting up mongod, the core database server. mongod is used in a single-server setup as either master or slave, as a member of a replica sets, and as a shard.

Stepwise Installation

1) You must create a directory for the database to put its files. By default, the database will use */data/db,* although you can specify any other directory.
You can create the directory and set the permissions by running the following:

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

mkdir -p creates the directory and all its parents, if necessary (i.e., if the */data* directory didn’t exist, it will create the /data directory and then the */data/db* directory). chown changes the ownership of */data/db* so that your user can write to it.

2)Decompress the *.tar.gz* file you downloaded from *http://www.mongodb.org*.

```
$ tar zxf mongodb-linux-i686-1.6.0.tar.gz
$ cd mongodb-linux-i686-1.6.0
```

3)Now you can start the database:

```
$ bin/mongod
```

Or if you’d like to use an alternate database path, specify it with the --dbpath option:

```
$ bin/mongod --dbpath ~/db
```

Conclusion: We have studied how to install and configure mysql and mongodb on fedora 20

ASSIGNMENT NO.3

TITLE: Study the SQLite database and its uses. Also elaborate on building and installing of SQLite.

OBJECTIVE:

1. To understand the SQLite Database and Their Installation on Fedora 20.

Theory:

SQLite:

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is a database, which is zero-configured, which means like other databases you do not need to configure it in your system.

SQLite engine is not a standalone process like other databases, you can link it statically or dynamically as per your requirement with your application. SQLite accesses its storage files directly.

Features:

- SQLite does not require a separate server process or system to operate (serverless).
- SQLite comes with zero-configuration, which means no setup or administration needed.
- A complete SQLite database is stored in a single cross-platform disk file.
- SQLite is very small and light weight, less than 400KiB fully configured or less than 250KiB with optional features omitted.
- SQLite is self-contained, which means no external dependencies.
- SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.
- SQLite supports most of the query language features found in SQL92 (SQL2) standard.
- SQLite is written in ANSI-C and provides simple and easy-to-use API.
- SQLite is available on UNIX (Linux, Mac OS-X, Android, iOS) and Windows (Win32, WinCE, WinRT).

SQLite A Brief History

- 2000 - D. Richard Hipp designed SQLite for the purpose of no administration required for operating a program.
- 2000 - In August, SQLite 1.0 released with GNU Database Manager.

- 2011 - Hipp announced to add UNQl interface to SQLite DB and to develop UNQLite (Document oriented database).

Situations Where SQLite Works Well

- **Embedded devices and the internet of things**

  Because an SQLite database requires no administration, it works well in devices that must operate without expert human support. SQLite is a good fit for use in cellphones, set-top boxes, televisions, game consoles, cameras, watches, kitchen appliances, thermostats, automobiles, machine tools, airplanes, remote sensors, drones, medical devices, and robots: the "internet of things".

- **Application file format**

  SQLite is often used as the on-disk file format for desktop applications such as version control systems, financial analysis tools, media cataloging and editing suites, CAD packages, record keeping programs, and so forth. There are many benefits to this approach, including improved application performance, reduced cost and complexity, and improved reliability..

- **Websites**

  SQLite works great as the database engine for most low to medium traffic websites (which is to say, most websites). The amount of web traffic that SQLite can handle depends on how heavily the website uses its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine with SQLite. The 100K hits/day figure is a conservative estimate, not a hard upper bound. SQLite has been demonstrated to work with 10 times that amount of traffic.

- **Data analysis**

  People who understand SQL can employ the sqlite3 command-line shell (or various third-party SQLite access programs) to analyze large datasets. Raw data can be imported from CSV files, then that data can be sliced and diced to generate a myriad of summary reports. More complex analysis can be done using simple scripts written in Tcl or Python (both of which come with SQLite built-in) or in R or other languages using readily available adaptors.

- **Cache for enterprise data**

  Many applications use SQLite as a cache of relevant content from an enterprise RDBMS. This reduces latency, since most queries now occur against the local cache and avoid a network round-trip. It also reduces the load on the network and on the central database server.

- **Server-side database**

  Systems designers report success using SQLite as a data store on server applications running in the data center, or in other words, using SQLite as the underlying storage engine for an application-specific database server.

- **File archives**

  The SQLite Archiver project shows how SQLite can be used as a substitute for ZIP archives or Tarballs. An archive of files stored in SQLite is only very slightly larger, and in some cases actually smaller, than the equivalent ZIP archive. And an SQLite archive features incremental and atomic updating and the ability to store much richer metadata.

- **Replacement for *ad hoc* disk files**

  Many programs use fopen(), fread(), and fwrite () to create and manage files of data in home-grown formats. SQLite works particularly well as a replacement for these *ad hoc* data files.

- **Internal or temporary databases**

  For programs that have a lot of data that must be sifted and sorted in diverse ways, it is often easier and quicker to load the data into an in-memory SQLite database and use queries with joins and ORDER BY clauses to extract the data in the form and order needed rather than to try to code the same operations manually. Using an SQL database internally in this way also gives the program greater flexibility since new columns and indices can be added without having to recode every query.

- **Stand-in for an enterprise database during demos or testing**

  Client applications typically use a generic database interface that allows connections to various SQL database engines. It makes good sense to include SQLite in the mix of supported databases and to statically link the SQLite engine in with the client. That way the client program can be used standalone with an SQLite data file for testing or for demonstrations.

- **Education and Training**

  Because it is simple to setup and use (installation is trivial: just copy the **sqlite3** or **sqlite3.exe** executable to the target machine and run it) SQLite makes a good database engine for use in teaching SQL. Students can easily create as many databases as they like and can email databases to the instructor for comments or grading. For more advanced students who are interested in studying how an RDBMS is implemented, the modular and well-commented and documented SQLite code can serve as a good basis.

- **Experimental SQL language extensions**

  The simple, modular design of SQLite makes it a good platform for prototyping new, experimental database language features or ideas.

Installation on Fedora 20

```
root:~$ sudo dnf install sqlite
root:~$ sudo dnf install sqlite-devel sqlite-tcl sqlite-jdbc
```

To start with SQLite

```
root:~$ $sqlite3
SQLite version 3.3.6
Enter ".help" for instructions
sqlite>
```

SQLite Commands

The standard SQLite commands to interact with relational databases are similar to SQL. They are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP.

In SQLite, **sqlite3** command is used to create a new SQLite database.

```
$sqlite3 DatabaseName.db
```

Following is the basic syntax of CREATE TABLE statement.

```
Syntax:CREATE TABLE database_name.table_name(
   column1 datatype PRIMARY KEY(one or more columns),
   column2 datatype,
   column3 datatype,
   .....
   columnN datatype,
);
E.g:  sqlite>

CREATE TABLE COMPANY(
   ID INT PRIMARY KEY     NOT NULL,
   NAME           TEXT    NOT NULL,
   AGE            INT     NOT NULL,
   ADDRESS        CHAR(50),
   SALARY         REAL
);
```

CONCLUSION: We have studied SQLite database and Its Uses.

FAQ:

A batch

1) What is the difference between SQL and SQLite?
2) List out the standard SQLite commands?
3) How would you drop a table in SQLite database?

B batch

1) List out the advantages of SQLite?
2) How to insert data in a table in SQLite?
3) What is the use of UPADTE query in SQLIte?

C batch

1) Mention what are the SQLite storage classes?
2) How would you create a database in SQLite?
3) How can you delete the existing records from a table in SQLite?

D batch

1|)Mention what is the command used to create a database in SQLite?
2)How would you create a table in SQLite database?
3) What is the use of UPADTE query in SQLIte?

GROUP B: SQL and PL/SQL

ASSIGNMENT NO.1

TITLE: Design any database with at least 3 entities and relationships between them. Apply DCL and DDL commands. Draw suitable ER/EER diagram for the system.

OBJECTIVE:

1. To understand the DDL commands and DCL commands.

THEORY:

SQL stands for Structured Query Language.
SQL lets you access and manipulate databases.
SQL is an ANSI (American National Standards Institute) standard.
SQL is structure query language.SQL contains different data types those are

1. char(size)
2. varchar2(size)
3. date
4. number(p,s)
5. long
6. raw/long raw

Different types of commands in SQL:

A). **DDL commands: -** To create a database objects
B). **DML commands: -** To manipulate data of a database objects
C). **DQL command: -** To retrieve the data from a database.
D). **DCL command -** To control the data of a database…

DDL commands:

The DDL part of SQL permits database tables to be created or deleted. It also define indexes (keys), specify links between tables, and impose constraints between tables.

- **Create Database:**

You would need special privileges to create or to delete a MySQL database. So assuming you have access to root user, you can create any database using MySQL.
Example:
Here is a simple example to create database called **TUTORIALS**:

```
MySQL> create TUTORIALS
```

This will create a MySQL database TUTORIALS.

- **Delete / Drop Database:**

You would need special privileges to create or to delete a MySQL database. So assuming you have access to root user, you can create any database using MySQL . Be careful while deleting any database because you will lose your all the data available in your database.
Here is an example to delete a database created:

```
MySQL > drop TUTORIALS
```

This will give you a warning and it will confirm if you really want to delete this database or not.

```
Dropping the database is potentially a very bad thing to do.
Any data stored in the database will be destroyed.

Do you really want to drop the 'TUTORIALS' database [y/N] y
Database "TUTORIALS" dropped
```

- **Selecting Databases:**

Once you get connection with MySQL server, it is required to select a particular database to work with. This is because there may be more than one database available with MySQL Server.
Selecting MySQL Database from Command Prompt:
This is very simple to select a particular database from mysql> prompt. You can use SQL command **use** to select a particular database.

