Dr. D. Y. Patil Institute of Technology
Pimpri Pune-411018

Department of Artificial Intelligence and Data Science

Laboratory Manual

Savitribai Phule Pune University

Third Year of Artificial Intelligence and Data Science (2020 Course)

Subject Code: 317523

Software Laboratory 1

Prepared by

1. **Ms. Sonam Singh**
2. **Ms. Ritu Dudhmal**
3. **Ms. Seetu Patel**

Academic Year

2024-2025

Dr. D. Y. Patil Institute of Technology
Pimpri Pune-411018

Department of Artificial Intelligence and Data Science

Vision of the Institute

- Empowerment through knowledge

Mission of the Institute

- Developing human potential to serve the Nation
- Dedicated efforts for quality education.
- Yearning to promote research and development.
- Persistent endeavor to imbibe moral and professional ethics.
- Inculcating the concept of emotional intelligence.
- Emphasizing extension work to reach out to the society.
- Treading the path to meet the future challenges.

Vision of the Department

- To produce globally competent engineers in the field of Artificial Intelligence and Data Science with human values

Mission of the Department

- To develop students with a sound understanding in the area of Artificial Intelligence, Machine Learning and Data Science.
- To enable students to become innovators, researchers, entrepreneurs and leaders globally.
- Equip the department with new advancement in high performance equipments and software to carrying out research in emerging technologies in AI and DS.
- To meet the pressing demands of the nation in the areas of Artificial Intelligence and Data Science**.**

DBMS Lab

Subject Code: 317523

Teaching Scheme	Credit	Examination Scheme
PR: 02 Hours/Week	02	TW: 25 Marks PR: 25 Marks

Guidelines for Instructor's Manual

The instructor‘s manual is to be developed as a reference and hands-on resource. It should include prologue (about University/program/ institute/ department/foreword/ preface), curriculum of the course, conduction and Assessment guidelines, topics under consideration, concept, objectives, outcomes, set of typical applications/assignments/ guidelines, and references

Guidelines for Student Journal

The laboratory assignments are to be submitted by student in the form of journal. Journal consists of Certificate, table of contents, and handwritten write-up of each assignment (Title, Date of Completion, Objectives, Problem Statement, Software and Hardware requirements, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, test cases, Test Data Set(if applicable), mathematical model (if applicable), conclusion/analysis. Program codes with sample output of all performed assignments are to be submitted as softcopy. As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal must be avoided. Use of DVD containing students programs maintained by Laboratory In-charge is highly encouraged. For reference one or two journals may be maintained with program prints in the Laboratory.

Guidelines for Laboratory /Term Work Assessment

Continuous assessment of laboratory work should be based on overall performance of Laboratory assignments by a student. Each Laboratory assignment assessment will assign grade/marks based on parameters, such as timely completion, performance, innovation, efficient codes, and punctuality.

Guidelines for Practical Examination

Problem statements must be decided jointly by the internal examiner and external examiner. During practical assessment, maximum weightage should be given to satisfactory implementation of the problem statement. Relevant questions may be asked at the time of evaluation to test the student’s understanding of the fundamentals, effective and efficient implementation. This will encourage, transparent evaluation and fair approach, and hence will not create any uncertainty or doubt in the minds of the students. So, adhering to these principles will consummate our team efforts to the promising start of student's academics.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. Use of open source software is encouraged. Based on the concepts learned. Instructor may also set one assignment or mini-project that is suitable to AI & DS branch beyond the scope of the syllabus.