Example:
Here is an example to select database called **TUTORIALS**:

```
mysql> use TUTORIALS;
Database changed
mysql>
```

Now, you have selected TUTORIALS database and all the subsequent operations will be performed on TUTORIALS database.

- **Creating Table/Relation:**

  1. **The Create Table Command: -** it defines each column of the table uniquely. Each column has minimum of three attributes, a name , data type and size.

The table creation command requires:

- Name of the table
- Names of fields
- Definitions for each field

Syntax:
Here is generic SQL syntax to create a MySQL table:

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

Now, we will create following table in **TUTORIALS** database.

```
CREATE TABLE 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 few items need explanation:

- Field Attribute **NOT NULL** is being used because we do not want this field to be NULL. So if user will try to create a record with 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 primary key. You can use multiple columns separated by comma to define a primary key.

Creating Tables from Command Prompt:
This is easy to create a MySQL table from mysql> prompt. You will use SQL command **CREATE TABLE** to create a table.
Example:
Here is an example, which creates **tutorials_tbl**:

```
mysql> use TUTORIALS;
Database changed
mysql> CREATE TABLE 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 )
  -> );
Query OK, 0 rows affected (0.16 sec)
mysql>
```

NOTE: MySQL does not terminate a command until you give a semicolon (;) at the end of SQL command.

● Deleting / Dropping Table :

It is very easy to drop an existing MySQL table, but you need to be very careful while deleting any existing table because data lost will not be recovered after deleting a table.
Syntax:
Here is generic SQL syntax to drop a MySQL table:

```
DROP TABLE table_name ;
```

Dropping Tables from Command Prompt:
This needs just to execute **DROP TABLE** SQL command at mysql> prompt.
Example:
Here is an example, which deletes **tutorials_tbl**:

```
mysql> use TUTORIALS;
Database changed
mysql> DROP TABLE tutorials_tbl
Query OK, 0 rows affected (0.8 sec)
mysql>
```

- **Modifying Relation Schemas /Table Structure**

MySQL **ALTER** command is very useful when you want to change a name of your table, any table field or if you want to add or delete an existing column in a table. Let's begin with creation of a table called **testalter_tbl.**

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> create table testalter_tbl
   -> (
   -> i INT,
   -> c CHAR(1)
   -> );
Query OK, 0 rows affected (0.05 sec)
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| i     | int(11) | YES  |     | NULL    |       |
| c     | char(1) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

Dropping, Adding or Repositioning a Column:

Suppose you want to drop an existing column **i** from above MySQL table then you will use **DROP** clause along with **ALTER** command as follows:

```
mysql> ALTER TABLE testalter_tbl  DROP i;
```

A **DROP** will not work if the column is the only one left in the table.
To add a column, use ADD and specify the column definition. The following statement restores the **i**column to testalter_tbl:

```
mysql> ALTER TABLE testalter_tbl ADD i INT;
```

After issuing this statement, testalter will contain the same two columns that it had when you first created the table, but will not have quite the same structure. That's because new columns are added to the end of the table by default. So even though **i** originally was the first column in mytbl, now it is the last one.

```
mysql> SHOW COLUMNS FROM testalter_tbl;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| c     | char(1) | YES  |     | NULL    |       |
| i     | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

To indicate that you want a column at a specific position within the table, either use FIRST to make it the first column or AFTER col_name to indicate that the new column should be placed after col_name. Try the following ALTER TABLE statements, using SHOW COLUMNS after each one to see what effect each one has:

```
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT FIRST;
ALTER TABLE testalter_tbl DROP i;
ALTER TABLE testalter_tbl ADD i INT AFTER c;
```

The FIRST and AFTER specifiers work only with the ADD clause. This means that if you want to reposition an existing column within a table, you first must DROP it and then ADD it at the new position.

Renaming a Table:
To rename a table, use the **RENAME** option of the ALTER TABLE statement. Try out the following example to rename testalter_tbl to alter_tbl.

```
mysql> ALTER TABLE testalter_tbl RENAME TO alter_tbl;
```

- **Database Constraint:**

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.
Following are commonly used constraints available in SQL.

- **NOT NULL Constraint**: Ensures that a column cannot have NULL value.
- **DEFAULT Constraint**: Provides a default value for a column when none is specified.
- **UNIQUE Constraint**: Ensures that all values in a column are different.
- **PRIMARY Key**: Uniquely identified each rows/records in a database table.
- **FOREIGN Key**: Uniquely identified a rows/records in any another database table.
- **CHECK Constraint**: The CHECK constraint ensures that all values in a column satisfy certain conditions.
- **INDEX**: Use to create and retrieve data from the database very quickly.

NOT NULL Constraint
By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.
A NULL is not the same as no data, rather, it represents unknown data.
Example:
For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

```
CREATE TABLE CUSTOMERS(
     ID  INT           NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT             NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

DEFAULT Constraint:

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value. Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
       ID  INT              NOT NULL,
       NAME VARCHAR (20)    NOT NULL,
       AGE  INT             NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
       PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a DFAULT constraint to SALARY column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint:

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
   ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint:

The UNIQUE Constraint prevents two records from having identical values in a particular column.
For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE CUSTOMERS(
       ID  INT              NOT NULL,
       NAME VARCHAR (20)    NOT NULL,
       AGE  INT             NOT NULL UNIQUE,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
  MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

DROP a UNIQUE Constraint:
To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
  DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax:

```
ALTER TABLE CUSTOMERS
  DROP INDEX myUniqueConstraint;
```

PRIMARY Key:
A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.
A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**.
If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

To create a PRIMARY KEY constraint on the "ID" column when CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

NOTE: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).
For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE CUSTOMERS(
      ID  INT              NOT NULL,
      NAME VARCHAR (20)    NOT NULL,
      AGE  INT             NOT NULL,
      ADDRESS  CHAR (25) ,
      SALARY   DECIMAL (18, 2),
      PRIMARY KEY (ID, NAME)
);
```

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

Delete Primary Key:
You can clear the primary key constraints from the table, Use Syntax:

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;
```

FOREIGN Key:

A foreign key is a key used to link two tables together. This is sometimes called a referencing key.
Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.
The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.
If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s). Example:

Consider the structure of the two tables as follows:

```
CUSTOMERS table:
CREATE TABLE CUSTOMERS(
      ADDRESS  CHAR (25) ,
      SALARY   DECIMAL (18, 2),
      PRIMARY KEY (ID)
);
```

ORDERS table:

```
CREATE TABLE ORDERS (
       ID         INT        NOT NULL,
       DATE       DATETIME,
       CUSTOMER_ID INT references CUSTOMERS(ID),
       AMOUNT     double,
       PRIMARY KEY (ID)
);
```

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
   ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint:
To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE ORDERS
   DROP FOREIGN KEY;
```

CHECK Constraint
The CHECK Constraint enables a condition to check the value being entered into a
record. If the condition evaluates to false, the record violates the constraint and isn't entered into the table.
Example:
For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you can not have any CUSTOMER below 18 years:

```
CREATE TABLE CUSTOMERS(
       ID  INT              NOT NULL,
       NAME VARCHAR (20)    NOT NULL,
       AGE  INT             NOT NULL CHECK (AGE >= 18),
       ADDRESS  CHAR (25) ,
       SALARY  DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a CHECK constraint to
AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

DROP a CHECK Constraint:
To drop a CHECK constraint, use the following SQL. This syntax does not work with MySQL:

```
ALTER TABLE CUSTOMERS
  DROP CONSTRAINT myCheckConstraint;
```

- **Creating Index on Tables:**

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 considered that what are columns which will be used to make SQL queries and create one or more indexes on those columns.
Practically, indexes are also 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 Database Search Engine to locate records very fast.
INSERT and UPDATE statements take more time on tables having indexes where as SELECT statements become fast on those tables. The reason is that while doing insert or update, database need to insert or update index values as well.

The INDEX is used to create and retrieve data from the database very quickly. Index can be created by using single or group of columns in a table. When index is created, it is assigned a ROWID for each row before it sorts out the data.
Proper indexes are good for performance in large databases, but you need to be careful while creating index. Selection of fields depends on what you are using in your SQL queries.
Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

```
CREATE TABLE CUSTOMERS(
      ID  INT           NOT NULL,
      NAME VARCHAR (20)   NOT NULL,
      AGE  INT            NOT NULL,
      ADDRESS  CHAR (25) ,
      SALARY  DECIMAL (18, 2),
      PRIMARY KEY (ID));
```

Now, you can create index on single or multiple columns using the following syntax:

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

To create an INDEX on AGE column, to optimize the search on customers for a particular age, following is the SQL syntax:

```
CREATE INDEX idx_age
   ON CUSTOMERS ( AGE );
```

DROP an INDEX Constraint:
To drop an INDEX constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
  DROP INDEX idx_age;
```

Data Control Language (DCL):

Data Control Language(DCL) is used to control privilege in Database. To perform any operation in the database, such as for creating tables, sequences or views we need privileges. Privileges are of two types,

- **System :** creating session, table etc are all types of system privilege.
- **Object :** any command or query to work on tables comes under object privilege.

DCL defines two commands,

- **Grant :** Gives user access privileges to database.
- **Revoke :** Take back permissions from user.

SQL GRANT Command:
SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

```
GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];
```

- *privilege_name* is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- *object_name* is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- *user_name* is the name of the user to whom an access right is being granted.
- *user_name* is the name of the user to whom an access right is being granted.
- *PUBLIC* is used to grant access rights to all users.
- *ROLES* are a set of privileges grouped together.
- *WITH GRANT OPTION* - allows a user to grant access rights to other users.