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

Pract .No.	**Laboratory Assignments**
1.	**SQL Queries:** • Design and Develop SQL DDL statements which demonstrate the use of SQL objects such as Table, View, Design and Develop SQL DDL statements which demonstrate the use of SQL Index, Sequence, Synonym, different constraints etc. • Write at least 10 SQL queries on the suitable database application using SQL DML statements.
2.	**SQL Queries – all types of Join, Sub-Query and View:** Write at least10 SQL queries for suitable database application using SQL DML statements. Note: Instructorwill design the queries which demonstrate the use of concepts like all types of Join ,Sub-Query
3.	**MongoDB Queries:** Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method,logical operators etc.).
4.	**Unnamed PL/SQLcode block: Use of Control structure and Exception handling is mandatory.** Suggested Problem statement: Consider Tables: 1. Borrower (Roll_no, Name, Date_of_Issue, Name_of_Book, Status) 2. Fine (Roll_no, Date, Amt) • Accept Roll_no and Name_of_Book from user. • Check the number of days (from Date_of_Issue). • If days are between 15 to 30 then fine amount will be Rs 5per day. • If no. of days>30, per day fine will be Rs 50 per day and for days less than 30, Rs. 5 per day. • After submitting the book, status will change from I to R. • If condition of fine is true, then details will be stored into fine table. **OR** Also handles the exception by named exception handler or user define exception handler. • MongoDB – Aggregation and Indexing: Design and Develop MongoDB Queries using aggregation andindexing with suitable example using MongoDB. • MongoDB – Map-reduce operations: Implement Map-reduce operation with suitable example usingMongoDB.
5.	**Cursors: (All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor)** Write a PL/SQL block of code using parameterized Cursor that will merge the data available in the newly created table N_Roll_Call with the data available in the table O_Roll_Call. If the data in the first table alreadyexists in the second table then that data should be skipped. Note: Instructor will frame the problem statement for writing PL/SQL block using all types of Cursors in linewith above statement**.**
6.	**Database Connectivity:** Write a program to implement MySQL/Oracle database connectivity with any front end language toimplement Database navigation operations (add, delete, edit etc.)
7.	Develop an application with following details: 1. Follow the same problem statement decided in Assignment-1 of Group A. **2.** Follow the Software Development Life cycle and other concepts learnt in **Software Engineering Course** throughout the implementation. 3. Develop application considering: • Front End: Python/Java/PHP/Perl/Ruby/.NET/ or any other language • Backend : MongoDB/ MySQL/ Oracle / or any standard SQL / NoSQL database 4. Test and validate application using Manual/Automation testing. 5. Student should develop application in group of 2-3 students and submit the Project Report which willconsist of documentation related to different phases of Software Development Life Cycle: • Title of the Project, Abstract, Introduction • Software Requirement Specification (SRS) • Conceptual Design using ER features, Relational Model in appropriate Normalize form • Graphical User Interface, Source Code • Testing document • Conclusion.

Assignment No.1 A

Title of Assignment: Design and Develop SQL/NoSQL DDL statements which demonstrate the use ofSQL objects such as Table, View, Index, Sequence, Synonym.

Objective: to learn and understand database creation, relation / table creation, setting index on table, sequences using MySQL DDL statements.

Relevant Theory
SQL Data Definition:
The set of relations in a database must be specified to the system by means of a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:
The schema for each relation.
The types of values associated with each attribute.
The integrity constraints.
The set of indices to be maintained for each relation.
The security and authorization information for each relation.
The physical storage structure of each relation on disk.

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.
NOTE: All the database names, table names, table fields name are case sensitive. So you would have to use proper names while giving any SQL command.

Creating Table/Relation:

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_namecolumn_type);
```

Now, we will create following table in TUTORIALS database.

```
tutorials_tbl(
tutorial_id INT NOT NULL AUTO_INCREMENT,
tutorial_title VARCHAR(100) NOT NULL,
tutorial_author VARCHAR(40) NOT NULL,
submission_date DATE,
  PRIMARY KEY (tutorial_id)
);
```

Here 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 tablename;
```

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>
```

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)
mysql>
```

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.
Displaying Data from Table:
The SQL SELECT command is used to fetch data from MySQL database. You can use this command at mysql> prompt as well as in any script like PHP.
Syntax:
Here is generic SQL syntax of SELECT command to fetch data from MySQL table:

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE Clause]
[OFFSET M ][LIMIT N]
```

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 fetch one or more fields in a single SELECT command.
You can specify star (*) in place of fields. In this case, SELECT will return all the fields.
You can specify any condition using WHERE clause.
You can specify an offset using OFFSET from where SELECT will start returning records. By default offset is zero.
You can limit the number of returns using LIMIT attribute.

Fetching Data from Command Prompt:
This will use SQL SELECT command to fetch data from MySQL table tutorials_tbl
Example:
Following example will return all the records from tutorials_tbl table:

```
mysql>use TUTORIALS;
Database changed
mysql> SELECT *from tutorials_tbl
+...............+..................+...................+....................+
|tutorial_id|tutorial_title|tutorial_author|submission_date|
+...............+..................+...................+....................+
|1|Learn PHP      |JohnPoul|2007-05-21|
|2|LearnMySQL|Abdul S         |2007-05-21|
|3| JAVA Tutorial|Sanjay|2007-05-21|
+...............+..................+...................+....................+
3 rows inset(0.01 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 inset(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 icolumn 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 inset(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 specifies 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. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.
Example:
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 cannot 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).
Note: You would use these concepts while creating database tables.
Create Primary Key:
Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

```
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)
);
```

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(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       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, followingis 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;
```

MySQL Sequence:
A sequence is a set of integers 1, 2, 3, ..that are generated in order on demand. Sequences are frequently used in databases because many applications require each row in a table to contain a unique value andsequences provide an easy way to generate them. This chapter describes how to use sequences in MySQL. Using AUTO_INCREMENT column:
The simplest way in MySQL to use Sequences is to define a column as AUTO_INCREMENT and leaverest of the things to MySQL to take care.
Example:
Try out the following example. This will create table and after that it will insert few rows in this tablewhere it is not required to give record ID because it's auto incremented by MySQL.

```
mysql> CREATE TABLE insect
->(
->id INT UNSIGNED NOT NULL AUTO_INCREMENT,
-> PRIMARY KEY (id),
->name VARCHAR(30) NOT NULL,# type of insect
->dateDATE NOT NULL,# date collected
->origin VARCHAR(30) NOT NULL # where collected
);
Query OK,0 rows affected (0.02 sec)

mysql> INSERT INTO insect (id,name,date,origin) VALUES
->(NULL,'housefly','2001-09-10','kitchen'),
->(NULL,'millipede','2001-09-10','driveway'),
->(NULL,'grasshopper','2001-09-10','front yard');
Query OK,3 rows affected (0.02 sec)
Records:3Duplicates:0Warnings:0
mysql> SELECT * FROM insect ORDER BY id;
+.....+...............+.............+.............+
|id| name        | date       | origin     |
+.....+...............+.............+.............+
|1| housefly    |2001-09-10| kitchen    |
|2| millipede  |2001-09-10| driveway   |
|3| grasshopper |2001-09-10| front yard |
+.....+...............+.............+.............+
3 rows inset(0.00 sec)
```

Renumbering an Existing Sequence:
There may be a case when you have deleted many records from a table and you want to resequence all the records. This can be done by using a simple trick but you should be very careful to do so if your table is having joins with other table.
If you determine that resequencing an AUTO_INCREMENT column is unavoidable, the way to do it is to drop the column from the table, then add it again. The following example shows how to renumber the id values in the insect table using this technique:

```
mysql> ALTER TABLE insect DROP id;
mysql> ALTER TABLE insect
> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
-> ADD PRIMARY KEY (id);
```

Starting a Sequence at a Particular Value:
By default, MySQL will start sequence from 1 but you can specify any other number as well at the time of table creation. Following is the example where MySQL will start sequence from 100.

```
mysql> CREATE TABLE insect
->(
->id INT UNSIGNED NOT NULL AUTO_INCREMENT =100,
-> PRIMARY KEY (id),
->name VARCHAR(30) NOT NULL,# type of insect
->dateDATE NOT NULL,# date collected
->origin VARCHAR(30) NOT NULL # where collected
);
```

Alternatively, you can create the table and then set the initial sequence value with ALTER TABLE.

```
mysql> ALTER TABLE t AUTO_INCREMENT =100;
```

Creating Views
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 depend on the written SQL query to create a view.
Views, which are kind 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 such 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 very similar way as you use them in 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|
+----+----------+-----+-----------+----------+
```

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

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

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

```
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 is 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.

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 as given below:

```
DROP VIEW view_name;
```

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