E.g:GRANT SELECT ON employee TO user1;

SQL REVOKE Command:

The REVOKE command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

```
REVOKE privilege_name
ON object_name
FROM {user_name |PUBLIC |role_name}
```

E.g.: REVOKE SELECT ON employee FROM user1;

CONCLUSION: We have studied and implemented various DDL and DCL statements in SQL.

FAQ: A batch

1. What are super, primary, foreign and candidate keys?
2. What is the difference between a primary key and a unique constraint?
3. What is DML compiler?

B batch

1. What is a query?
2. Explain different types of SQL statements.
3. What is the use of DCL command?

C batch

1. What is DCL?
2. Explain Different Notations of ER Diagram?
3. Explain the difference between delete, truncate and drop command.

D batch

1. Explain the Referential Integrity Constraint?.
2. What are the properties of a relational table?
3. What is on delete cascade?

ASSIGNMENT NO.2

TITLE: Design and implement a database and apply at least 10 different DML queries for the following task. For a given input string display only those records which match the given pattern or a phrase in the search string. Make use of wild characters and LIKE operator for the same. Make use of Boolean and arithmetic operators wherever necessary.

OBJECTIVE:

1. To understand the DML commands.

THEORY:

DML commands:

DML commands are the most frequently used SQL commands and is used to query and manipulate the existing database objects. Some of the commands are Insert, Select, Update, Delete

Insert Command
This is used to add one or more rows to a table. The values are separated by commas and the data types char and date are enclosed in apostrophes. The values must be entered in the same order as they are defined.

Select Commands
It is used to retrieve information from the table. it is generally referred to as querying the table. We can either display all columns in a table or only specify column from the table.

Update Command
It is used to alter the column values in a table. A single column may be updated or more than one column could be updated.

Delete command
After inserting row in a table we can also delete them if required. The delete command consists of a from clause followed by an optional where clause.

The SQL **SELECT** statement returns a result set of records from one or more tables. A **SELECT** statement retrieves zero or more rows from one or more database tables or database views. In most applications, SELECT is the most commonly used Data Manipulation Language (DML) command. As SQL is a declarative programming language, SELECT queries specify a result set, but do not specify how to calculate it. The database translates the query into a "query plan" which may vary between executions, database versions and database software. This functionality is called the "query optimizer" as it is responsible for

finding the best possible execution plan for the query, within applicable constraints.
The SELECT statement has many optional clauses:

- WHERE specifies which rows to retrieve.
- GROUP BY groups rows sharing a property so that an aggregate function can be applied to each group.
- HAVING selects among the groups defined by the GROUP BY clause.
- ORDER BY specifies an order in which to return the rows.
- AS provides an alias which can be used to temporarily rename tables or columns.

WHERE Clause

We have seen SQL **SELECT** command to fetch data from MySQL table. We can use a conditional clause called **WHERE** clause to filter out results. Using WHERE clause, we can specify a selection criteria to select required records from a table.
Syntax:
Here is generic SQL syntax of SELECT command with WHERE clause to fetch data from MySQL table:

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE condition1 [AND [OR]] condition2.....
```

- You can use one or more tables separated by comma to include various conditions using a WHERE clause, but WHERE clause is an optional part of SELECT command.
- You can specify any condition using WHERE clause.
- You can specify more than one conditions using **AND** or **OR** operators.
- A WHERE clause can be used along with DELETE or UPDATE SQL command also to specify a condition.

The **WHERE** clause works like an if condition in any programming language. This clause is used to compare given value with the field value available in MySQL table. If given value from outside is equal to the available field value in MySQL table, then it returns that row.
Here is the list of operators, which can be used with **WHERE** clause.
Assume field A holds 10 and field B holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater	(A > B) is

	than the value of right operand, if yes then condition becomes true.	not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The WHERE clause is very useful when you want to fetch selected rows from a table, especially when you use **MySQL Join**.
It is a common practice to search records using **Primary Key** to make search fast.
If given condition does not match any record in the table, then query would not return any row.

Fetching Data from Command Prompt:
This will use SQL SELECT command with WHERE clause to fetch selected data from MySQL table tutorials_tbl.
Example:
Following example will return all the records from **tutorials_tbl** table for which author name is **Sanjay**:

```
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl WHERE tutorial_author='Sanjay';
+-------------+----------------+-----------------+-----------------+
| tutorial_id | tutorial_title | tutorial_author | submission_date |
  ----------------------------   ----------------
+                            +++                ----------------  +
|           3 | JAVA Tutorial | Sanjay          | 2007-05-21      |
  ----------------------------   ----------------
+                            +++                ----------------  +
1 rows in set (0.01 sec)
```

```
mysql>
```

Unless performing a **LIKE** comparison on a string, the comparison is not case sensitive. You can make your search case sensitive using **BINARY** keyword as follows:

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed



mysql> SELECT * from tutorials_tbl \
        WHERE BINARY tutorial_author='sanjay';
Empty set (0.02 sec)

mysql>
```

LIKE Clause

We have seen SQL **SELECT** command to fetch data from MySQL table. We can also use a conditional clause called **WHERE** clause to select required records.

A WHERE clause with equals sign (=) works fine where we want to do an exact match. Like if "tutorial_author = 'Sanjay'". But there may be a requirement where we want to filter out all the results where tutorial_author name should contain "jay". This can be handled using SQL **LIKE** clause along with WHERE clause.

If SQL LIKE clause is used along with % characters, then it will work like a meta character (*) in UNIX while listing out all the files or directories at command prompt.

Without a % character, LIKE clause is very similar to equals sign along with WHERE clause.

Syntax:

Here is generic SQL syntax of SELECT command along with LIKE clause to fetch data from MySQL table:

```
SELECT field1, field2,...fieldN table_name1, table_name2...
WHERE field1 LIKE condition1 [AND [OR]] filed2 = 'somevalue'
```

- You can specify any condition using WHERE clause.
- You can use LIKE clause along with WHERE clause.
- You can use LIKE clause in place of equals sign.
- When LIKE is used along with % sign then it will work like a meta character search.
- You can specify more than one conditions using **AND** or **OR** operators.
- A WHERE...LIKE clause can be used along with DELETE or UPDATE SQL command also to specify a condition.

Using LIKE clause at Command Prompt:

This will use SQL SELECT command with WHERE...LIKE clause to fetch selected data from MySQL table tutorials_tbl.

Example:

Following example will return all the records from **tutorials_tbl** table for which author name ends with **jay**:

```
root@host# mysql -u root -p
password; Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> SELECT * from tutorials_tbl
   -> WHERE tutorial_author LIKE '%jay';
```

```
+ -------------+----------------+-----------------+   ----------------    +
| tutorial_id | tutorial_title | tutorial_author | submission_date |
+ -------------+----------------+-----------------+   ----------------    +
|           3 | JAVA Tutorial | Sanjay          | 2007-05-21         |
+ -------------+----------------+-----------------+   ----------------    +

   1 rows in set (0.01 sec)

   mysql>
```

- **Inserting Data into Table:**

To insert data into MySQL table, you would need to use SQL **INSERT INTO** command. You can insert data into MySQL table by using mysql> prompt or by using any script like PHP.

Syntax:

Here is generic SQL syntax of INSERT INTO command to insert data into MySQL table:

```
INSERT INTO table_name ( field1, field2,...fieldN )
                       VALUES
                       ( value1, value2,...valueN );
```

To insert string data types, it is required to keep all the values into double or single quote, for example:-"**value**".

Inserting Data from Command Prompt:

This will use SQL INSERT INTO command to insert data into MySQL table tutorials_tbl.

Example:

Following example will create 3 records into **tutorials_tbl** table:

```
mysql> use TUTORIALS;
Database changed
mysql> INSERT INTO tutorials_tbl
    ->(tutorial_title, tutorial_author, submission_date)
    ->VALUES
    ->("Learn PHP", "John Poul", NOW());
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO tutorials_tbl
    ->(tutorial_title, tutorial_author, submission_date)
    ->VALUES
    ->("Learn MySQL", "Abdul S", NOW());
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO tutorials_tbl
     ->(tutorial_title, tutorial_author, submission_date) VALUES
     ->("JAVA Tutorial", "Sanjay", '2007-05-
 06'); Query OK, 1 row affected (0.01 sec)
```

NOTE: Please note that all the arrow signs (->) are not part of SQL command; they are indicating a new line and they are created automatically by MySQL prompt while pressing enter key without giving a semicolon at the end of each line of the command.
In the above example, we have not provided tutorial_id because at the time of table creation, we had given AUTO_INCREMENT option for this field. So MySQL takes care of inserting these IDs automatically. Here, **NOW()** is a MySQL function, which returns current date and time.

- **UPDATE Query**

There may be a requirement where existing data in a MySQL table needs to be modified. You can do so by using SQL **UPDATE** command. This will modify any field value of any MySQL table. Syntax:

Here is generic SQL syntax of UPDATE command to modify data into MySQL table:

```
UPDATE table_name SET field1=new-value1, field2=new-value2
[WHERE Clause]
```

- You can update one or more field altogether.
- You can specify any condition using WHERE clause.
- You can update values in a single table at a time.