```
DROP VIEW CUSTOMERS_VIEW;
```

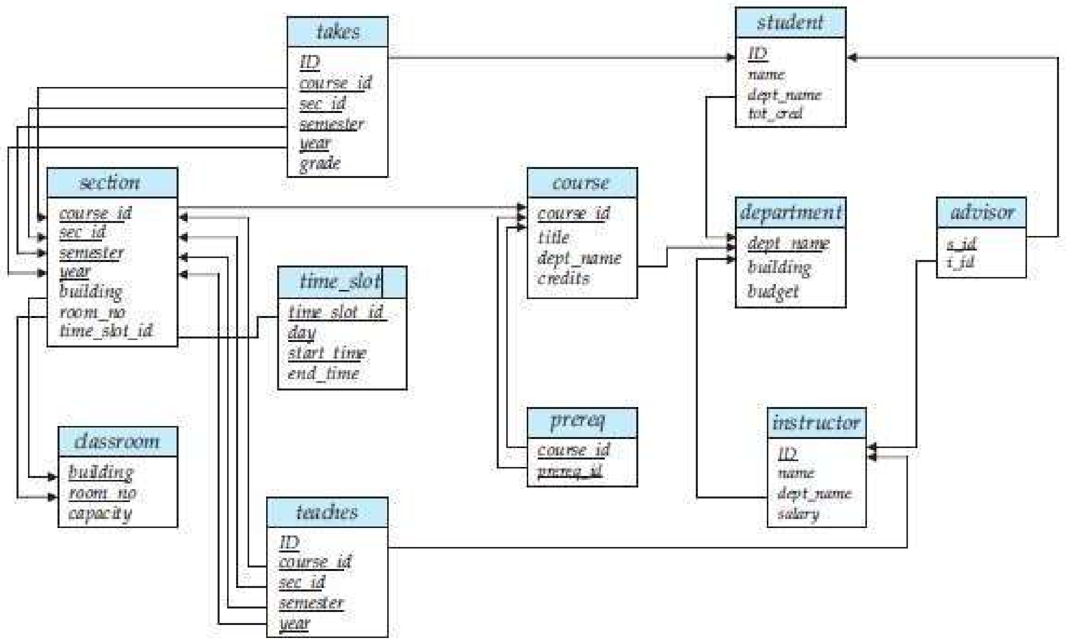
Exercises:

Use FollowingLibrary Database Schema

BOOK (Book_ISBN [PK], Title[Not Null], Publisher_ Name [FK],price[Check Price>0], Date_Of_Publication,Book_Copy),

1. Design and Develop SQL DDL statements for above database using MySQL Client- MySQL Server Data sever.
Create and use library database using Mysql Client
Create tables for above database with all constraints and Insert at least five records in each table.
Create a view on Book table as "DBMS BOOKS" by selecting all books titled DBMS.
Drop the view "DBMS BOOKS" just created
Create an index on Book_Author on table on attribute "Author_Name".

2. Design and Develop SQL DDL statements with all constraints for below university database Schema using MySQL Client- MySQL Server Data sever.



Conclusion:
Thus we have studied and understood SQL DDL statements with all database constrains successfully.

Question:

Consider the following table structure for this assignment:
Table Name: **CUSTOMER**
Fields:

Cust_novarchar(10) Primary Key, FnameVarchar(20) Not NULL, MnameVarchar(20), LnameVarchar(20) Not NULL, Address Varchar(30), City Varchar(15),State Varchar(20), Mobile_no Number(15) Not NULL, Occupation Varchar(20), Company_NameVarchar(25).

Perform the following command/operation on the above table:
1) Create table and Describe that Table
2) Insert values (Minimum 10 rows)
3) Select

Assignment No.1 b

Title of Assignment: Design at least 10 SQL/NoSQL queries for suitable database application using SQL/NoSQL DML statements: Insert, Select, Update, Delete with operators, functions, and set operator.

Objective :to learn and understand database operations such as Insert, Select, Update, and Delete using MySQL DML statements.

Relevant Theory :

Data-Definition Language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

Data-Manipulation Language (DML): The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.

Data Manipulation Language (DML)

A data manipulation language (DML) is a family of syntax elements similar to a computer programming language used for selecting, inserting, deleting and updating data in a database. Performing read-only queries of data is sometimes also considered a component of DML.
Data manipulation language comprises the SQL data change statements,[2] which modify stored data but not the schema or database objects.
Data manipulation languages have their functional capability organized by the initial word in a statement, which is almost always a verb. In the case of SQL, these verbs are:
SELECT ... FROM ... WHERE ...
INSERT INTO ... VALUES ...
UPDATE ... SET ... WHERE ...
DELETE FROM ... WHERE ...
The purely read-only SELECT query statement is classed with the 'SQL-data' statements and so is considered by the standard to be outside of DML. The SELECT ... INTO form is considered to be DML because it manipulates (i.e. modifies) data. In common practice though, this distinction is not made and SELECT is widely considered to be part of DML.
Most SQL database implementations extend their SQL capabilities by providing imperative, i.e. procedural languages.

- ### 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)
```

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.

- **Fetching Data from Table:**

The SQL SELECT command is used to fetch data from MySQL database. You can use this command at mysql> prompt as well as in any script like PHP.
Syntax:
Here is generic SQL syntax of SELECT command to fetch data from MySQL table:

```
SELECT field1, field2,...fieldN table_name1, table_name2...
[WHERE Clause]
[OFFSET M ][LIMIT N]
```

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 fetch one or more fields in a single SELECT command.
You can specify star (*) in place of fields. In this case, SELECT will return all the fields.
You can specify any condition using WHERE clause.
You can specify an offset using OFFSET from where SELECT will start returning records. By default offset is zero.
You can limit the number of returns using LIMIT attribute.

Fetching Data from Command Prompt:

This will use SQL SELECT command to fetch data from MySQL table tutorials_tbl
Example:
Following example will return all the records from tutorials_tbl table:

```
mysql>use TUTORIALS;
Database changed
mysql> SELECT *from tutorials_tbl
+-------------+----------------+-----------------+-----------------+
|tutorial_id|tutorial_title|tutorial_author|submission_date|
+-------------+----------------+-----------------+-----------------+
|1|Learn PHP      |JohnPoul|2007-05-21|
|2|LearnMySQL|Abdul S        |2007-05-21|
|3| JAVA Tutorial|Sanjay|2007-05-21|
+-------------+----------------+-----------------+-----------------+
3 rows inset(0.01 sec)

mysql>
```

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 than 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 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.

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';Emptyset(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 UNIXwhile 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 MySQLtable:

```
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 MySQLtable 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 inset(0.01 sec)
```

```
mysql>
```

GROUP BY Clause

You can use GROUP BY to group values from a column, and, if you wish, perform calculations on thatcolumn. 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|
|4|Jill|2007-04-06|220|
|5|Zara|2007-06-06|300|
|5|Zara|2007-02-06|350|
+.........+.........+..................+.............................+7 rows inset(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
->GROUP BY name;
+........+...............+
|name| COUNT(*)|
+........+...............+
|Jack|2|

|Jill|1|
|John|1|
|Ram|1|
|Zara|2|
+........+...............+
5 rows inset(0.04 sec)
```

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 inset(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|
+...........+
1 row inset(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|
+.............+
1 row inset(0.04 sec)
```

NOTE: All the SQL queries are case insensitive so it does not make any difference if you give ZARA or Zara in WHERE condition.

- 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|
|4|Jill|2007-04-06|220|
|5|Zara|2007-06-06|300|
|5|Zara|2007-02-06|350|
+.......+.......+..............+........................+
7 rows inset(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|
+.............................+
1 row inset(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;
```

```
mysql> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max
-> FROM employee_tbl;
+.........+........+
|least| max  |
+.........+........+
|100|350|
+.........+........+
1 row inset(0.01 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|
|4|Jill|2007-04-06|220|

|5|Zara|2007-06-06|300|
|5|Zara|2007-02-06|350|
+........+........+...............+.........................+
7 rows inset(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|
+.............................+
1 row inset(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;
+........+........+............................+
|id| name | MIN(daily_typing_pages)|
+........+........+............................+
|3|Jack|100|
|4|Jill|220|
+.........+.........+....................................+
5 rows inset(0.00 sec)
```

You can use MIN Function along with MAX function to find out minimum value as well. Try out the following example:

```
mysql> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max
-> FROM employee_tbl;
+..........+.........+
|least| max |
+..........+.........+
|100|350|
+..........+.........+
1 row inset(0.01 sec)
```

- 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 inset(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|
+.....................................+
1 row inset(0.03 sec)
```