The WHERE clause is very useful when you want to update selected rows in a table.
Updating Data from Command Prompt:
This will use SQL UPDATE command with WHERE clause to update selected data into MySQL table tutorials_tbl.
Example:
Following example will update **tutorial_title** field for a record having tutorial_id as 3.

```
root@host# mysql -u root -p password;
Enter password:*******
mysql> use TUTORIALS;
Database changed
mysql> UPDATE tutorials_tbl
    -> SET tutorial_title='Learning JAVA'
    -> WHERE tutorial_id=3;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

- **DELETE Query**

If you want to delete a record from any MySQL table, then you can use SQL command **DELETE FROM**. You can use this command at mysql> prompt as well as in any script like PHP.
Syntax:
Here is generic SQL syntax of DELETE command to delete data from a MySQL table:

DELETE FROM table_name [WHERE Clause]

- If WHERE clause is not specified, then all the records will be deleted from the given MySQL table.
- You can specify any condition using WHERE clause.
- You can delete records in a single table at a time.

The WHERE clause is very useful when you want to delete selected rows in a table.
Deleting Data from Command Prompt:
This will use SQL DELETE command with WHERE clause to delete selected data into MySQL table tutorials_tbl.

Example:
Following example will delete a record into tutorial_tbl whose tutorial_id is 3.

```
mysql> use TUTORIALS;
Database changed
mysql> DELETE FROM tutorials_tbl WHERE
tutorial_id=3; Query OK, 1 row affected (0.23 sec)
```

CONCLUSION: We have studied and implemented various DML statements in SQL with like clause and operators.

FAQ:
A batch
1. Write SQL Statement for awarding 5 bonus marks to all student following schema are? Student_Branch (roll_no, student_name, branch, address, marks)
2. Which command allow us to add to our database file?
3. How many type of SQL Domain?

B batch
1. Can you tell name of commercial query language?
2. What is procedural and non-procedural DML?
3. Write example of delete query?

C batch
1. What is wild card characters?
2. Define various constraints.
3. Define aggregate and scalar functions.

D batch
1. What are Boolean opeartors?
2. Write example of update query.
3. How many types of relationships exist in database design?

ASSIGNMENT NO.3

TITLE: Execute the aggregate functions like count, sum, avg etc. on the suitable database. Make use of built in functions according to the need of the database chosen. Retrieve the data from the database based on time and date functions like now (), date (), day (), time () etc.Use group by and having clauses.

OBJECTIVE:

1. To understand the aggregate, Date –time functions and group by, having clause.

THEORY:

Aggregate Functions:

COUNT Function

MySQL **COUNT** Function is the simplest function and very useful in counting the number of records, which are expected to be returned by a SELECT statement. To understand **COUNT** function, consider an **employee_tbl** table, which is having the following records:

```
mysql> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

Now, suppose based on the above table you want to count total number of rows in this table, then you can do it as follows:

```
mysql>SELECT COUNT(*) FROM employee_tbl ;
+----------+
| COUNT(*) |
+----------+
|        7 |
+----------+
 row in set (0.01 sec)
```

Similarly, if you want to count the number of records for Zara, then it can be done as follows:

```
mysql>SELECT COUNT(*) FROM employee_tbl
            -> WHERE name="Zara";
+----------+
| COUNT(*)
```

```
                |
+---------+
|       2 |
+---------+
row in set (0.04 sec)
```

MAX Function

MySQL **MAX** function is used to find out the record with maximum value among a record set.
To understand **MAX** function, consider an **employee_tbl** table, which is having the following records:

```
mysql> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
                                            220
|    4 | Jill | 2007-04-06 |                    |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

Now, suppose based on the above table you want to fetch maximum value of daily_typing_pages, then you can do so simply using the following command:

```
mysql> SELECT MAX(daily_typing_pages)
                  -> FROM
            employee_tbl;
+-------------------------+
                          |
 MAX(daily_typing_pages)
                          |
+-------------------------+
|              350 |
+-------------------------+
  row in set (0.00 sec)
```

You can find all the records with maximum value for each name using **GROUP BY** clause as follows:

```
mysql> SELECT id, name, MAX(daily_typing_pages)
   -> FROM employee_tbl GROUP BY name;
+------+------+-------------------------+
| id   | name | MAX(daily_typing_pages) |
+------+------+-------------------------+
|    3 | Jack |                     170 |
|    4 | Jill |                   220 |
|    1 | John |                     250 |
|    2 | Ram  |                     220 |
|    5 | Zara |                   350 |
+------ +------ +------------------------         +
5 rows in set (0.00 sec)
```

MIN Function

MySQL **MIN** function is used to find out the record with minimum value among a record set.
To understand **MIN** function, consider an **employee_tbl** table, which is having the following records:

```
mysql> SELECT * FROM employee_tbl;
+------  +------+------------+-------------------     +
| id   | name | work_date  | daily_typing_pages |
+------  +------+------------+-------------------     +
|    1 | John | 2007-01-24 |           250 |
|    2 | Ram  | 2007-05-27 |           220 |
|    3 | Jack | 2007-05-06 |           170 |
|    3 | Jack | 2007-04-06 |           100 |
                                        220
|    4 | Jill | 2007-04-06 |              |
|    5 | Zara | 2007-06-06 |           300 |
|    5 | Zara | 2007-02-06 |           350 |
+------  +------+------------+-------------------     +
7 rows in set (0.00 sec)
```

Now, suppose based on the above table you want to fetch minimum value of daily_typing_pages, then you can do so simply using the following command:

```
mysql> SELECT MIN(daily_typing_pages)
                    -> FROM
            employee_tbl;
                          +
| MIN(daily_typing_pages)
                                |
                          +
|               100 |
                          +
  row in set (0.00 sec)
```

You can find all the records with minimum value for each name using **GROUP BY** clause as follows:

```
mysql>SELECT id, name, MIN(daily_typing_pages)
             -> FROM employee_tbl GROUP BY name;
```

AVG Function

MySQL **AVG** function is used to find out the average of a field in various records. To understand **AVG** function, consider an **employee_tbl** table, which is having following records:

```
mysql> SELECT * FROM employee_tbl;
+------  +------+------------+-------------------     +
| id   | name | work_date  | daily_typing_pages |
+------  +------+------------+-------------------     +
|    1 | John | 2007-01-24 |           250 |
|    2 | Ram  | 2007-05-27 |           220 |
|    3 | Jack | 2007-05-06 |           170 |
|    3 | Jack | 2007-04-06 |           100 |
|    4 | Jill | 2007-04-06 |           220
|    5 | Zara | 2007-06-06 |           300 |
|    5 | Zara | 2007-02-06 |           350 |
+------   +------+------------+--------------------     +
7 rows in set (0.00 sec)
```

Now, suppose based on the above table you want to calculate average of all the dialy_typing_pages, then you can do so by using the following command:

```
mysql> SELECT AVG(daily_typing_pages)
                 -> FROM
            employee_tbl;
                        +
| AVG(daily_typing_pages) |
                        +
|           230.0000 |
                        +
```

You can take average of various records set using **GROUP BY** clause. Following example will take average all the records related to a single person and you will have average typed pages by every person.

```
mysql> SELECT name, AVG(daily_typing_pages)
          -> FROM employee_tbl GROUP BY
                                        name;
+------+------------------------          +
| name | AVG(daily_typing_pages) |
+------+------------------------          +
| Jack |            135.0000 |
                    220.0000
| Jill |                        |
| John |            250.0000 |
| Ram  |            220.0000 |
| Zara |            325.0000 |
+------+------------------------          +
5 rows in set (0.20 sec)
```

SUM Function

MySQL **SUM** function is used to find out the sum of a field in various records. To understand **SUM** function, consider an **employee_tbl** table, which is having the following records:

```
mysql> SELECT * FROM employee_tbl;
+------   +------+------------+--------------------     +
| id   | name | work_date  | daily_typing_pages |
+------   +------+------------+--------------------     +
|    1 | John | 2007-01-24 |           250 |
|    2 | Ram  | 2007-05-27 |           220 |
|    3 | Jack | 2007-05-06 |           170 |
|    3 | Jack | 2007-04-06 |           100 |
                                       220
|    4 | Jill | 2007-04-06 |              |
|    5 | Zara | 2007-06-06 |           300 |
|    5 | Zara | 2007-02-06 |              350 |
+------   +------ +------------+--------------------     +
7 rows in set (0.00 sec)
```

Now, suppose based on the above table you want to calculate total of all the dialy_typing_pages, then you can do so by using the following command:

```
mysql> SELECT SUM(daily_typing_pages)
                              -> FROM
                     employee_tbl;
                                            +
| SUM(daily_typing_pages) |
                                            +
|                    1610 |
                                            +
   row in set (0.00 sec)
```

You can take sum of various records set using **GROUP BY** clause. Following example will sum up all the records related to a single person and you will have total typed pages by every person.

```
mysql> SELECT name, SUM(daily_typing_pages)
            -> FROM employee_tbl GROUP BY
                                         name;
+-----+------------------------           +
| name | SUM(daily_typing_pages) |
+-----+------------------------           +
| Jack |                   270 |
                           220
| Jill |                       |
| John |                   250 |
| Ram  |                   220 |
| Zara |                   650 |
+-----+------------------------           +
5 rows in set (0.17 sec)
```

GROUP BY Clause

You can use **GROUP BY** to group values from a column, and, if you wish, perform calculations on that column. You can use COUNT, SUM, AVG, etc., functions on the grouped column.
To understand **GROUP BY** clause, consider an **employee_tbl** table, which is having the following records:

```
mysql> SELECT * FROM employee_tbl;
+------   +------+------------+-------------------     +
| id   | name | work_date  | daily_typing_pages |
+------   +------+------------+-------------------     +
|    1 | John | 2007-01-24 |             250 |
|    2 | Ram  | 2007-05-27 |             220 |
|    3 | Jack | 2007-05-06 |             170 |
|    3 | Jack | 2007-04-06 |             100 |
                                          220
|    4 | Jill | 2007-04-06 |                  |
|    5 | Zara | 2007-06-06 |          300 |
|    5 | Zara | 2007-02-06 |          350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

Now, suppose based on the above table we want to count number of days each employee did work.
If we will write a SQL query as follows, then we will get the following result:

```
mysql> SELECT COUNT(*) FROM employee_tbl;
+--------------------------
-                         +
| COUNT(*)                |
+--------------------------
-                         +
| 7                     |
+--------------------------
-                         +
```

But this is not serving our purpose, we want to display total number of pages typed by each person separately. This is done by using aggregate functions in conjunction with a **GROUP BY** clause as follows:

```
mysql> SELECT name, COUNT(*)
    -> FROM    employee_tbl
```

```
| Jill |        1 |
| John |        1 |
| Ram  |        1 |
| Zara |        2 |
+------+----------+
5 rows in set (0.04 sec)
```

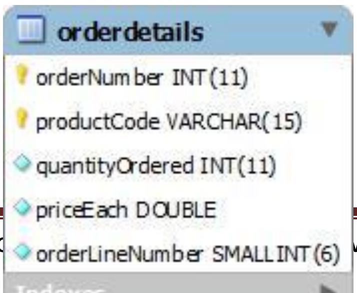
We will see more functionality related to GROUP BY in other functions like SUM, AVG, etc.