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|
|Jill|220.0000|
|John|250.0000|
|Ram|220.0000|
|Zara|325.0000|
+........+....................................+
5 rows inset(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|
|4|Jill|2007-04-06|220|
|5|Zara|2007-06-06|300|
|5|Zara|2007-02-06|350|
+------+------+-----------+--------------------+
7 rows inset(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|
+---------------------------+
1 row inset(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|
|Jill|220|
|John|250|
|Ram|220|
|Zara|650|
+------+--------------------------+
5 rows inset(0.17 sec)
```

- **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 theGROUP 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
```

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
```

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.

- **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:1Changed:1Warnings: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)
```

Exercises:

se FollowingLibrary Database Schema

BOOK (Book_ISBN [PK], Title[Not Null], Publisher_ Name [FK],price[Check Price>0], Date_Of_Publication,Book_Copy),

- Design and Develop SQL DML statements for above database using MySQL Client- MySQL Server Data sever.
- Insert at least five records in each table.
- Select details of Books whose Author lives in "Pune".
- Select Book Names from table Book whose copies are in between 10 to 15.
- Update Book Copies as "10" whose Book Publisher is "Tata MacGraw Hill".
- Select the Name of Publisher who supplied maximum books.
- Display name of publishers as per no of books published by them in ascending order.
- Get publisher names who published at least one book written by author name like 'K%'.
- Get book name and Authors names where book written by maximum authors.
- Get publisher names accordingly books published alphabetically
- Find the no of books published in 01 Jan 2014 to till date.
- Delete the book from Book table written by Author 'Korth'.
- (Refer sample database provided with assignment)

Design and develop below database and execute following SQL DML statements using MySQL Client-MySQL Server Data sever.
emp (eno, ename, bdate, title, salary, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) Write an SQL query that returns the project number and name for projects with a budget greater than $100,000.

2) Write an SQL query that returns all works on records where hours worked is less than 10 and theresponsibility is 'Manager'.
3) Write an SQL query that returns the employees (number and name only) who have a title of 'EE' or 'SA'and make more than $35,000.
4) Write an SQL query that returns the employees (name only) in department 'D1' ordered by decreasingsalary.
5) Write an SQL query that returns the departments (all fields) ordered by ascending department name.
6) Write an SQL query that returns the employee name, department name, and employee title.
7) Write an SQL query that returns the project name, hours worked, and project number for all works onrecords where hours > 10.
8) Write an SQL query that returns the project name, department name, and budget for all projects with abudget < $50,000.
9) Write an SQL query that returns the employee numbers and salaries of all employees in the 'Consulting'department ordered by descending salary.
10) Write an SQL query that returns the employee name, project name, employee title, and hours for allworks on records.

Conclusion:

Thus, we have studied and understood SQL DML statements with all functions and operators

Question:

Consider the following table structure for this assignment:
Table Name: **CUSTOMER**
Fields:

Cust_novarchar(10) Primary Key, FnameVarchar(20) Not NULL, MnameVarchar(20), LnameVarchar(20) Not NULL, Address Varchar(30), City Varchar(15),State Varchar(20), Mobile_no Number(15) Not NULL, Occupation Varchar(20), Company_NameVarchar(25).

Perform the following command/operation on the above table:

1) Alter Command

i) Add column salary
ii) Modify any column
iii)Drop column Mname
iv) Rename any column
v) Rename table

Assignment No.2

Title of Assignment: Design at least 10 SQL/NoSQL queries for suitable database application using SQL DML statements: all types of Join, Sub-Query.

Objective: to understand all types of Joins in SQL, Sub queries, operations on views such as insert, update, delete

Relevant Theory

- **Joins:**

"JOIN" is an SQL keyword used to query data from two or more related tables.
The join operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple.

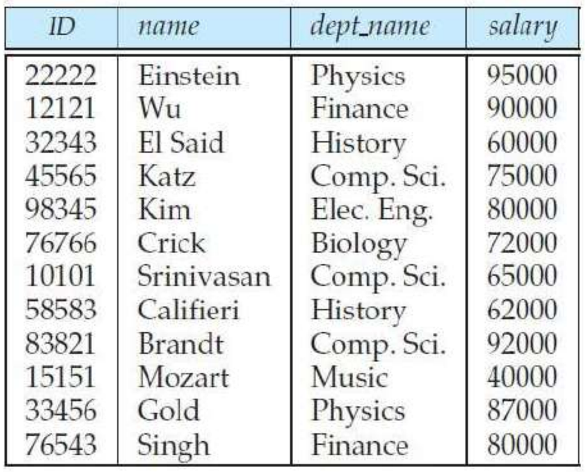
In a database such as MySQL, data is divided into a series of tableswhich are then connected together in SELECT commands to generate the output required.

There are a number of different ways to join relations such as
The Natural Join or Regular Join
Inner Join
Left outer join
Right Outer join
Full outer join

We will see how these join works in MySQL

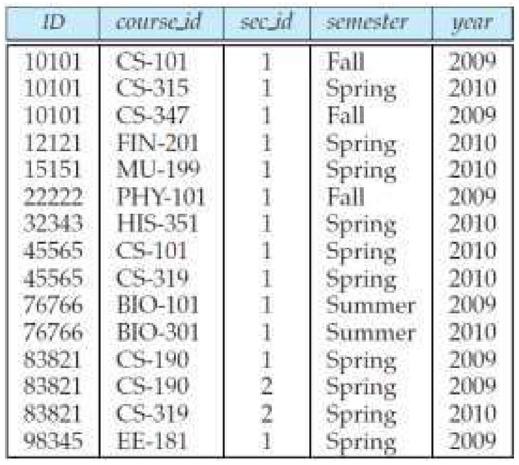
The Natural Join or Regular Join

The natural join operation operates on two relations and produces a relationas the result.
Natural join considers only those pairs of tuples with the same value on those attributes that appear inthe schemas of both relations.
For example consider following two relations



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

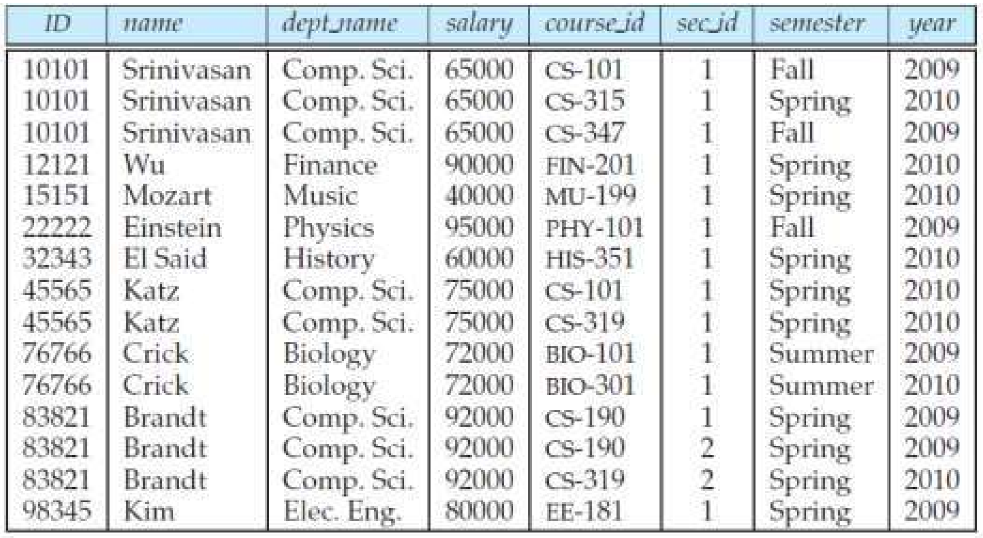
Instructor Relegation



ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Teaches Relation

Computing instructor natural join teaches considers only those pairs of tuples where both the tuple from instructor and the tuple from teaches have the same value on the common attribute, ID.
The result relation, shown in below, has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once.
Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.



ID	name	dept_name	salary	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

The natural join of the instructor relation with the teaches relation.

Consider the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught", which we wrote as:
select name, course id
from instructor, teaches
where instructor.ID= teaches.ID;
This query can be written more concisely using the natural-join operation inSQL as:
select name, course id
from instructor natural join teaches;

To understand how to compute natural join in MySQLconsider below sample data such as :
Mr. Brown, Person number 1, has a phone number 01225 708225
Miss Smith, Person number 2, has a phone number 01225 899360
Mr. Pullen, Person number 3, has a phone number 01380 724040
and also:
Person number 1 is selling property number 1 - Old House Farm
Person number 3 is selling property number 2 - The Willows
Person number 3 is (also) selling property number 3 - Tall Trees
Person number 3 is (also) selling property number 4 - The Melksham Florist
Person number 4 is selling property number 5 - Dun Roamin.