HAVING clause

The MySQL HAVING clause is used in the SELECT statement to specify filter conditions for group of rows or aggregates.
The MySQL HAVING clause is often used with the GROUP BY clause. When using with the GROUP BY clause, you can apply a filter condition to the columns that appear in the GROUP BY clause. If the GROUP BY clause is omitted, the MySQL HAVING clause behaves like the WHERE clause. Notice that the MySQL HAVING clause applies the condition to each group of rows, while the WHERE clause applies the condition to each individual row.

Examples of using MySQL HAVING clause

Let's take a look at an example of using MySQL HAVING clause.
We will use the orderdetails table in the sample database for the sake of demonstration.

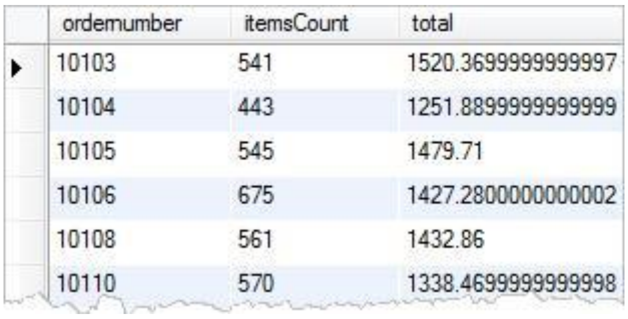


We can use the MySQL GROUP BY clause to get order number, the number of items sold per order and total sales for each:

```
SELECT ordernumber,
       SUM(quantityOrdered) AS itemsCount,
       SUM(priceeach) AS total
FROM orderdetails
GROUP BY ordernumber
```

Now, we can find which order has total sales greater than $1000. We use the MySQL HAVING clause on the aggregate as follows:

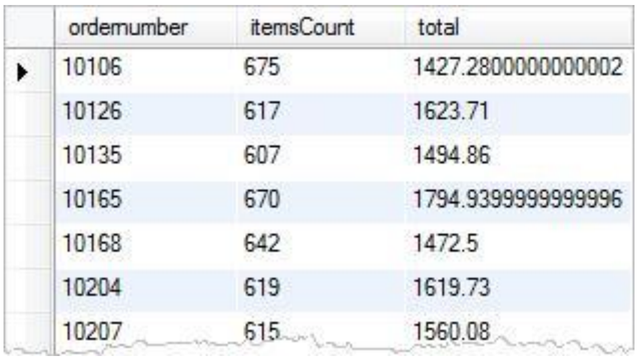
```
SELECT ordernumber,
       SUM(quantityOrdered) AS itemsCount,
       SUM(priceeach) AS total
        FROM orderdetails
       GROUP BY ordernumber
        HAVING total > 1000
```



	ordernumber	itemsCount	total
▶	10103	541	1520.3699999999997
	10104	443	1251.8899999999999
	10105	545	1479.71
	10106	675	1427.2800000000002
	10108	561	1432.86
	10110	570	1338.4699999999998

We can construct a complex condition in the MySQL HAVING clause using logical operators such as OR and AND. Suppose we want to find which order has total sales greater than $1000 and contains more than 600 items, we can use the following query:

```
SELECT ordernumber,
       sum(quantityOrdered) AS
       itemsCount, sum(priceeach) AS total
        FROM orderdetails
        GROUP BY ordernumber
        HAVING total > 1000 AND itemsCount > 600
```



	ordernumber	itemsCount	total
▶	10106	675	1427.2800000000002
	10126	617	1623.71
	10135	607	1494.86
	10165	670	1794.9399999999996
	10168	642	1472.5
	10204	619	1619.73
	10207	615	1560.08

The MySQL HAVING clause is only useful when we use it with the GROUP BY clause to generate the output of the high-level reports. For example, we can use the MySQL HAVING clause to answer some kinds of queries like give me all the orders in this month, this quarter and this year that have total sales greater than 10K.

Date-Time Functions

DATE(expr)

Extracts the date part of the date or datetime expression expr.

```
m ysql> SELECT DATE('2003-12-31 01:02:03');
+---------------------------------------------------------+
| DATE('2003-12-31 01:02:03') |
+---------------------------------------------------------+
| 2003-12-31 |
+---------------------------------------------------------+
```

DAY(date)

DAY() is a synonym for DAYOFMONTH().

NOW ()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. The value is expressed in the current time zone.

```
mysql> SELECT NOW();
+---------------------------------------------------------+
| NOW() |
+---------------------------------------------------------+
| 2017-07-11 23:50:26 |
+---------------------------------------------------------+
```

TIME(expr)

Extracts the time part of the time or datetime expression expr and returns it as a string.

```
m ysql> SELECT TIME('2017-12-31 01:02:03');
+---------------------------------------------------------+
| TIME('2017-12-31 01:02:03') |
+---------------------------------------------------------+
| 01:02:03 |
+---------------------------------------------------------+
```

DAYOFMONTH(date)

Returns the day of the month for date, in the range 0 to 31.

```
m ysql> SELECT DAYOFMONTH('1998-02-03');
+---------------------------------------------------------+
| DAYOFMONTH('1998-02-03') |
+---------------------------------------------------------+
| 3 |
+------------------------------------------------------+
1 row in set (0.00 sec)
```

CONCLUSION: We have studied Aggregate and Date Time Functions and Clauses.

ASSIGNMENT NO.4

TITLE:

Implement nested sub queries. Perform a test for set membership (in, not in), set comparison (<some, >=some, <all etc.) and set cardinality (unique, not unique).

OBJECTIVE:

1. To understand concept of sub-queries

THEORY:

SUBQUERIES:

A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.
A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc. There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.

- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement:

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows:

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE column_name OPERATOR
    (SELECT column_name [, column_name ]
    FROM table1 [, table2 ]
    [WHERE])
```

Subqueries with the INSERT Statement:

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions. The basic syntax is as follows:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
        SELECT [ *|column1 [, column2 ]
        FROM table1 [, table2 ]
        [ WHERE VALUE OPERATOR ]
```

Example:

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy complete CUSTOMERS table into CUSTOMERS_BKP, following is the syntax:

```
SQL> INSERT INTO CUSTOMERS_BKP
   SELECT * FROM CUSTOMERS
   WHERE ID IN (SELECT ID FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows:

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.

Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers whose AGE is greater than or equal to 27: SQL> UPDATE CUSTOMERS

```
   SET SALARY = SALARY * 0.25
   WHERE     AGE     IN     (SELECT     AGE     FROM
            CUSTOMERS_BKP WHERE AGE >= 27 );
```

Subqueries with the DELETE Statement:

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
```

```
[ WHERE OPERATOR [ VALUE ]
   (SELECT COLUMN_NAME
   FROM TABLE_NAME)
   [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table.
Following example deletes records from CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

```
SQL> DELETE FROM CUSTOMERS
    WHERE AGE IN (SELECT AGE FROM
                CUSTOMERS_BKP WHERE AGE > 27 );
```

MySQL Subqueries with ALL, ANY, IN, or SOME

You can use a subquery after a comparison operator, followed by the keyword ALL, ANY, or SOME.

The ALL operator compares value to every value returned by the subquery. Therefore ALL operator (which must follow a comparison operator) returns TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.

Syntax :
operand comparison_operator ALL (subquery)

NOT IN is an alias for <> ALL. Thus, these two statements are the same :

```
SELECT c1 FROM t1 WHERE c1 <> ALL (SELECT c1 FROM t2);
SELECT c1 FROM t1 WHERE c1 NOT IN (SELECT c1 FROM t2);
```

Subqueries with ANY, IN, or SOME
Syntax:

operand comparison_operator ANY (*subquery*)
operand IN (*subquery*)
operand comparison_operator SOME (*subquery*)
Where `comparison_operator` is one of these operators:

`= > < >= <= <> !=`

The `ANY` keyword, which must follow a comparison operator, means “return `TRUE` if the comparison is `TRUE` for `ANY` of the values in the column that the subquery returns.” For example:

```
SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
```

Subqueries with EXISTS or NOT EXISTS
If a subquery returns any rows at all, `EXISTS subquery` is `TRUE`, and `NOT EXISTS subquery` is `FALSE`. For example:

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

Traditionally, an `EXISTS` subquery starts with `SELECT *`, but it could begin with `SELECT 5` or `SELECT column1` or anything at all. MySQL ignores the `SELECT` list in such a subquery, so it makes no difference.

For the preceding example, if `t2` contains any rows, even rows with nothing but `NULL` values, the `EXISTS` condition is `TRUE`. This is actually an unlikely example because a `[NOT] EXISTS` subquery almost always contains correlations. Here are some more realistic examples:

- What kind of store is present in one or more cities?

```
SELECT DISTINCT store_type FROM stores
  WHERE EXISTS (SELECT * FROM cities_stores
              WHERE cities_stores.store_type = stores.store_type);
```

Correlated Subqueries

A *correlated subquery* is a subquery that contains a reference to a table that also appears in the outer query. For example:

```
SELECT * FROM t1
  WHERE column1 = ANY (SELECT column1 FROM t2
                    WHERE t2.column2 = t1.column2);
```

Notice that the subquery contains a reference to a column of t1, even though the subquery's FROM clause does not mention a table t1. So, MySQL looks outside the subquery, and finds t1 in the outer query.

Test for the Absence of Duplicate Tuples:

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct9 returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query "Find all courses that were offered at most once in 2009" as follows:

select *T.course id* **from** *course* **as** *T* **where unique** (**select** *R.course id*
from *section* **as** *R* **where** *T.course id*= *R.course id* **and**
R.year = 2009);

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

CONCLUSION: We have studied Sub queries and their types.

ASSIGNMENT NO.5

TITLE:

Write and execute suitable database triggers .Consider row level and statement level triggers.

OBJECTIVE:

1. To understand the concept of Triggers in PL/SQL .

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: We have studied & implemented triggers in PL/SQL.

ASSIGNMENT NO.6

TITLE:

Write and execute PL/SQL stored procedure and function to perform a suitable task on the database. Demonstrate its use.

OBJECTIVE:

1. To understand the PL/SQL.

THEORY:

PL/SQL is the procedural extension to SQL with design features of programming languages. Data manipulation and query statements of SQL are included within procedural units of code.

Benefits of PL/SQL:

PL/SQL can improve the performance of an application. The benefits differ depending on the execution environment:

- PL/SQL can be used to group SQL statements together within a single block and to send the entire block to the server in a single call thereby reducing networking traffic. Without PL/SQL, the SQL statements are sent to the Oracle server one at a time. Each SQL statement results in another call to the Oracle server and higher performance overhead. In a network environment, the overhead can become significant.
- PL/SQL can also operate with Oracle server application development tools such as Oracle forms and Oracle reports.

Every unit of PL/SQL comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks which can contain any number of nested sub-blocks. Therefore one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Modularized program development:

- Group logically related statements within blocks.
- Nest sub-blocks inside larger blocks to build powerful programs.
- Break down a complex problem into a set of manageable well defined logical modules and implement the modules with blocks.
- Place reusable PL/SQL code in libraries to be shared between Oracle forms and Oracle reports, applications, or store it in a Oracle server to make it accessible to any application that can interact with an Oracle database.

- Declare variables, cursors, constants and exceptions and then use them in SQL and procedural statements.
- Declare variables belonging to scalar, reference, composite and large object (LOB) data types.

Declare variables dynamically based on the data structure of tables and columns in the database.

PL/SQL Block Syntax:

DECLARE [Optional]

  Variables, cursors, user defined exceptions

BEGIN [Mandatory]

- SQL Statements –
  - PL/SQL Statements – Exception [Optional]

- Actions to be performed when errors occur ----

END [Mandatory];

PL/SQL provides two kinds of subprograms:

- **Functions**: these subprograms return a single value, mainly used to compute and return a value.
- **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
    <   procedure_bod
        y > END
        procedure_nam
        e;
```

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION
function_name [(parameter_name [IN | OUT | IN OUT]
type [, ...])] RETURN return_datatype
{IS | AS}
BEGIN
<   function_body >
END [function_name];
```

To write PL/SQL block follow the steps given below:

STEP 1 : Create table :

SQL> **create table** stud(sno number primary key, sname char(15), sub1 number,sub2 number, sub3 number,grade char(15));

Table created.

STEP 2 : Insert the records in table named stud :

The last field of grade is left blank since it would be calculated later using procedure. SQL> insert into stud values(1,'Pray',98,94,90,' '); 1 row created.

STEP 3 : View the records in table named stud :

SQL> select * from stud;

STEP 4 : Create a function named givegrade which checks conditions and returns the grade in variable named g:

STEP 5 : Write the following procedure that will update the grade in the table :

STEP 6 : View the updated table named stud which shows grades of students.

select * from stud;

CONCLUSION: We have implemented PL/SQL block to calculate grade of students.

ASSIGNMENT NO.7

TITLE:
Write a PL/SQL block to implement all types of cursor.

OBJECTIVE:

1. To understand the PL/SQL cursors.

THEORY:

Cursors: Every SQL statement executed by the Oracle server has an individual cursor associated with it and are called implicit cursors. There are two types of cursors. Implicit cursors: Declared for all DML and PL/SQL SELECT statements.
Explicit cursors: Declared and names by the programmer.
Explicit Cursors:

Individually process each row returned by a multiple row select statement. A PL/SQL program opens a cursor, processes rows returned by a query, and then
closes the cursor. The cursor marks the current position in the active set.
Can process beyond the first row returned by the query, row by row.
Keep track of which row is currently being processed.
Allow the programmer to manually control explicit cursors in the PL/QL block.

Declare the cursor by naming it and defining the structure of the query to be performed. Within it.
Open the cursor: The OPEN statement executes the query and binds the variables that are referenced. Rows identified by the query are called the active set and are now available for fetching.
Fetch data from the cursor: After each fetch, you test the cursor for any existing row. If there are no more rows to process, then you must close the cursor.
Close the cursor: The CLOSE statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.
Working with MySQL cursor

First, you have to declare a cursor by using the DECLARE statement:

-
- DECLARE cursor_name CURSOR FOR SELECT_statement;

The cursor declaration must be after any variable declaration. If you declare a cursor before variables declaration, MySQL will issue an error. A cursor must always be associated with a SELECT statement.

- Next, you open the cursor by using the OPEN statement. The OPEN statement initializes the result set for the cursor therefore you must call the OPEN statement before fetching rows from the result set.
-

OPEN cursor_name;
Then, you use the FETCH statement to retrieve the next row pointed by the cursor and move the cursor to the next row in the result set.
FETCH cursor_name INTO variables list;

After that, you can check to see if there is any row available before fetching

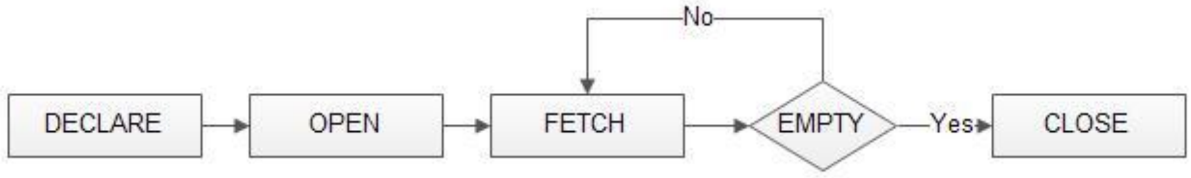
it.
Finally, you call the CLOSE statement to deactivate the cursor and release the memory associated with it as follows:

CLOSE cursor_name;

When the cursor is no longer used, you should close it.
When working with MySQL cursor, you must also declare a NOT FOUND handler to handle the situation when the cursor could not find any row.

Because each time you call the FETCH statement, the cursor attempts to read the next row in the result set. When the cursor reaches the end of the result set, it will not be able to get the data, and a condition is raised. The handler is used to handle this condition.
To declare a NOT FOUND handler, you use the following syntax:
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

The following diagram illustrates how MySQL cursor works.



MySQL Cursor Example

We are going to develop a stored procedure that builds an email list of all employees in the employees table in the company sample database provided with this assignment
○
First, we declare some variables, a cursor for looping over the emails of employees, and a NOT FOUND handler:
○
DECLARE finished INTEGER DEFAULT 0; ○
DECLARE email varchar(255) DEFAULT ""; ○
○ -- declare cursor for employee email ○
DEClARE email_cursor CURSOR FOR ○
SELECT email FROM employees; ○

○ -- declare NOT FOUND handler

DECLARE CONTINUE HANDLER
FOR NOT FOUND SET finished = 1;
○
○ Next, we open the email_cursor by using the OPEN statement: ○ OPEN email_cursor;
○ Then, we iterate the email list, and concatenate all emails where each email is separated by a semicolon(;):
○
○ get_email: LOOP
○ FETCH email_cursor INTO v_email; ○
IF v_finished = 1 THEN
○ LEAVE get_email; ○
END IF;
○-- build email list
○ SET email_list = CONCAT(v_email,";",email_list); ○
END LOOP get_email;

○

After that, inside the loop we used the v_finished variable to check if there is any email in the list to terminate the loop.
Finally, we close the cursor using the CLOSE statement:

○
○ CLOSE email_cursor;
○
○ The build_email_list stored procedure is as follows: ○

```
○ DELIMITER $$
○
○ CREATE PROCEDURE build_email_list (INOUT email_list varchar(4000))
○ BEGIN
○
○DECLARE v_finished INTEGER DEFAULT 0;
○ DECLARE v_email varchar(100) DEFAULT ""; ○

○ -- declare cursor for employee email ○
DEClARE email_cursor CURSOR FOR
○ SELECT email FROM employees; ○

○ -- declare NOT FOUND handler ○
DECLARE CONTINUE HANDLER
○ FOR NOT FOUND SET v_finished = 1; ○

○ OPEN email_cursor; ○

○ get_email: LOOP ○

○FETCH email_cursor INTO v_email;

          IF v_finished = 1 THEN
             LEAVE get_email;
          END IF;

 build email list
  SET email_list = CONCAT(v_email,";",email_list); ○
  END LOOP get_email; ○
  CLOSE email_cursor; ○
  END$$
○ DELIMITER ;
```

○ You can test the build_email_list stored procedure using the following script: ○

```
○   SET @email_list = "";
○ CALL build_email_list(@email_list);
○ SELECT @email_list;
○
```

<u>Syntax:</u>
<u>Declaring a cursor:</u>
CURSOR cursor_name IS
    Select_statement;
<u>Opening a cursor:</u>

OPEN cursor_name;

Fetch data from a cursor:

FETCH cursor_name INTO [variable1, variable2,….]| record_name];

Closing a cursor: Close cursor_name;

Attributes of an Explicit Cursor:

%ISOPEN [is cursor open]

%NOTFOUND [is row not found] ○

%FOUND [is row found]

○ %ROWCOUNT [rows returned so far]

Cursors can be passed parameters. Cursors also have FOR UPDATE option which allows more fine grained control of locking at a table level. WHERE CURRENT OF can be used to apply the update or delete operation to current row in the cursor.

1. Write a Cursor to display the first five records on the following

   table. Student(sno, sname, address, city)

2. Write a Cursor to display the employee number, name, department and salary of first employee getting the highest salary.

Emp (eno, ename, department, address, city)

Salary (eno, salary)

CONCLUSION: We have implemented implicit and explicit cursors.

FAQ:

A batch

1. What is Mysql cursor?
2. Explain the properties of Mysql cursor.
3. How to create cursor in Mysql explain with proper example

ASSIGNMENT NO.8

TITLE:
Execute DDL statements which demonstrate the use of views. Try to update the base table using its corresponding view. Also consider restrictions on updatable views and perform view creation from multiple tables.

OBJECTIVE:

1) To understand the concept of view and implementation

THEORY:

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following −

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| 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 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name     | age |
+----------+-----+
| Ramesh   |  32 |
| Khilan   |  25 |
| kaushik  |  23 |
| Chaitali |  25 |
| Hardik   |  27 |
| Komal    |  22 |
| Muffy    |  24 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM  CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.

- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
   SET AGE = 35
   WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | 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 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW
```

```
   WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | 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 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

```
DROP VIEW view_name;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW CUSTOMERS_VIEW;
```

CONCLUSION: We have implemented views with the help of DDL statements .

Group C: MongoDB

ASSIGNMENT NO.1

TITLE: Create a database with suitable example using MongoDB and implement

- Inserting and saving document (batch insert, insert validation)
- Removing document
- Updating document (document replacement, using modifiers, upserts, updating multiple documents, returning updated documents)

OBJECTIVE:

1. To understand the basic queries used in mongodb.

THEORY:

Connect to a Database

Connect to the database server, which runs as mongod, and begin using the mongo shell to select a logical database within the database instance and access the help text in the mongo shell.

Connect to a mongod

From a system prompt, start mongo by issuing the mongo command, as follows:
mongo
By default, mongo looks for a database server listening on port 27017 on the localhost interface. To connect to a server on a different port or interface, use the --*port* and --*host* options.

Select a Database

After starting the mongo shell your session will use the test database by default. At any time, issue the following operation at the mongo to report the name of the current database: db

1. From the mongo shell, display the list of databases, with the following operation:
2. show dbs
3. Switch to a new database named mydb, with the following operation:
4. use mydb
5. Confirm that your session has the mydb database as context, by checking the value of the db object, which returns the name of the current database, as follows:
6. db

At this point, if you issue the show dbs operation again, it will not include the mydb database. MongoDB will not permanently create a database until you insert data into that database. The *Create a Collection and Insert Documents* section describes the process for inserting data. New in version 2.4: show databases also returns a list of databases.

Display mongo Help

At any point, you can access help for the mongo shell using the following operation:
help
Furthermore, you can append the .help() method to some JavaScript methods, any cursor object, as well as the db and db.collection objects to return additional help information. Create a Collection and Insert Documents
In this section, you insert documents into a new *collection* named testData within the new *database* named mydb.

MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses *dynamic schemas,* you also need not specify the structure of your documents before inserting them into the collection.

1. From the mongo shell, confirm you are in the mydb database by issuing the following:
2. db
3. If mongo does not return mydb for the previous operation, set the context to the mydb database, with the following operation:
4. use mydb
5. Create two documents named j and k by using the following sequence of JavaScript operations:
6. j = { name : "mongo" }
7. k = { x : 3 }
8. Insert the j and k documents into the testData collection with the following sequence of operations:
9. db.testData.insert(j)
10. db.testData.insert(k)

When you insert the first document, the mongod will create both the mydb database and the testData collection.

11. Confirm that the testData collection exists. Issue the following operation:
12. show collections

The mongo shell will return the list of the collections in the current (i.e. mydb) database. At this point, the only collection is testData. All mongod databases also have a system.indexes collection.

13. Confirm that the documents exist in the testData collection by issuing a query on the collection using the find() method:
14. db.testData.find()

This operation returns the following results. The *ObjectId* values will be unique:

```
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
```

All MongoDB documents must have an _id field with a unique value. These operations do not explicitly specify a value for the _id field, so mongo creates a unique *ObjectId* value for the field before inserting it into the collection.

In MongoDB, the db.collection.find() method retrieves documents from a collection.

The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax:

Basic syntax of **createCollection()** command is as follows

db.createCollection(name, options)

The use Command

MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database, if it doesn't exist otherwise it will return the existing database.

Syntax:

Basic syntax of **use DATABASE** statement is as follows:

```
use DATABASE_NAME
```

Example:

If you want to create a database with name **<mydb>**, then **use DATABASE** statement would be as follows:

```
>use mydb
switched to db mydb
```

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

Syntax
Basic syntax of insert () **command is as follows:**
>db.COLLECTION_NAME.insert(document)
MongoDB Update () method
The update () method updates values in the existing document.
Syntax:
Basic syntax of **update()** method is as follows
>db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Batch Insert:

- db.car.insert(

```
... [
... { _id:1,name:"Audi",color:"Red",cno:"H101",mfdcountry:"Germany",speed:75 },
... { _id:2,name:"Swift",color:"Black",cno:"H102", speed:60 },
 ... {name:"Maruthi800",color:"Blue",cno:"H103",mfdcountry:"India",speed:70 },
...  ])
```

- db.car.insert(

```
...{ _id:2,name:"EEco",color:"White", speed:75 },
...)
```

Duplicate ID error

Validation of data:
- db.collection.validate(full)
  Validates a collection. The method scans a collection's data structures for correctness.
- MongoDB provides the capability to validate documents during updates and insertions. Validation rules are specified on a per-collection basis using the validator option, which takes a document that specifies the validation rules or expressions.
- You can specify document validation rules when creating a new collection using db.createCollection () with the validator option. When you add validation to a collection, existing documents do not undergo validation checks until modification

Add document validation to an existing collection:

- db.runCommand({ collMod: "contacts", validator: { $or: [{ phone: { $exists: true } }, { email: { $exists: true } }] }, validationLevel: "moderate" })
  { "_id": "125876", "name": "Anne", "phone": "+1 555 123 456", "city": "London", "status": "Complete" },{ "_id": "860000", "name": "Ivan", "city": "Vancouver" }
- The contacts collection now has a validator with the moderate validationLevel. If you attempted to update the document with _id of 125876, MongoDB would apply validation rules since the existing document matches the criteria. In contrast, MongoDB will not apply validation rules to updates to the document with _id of 860000 as it does not meet the validation rules.

Update:

- Upsert: To insert rows into a database table if they do not already exist, or update them if they do.
- db.collection.save**()**

Updates an existing document or inserts a new document, depending on its document parameter. If the document does **not** contain an _id field, then the save() method calls the insert() method. During the operation, the mongo shell will create an ObjectId and assign it to the _id field. If the document contains an _id field, then the save() method is equivalent to an update with the upsert option set to true and the query predicate on the _id field.

Examples:

```
db.products.save( { item: "book", qty: 40 } )
{ "_id" : ObjectId("50691737d386d8fadbd6b01d"), "item" : "book", "qty" : 40 }
db.products.save( { _id: 100, item: "water", qty: 30 } )
{ "_id" : 100, "item" : "water", "qty" : 30 }
db.products.save( { _id : 100, item : "juice" } )
```

Because the _id field holds a value that exists in the collection, the operation performs an update to replace the document and results in the following document:

```
{ "_id" : 100, "item" : "juice" }
```

Returning updated document:

- db.collection.findAndModify(

```
{ query: <document>,
remove: <boolean>,
update: <document>,
new: <boolean>,
upsert: <boolean>,
bypassDocumentValidation: <boolean>);
```

remove () Method:

MongoDB's **remove ()** method is used to remove document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag

1. **deletion criteria:** (Optional) deletion criteria according to documents will be removed.
2. **justOne :** (Optional) if set to true or 1, then remove only one document.

Syntax:

Basic syntax of **remove ()** method is as follows

```
> db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

CONCLUSION: We have studied basic CRUD operations in mongodb

FAQ:

What platforms does MongoDB support?

Does MongoDB support SQL?

Does MongoDB support transactions?

ASSIGNMENT NO.2

TITLE: Execute at least 10 queries on any suitable MongoDB database that demonstrates following querying techniques

- find and findOne (specific values)
- Query criteria (Query conditionals, OR queries, $not, Conditional semantics)
- Type-specific queries (Null, Regular expression, Querying arrays)

OBJECTIVE:

1. To understand the querying technique used in mongodb.

THEORY:

find and findOne (specific values): The find method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection, from no documents at all to the entire collection. Which documents get returned is determined by the first argument to find, which is a document specifying the query to be performed.
An empty query document (i.e., {}) matches everything in the collection. If find isn't given a query document, it defaults to {}.
For example, the following:
>1)db.c.find()
returns everything in the collection *c*.
2)where the value for "age" is 27, we can add that key/value pair to the query document:
> db.users.find({"age" : 27})
3) to get all users who are 27-year-olds with the username "joe," we can query for the following:
> db.users.find({"username" : "joe", "age" : 27})
Specifying Which Keys to Return: Sometimes, you do not need all of the key/value pairs in a document returned. If this is the case, you can pass a second argument to find (or findOne) specifying the keys you want. This reduces both the amount of data sent over the wire and the time and memory used to decode documents on the client side.
For example, if you have a user collection and you are interested only in the "user name" and "email" keys, you could return just those keys with the following query:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
"_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
"username" : "joe",
"email" : joe@example.com
}
```

Query Criteria:

1) Query Conditionals:
"$lt", "$lte", "$gt", and "$gte" are all comparison operators, corresponding to <, <=, >, and >=, respectively. They can be combined to look for a range of values.

For example,
1) to look for users who are between the ages of 18 and 30 inclusive, we can do this:
> db.users.find ({"age" : {"$gte" : 18, "$lte" : 30}})
2) to find people who registered before January 1, 2017, we can do this:
> start = new Date("01/01/2017")
> db.users.find({"registered" : {"$lt" : start}})
3) If you want to find all users who do not have the username "joe," you can query for them using this:
> db.users.find ({"username" : {"$ne": "joe"}})
"$ne" can be used with any type.

OR Queries:
There are two ways to do an OR query in MongoDB. "$in" can be used to query for a variety of values for a single key. "$or" is more general; it can be used to query for any of the given values across multiple keys.

For instance,
1)suppose we were running a raffle and the winning
ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

2)In the raffle case, using
"$or" would look like this:

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

$not:
"$not" is a metaconditional: it can be applied on top of any other criteria. As an example, let's consider the modulus operator, "$mod". "$mod" queries for keys whose values, when divided by the first value given, have a remainder of the second value:

```
> db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

The previous query returns users with "id_num"s of 1, 6, 11, 16, and so on. If we want, instead, to return users with "id_num"s of 2, 3, 4, 5, 7, 8, 9, 10, 12, and so on, we can use "$not":

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

Type-Specific Queries:
null:
null behaves a bit strangely. It does match itself, so if we have a collection with the following documents:

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

we can query for documents whose "y" key is null in the expected way:

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

Regular Expressions:
Regular expressions are useful for flexible string matching.

```
db.users.find({"name": /m/}) Hema, Madhu,  Jamila
db.users.find({name: /^pa/})  Starts with pa
db.users.find({name: /ro$/})  Ends with ro
db.collection.find({name:{'$regex' : 'string$', '$options' : 'i'}})
```

Querying Arrays:
Querying for elements of an array is simple. An array can mostly be treated as though each element is the value of the overall key.
Example:

```
db.bios.find( { contribs: "UNIX" } )
"contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ]
```

$all
If you need to match arrays by more than one element, you can use "$all".

$size
A useful conditional for querying arrays is "$size", which allows you to query for arrays of a given size.

The $slice operator
The special "$slice" operator can be used to return a subset of elements for an array key.

suppose we had a blog post document and we wanted to return the first 10 comments:

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

CONCLUSION: We have studied querying technique in mongodb

ASSIGNMENT NO.3

TITLE: Execute at least 10 queries on any suitable MongoDB database that demonstrates following:

- $ where queries
- Cursors (Limits, skips, sorts, advanced query options)
- Database commands

OBJECTIVE:

1. To understand the advanced querying technique used in mongodb.

THEORY:

$where Queries:

Key/value pairs are a fairly expressive way to query, but there are some queries that they cannot represent. For queries that cannot be done any other way, there are "$where" clauses, which allow you to execute arbitrary JavaScript as part of your query. This allows you to do (almost) anything within a query.

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

In the second document, "spinach" and "watermelon" have the same value, so we'd like that document returned. It's unlikely MongoDB will ever have a $ conditional for this, so we can use a "$where" clause to do it with JavaScript:

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
... for (var other in this) {
... if (current != other && this[current] == this[other]) {
... return true;
... }
... }
... }
... return false;
... }});
```

If the function returns true, the document will be part of the result set; if it returns false, it won't be.

We used a function earlier, but you can also use strings to specify a "$where" query; the following two "$where" queries are equivalent:

```
> db.foo.find({"$where" : "this.x + this.y == 10"})
> db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})
```

Cursors:

The database returns results from find using a *cursor*. The client-side implementations of cursors generally allow you to control a great deal about the eventual output of a query.

we create a very simple collection and query it, storing the results in the cursor variable:

```
> for(i=0; i<100; i++) {
... db.c.insert({x : i});
... }
> var cursor = db.collection.find();
```

Limits, Skips, and Sorts:
The most common query options are limiting the number of results returned, skipping a number of results, and sorting.

Example:
To set a limit, chain the limit function onto your call to find. For example, to only return three results, use this:

```
> db.c.find().limit(3)
```

skip works similarly to limit:

```
> db.c.find().skip(3)
```

For instance, to sort the
results by "username" ascending and "age" descending, we do the following:

```
> db.c.find().sort({username : 1, age : -1})
```

- Combine Cursor Methods

```
>db.bios.find().sort( { name: 1 } ).limit( 5 )
>db.products.find().max( {type: 'jonagold' } )
   { "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
   { "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
   { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
   { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
```

Advanced Query Options:

There are two types of queries: *wrapped* and *plain*. A plain query is something like this:

```
> var cursor = db.foo.find({"foo" : "bar"})
```

There are a couple options that “wrap” the query. For example, suppose we perform a sort:

```
> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})
```

Instead of sending {"foo" : "bar"} to the database as the query, the query gets wrapped in a larger document. The shell converts the query from {"foo" : "bar"} to {"$query" : {"foo" : "bar"}, "$orderby" : {"x" : 1}}.

CONCLUSION: We have studied advanced querying technique in mongodb

ASSIGNMENT NO.4

TITLE: IMPLEMENT MAP REDUCES OPERATION WITH SUITABLE EXAMPLE USING MONGODB

OBJECTIVE:

1. To understand the what is map reduce function in mongodb.

THEORY:

Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results. For map-reduce operations, MongoDB provides the mapReduce database command. MongoDB applies the *map* phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the mongod process. Map-reduce operations take the documents of a single *collection* as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage. mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

Example of map reduce function: Return the Total Price Per Customer

Perform the map-reduce operation on the orders collection to group by the cust_id, and calculate the sum of the price for each cust_id:

1. Define the map function to process each input document:
   In the function, this refers to the document that the map-reduce operation is processing.
   - The function maps the price to the cust_id for each document and emits the cust_id and price pair.
2. var mapFunction1 = function() {
3. emit(this.cust_id, this.price);
4. };
5. Define the corresponding reduce function with two arguments keyCustId and valuesPrices:
   - The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId.
   - The function reduces the valuesPrice array to the sum of its elements.
6. var reduceFunction1 = function(keyCustId, valuesPrices) {
7. return Array.sum(valuesPrices);
8. };
9. Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.
10. db.orders.mapReduce(
11. mapFunction1,
12. reduceFunction1,
13. { out: "map_reduce_example" }
14.)

This operation outputs the results to a collection named map_reduce_example. If the map_reduce_example collection already exists, the operation will replace the contents with the results of this map-reduce operation

CONCLUSION: We have implemented map reduce operation using mongodb

ASSIGNMENT NO.5

TITLE:

IMPLEMENT AGGREGATION AND INDEXING WITH SUITABLE EXAMPLE USING MONGODB

OBJECTIVE:

1. To understand the aggregation operation and indexing.

THEORY: Aggregation:

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

The aggregate() Method

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

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

Expression Description	Example
$sum: Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg: Calculates the average of all given values from all documents in the collection	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$min: Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max: Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : : "$likes"}}}]) {$max
$push: Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])

$addToSet: Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first: Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])
$last: Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "$sort"-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Possible stages in aggregation framework are following:

- **$project:** Used to select some specific fields from a collection.
- **$match:** This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **$group:** This does the actual aggregation as discussed above.
- **$sort:** Sorts the documents.
- **$skip:** With this it is possible to skip forward in the list of documents for a given amount of documents.
- **$limit:** This limits the amount of documents to look at by the given number starting from the current position.s
- $unwind: This is used to unwind document that are using arrays. when using an array the data is kind of pre-joinded and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.

Indexing

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

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

Create an Index on a Single Field:

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

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

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

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

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

```
db.records.find( { userid: 2 } )
db.records.find( { userid: { $gt: 10 } } )
```

But the created index does not support the following query on the profile_url field:

```
db.records.find( { profile_url: 2 } )
```

For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Create a unique index.
Use the ensureIndex() method create a unique index.

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

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

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

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

CONCLUSION: We have implemented aggregation using various aggregation operations and indexing.

FAQ:
A batch
1. What is aggregation? Write aggregate() method with proper example.
2. Write a mongo query to calculate average marks of a student in 5 subjects
3. Write use of “project” operator with proper example.

B batch
1. What is indexing? How to create a unique index?
2. Write a mongo query to find minimum and maximum salary of a employee from employee collection.
3. Write query to drop the index.

C batch
1. Explain “match” and “group” operator with proper example.
2. How to create an index on a single field explain with suitable example.
3. Explain “push” operator with example.

D batch
1. What is aggregation pipeline?
2. Write the query which will count all the documents in the collection.
3. How to use project operator to create new field?