Create two tables such as demo_people and demo_property and insert above data in to it. Then after select all data from both the tables using below expressions

```
mysql> select * from demo_people;

+-------------+---------------+------+

| name        | phone         | pid |

+-------------+---------------+------+

| Mr Brown    | 01225 708225 |    1 |

| Miss Smith | 01225 899360 |    2 |

| MrPullen   | 01380 724040 |    3 |

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

```
mysql> select * from demo_property;

+-------+-------+----------------------+

| pid  |spid | selling              |

+-------+-------+----------------------+

|    1 |    1 | Old House Farm       |

 |   3 |     2 | The Willows         |

 |   3 |       3 | Tall Trees       |

|    3 |    4 | TheMelksham Florist |

|    4 |    5 | Dun Roamin           |

+-------+-------+----------------------+

5 rows in set (0.00 sec)

mysql>
```

Natural join or Regular join on above both tables will be done by following query expression

```
mysql> select name, phone, selling

from demo_people join demo_property

on demo_people.pid = demo_property.pid;

+-----------+-------------+----------------------+

| name      | phone        | selling              |

+-----------+-------------+----------------------+

| MrBrown  | 01225 708225 | Old House Farm       |

| Mr Pullen | 01380 724040 | The Willows          |

| Mr Pullen | 01380 724040 | Tall Trees           |

| Mr Pullen | 01380 724040 | TheMelksham Florist |

+-----------+-------------+----------------------+

4 rows in set (0.01 sec)
```

It will give all records that match the value of appropriate attributes in the two tables, and records in both incoming tables that do not match are discarded.

Such types of join operation that do not preserve non-matched tuples are called as inner join operations.

- ## Outer Join

An outer join does not require each record in the two joined tables to have a matching record or attribute. The joined table retains each record—even if no other matching record exists.
There are in fact three forms of outer join:
The left outer join preserves tuples only in the relation named before (to theleft of) the left outer join operation.
The right outer join preserves tuples only in the relation named after (to theright of) the right outer join operation.
The full outer join preserves tuples in both relations. (In this case left and right refer to the two sides of the JOIN keyword.)
No implicit join-notation for outer joins exists in standard SQL.

- Left Outer Join

The result of a left outer join for tables A and B always contains all records of the "left" table (table A), even if the join-condition does not find any matching record in the "right" table (tableB).
This means that if the ON clause matches zero records in B for a given record in A, the join will still return a row in the result for that record—but with NULL in each column from B.
A left outer join returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link field.
To understand how to compute left outer join in MySQL again consider demo_people and demo_propertytables
After computing LEFT JOIN, result contains all records that match in the same way and IN ADDITION it will shows an extra record for each unmatched record in the left table of the join - thus ensuring that every PERSON is included in result generated by left outer join:

```
mysql> select name, phone, selling

from demo_people left join demo_property

on demo_people.pid = demo_property.pid;

+------------+-------------+---------------------+

| name       | phone       | selling             |

+------------+-------------+---------------------+

| Mr Brown   | 01225 708225 | Old House Farm     |

| Miss Smith | 01225 899360 | NULL               |

| MrPullen   | 01380 724040 | The Willows        |

| MrPullen   | 01380 724040 | Tall Trees         |

| MrPullen   | 01380 724040 | The Melksham Florist |

+------------+-------------+---------------------+

5 rows in set (0.00 sec)

mysql>
```

- ## Right Outer Join

The result of a right outer join for tables A and B always contains all records of the "right" table (table B), even if the join-condition does not find any matching record in the "Left" table (table A).
This means that if the ON clause matches zero records in A for a given record in B, the join will still return a row in the result for that record—but with NULL in each column from A.

A right outer join returns all the values from an inner join plus all values in the right table that do not match to the left table, including rows with NULL (empty) values in the link field.

To understand how to compute right outer join in MySQL again consider demo_people and demo_property tables

After computing RIGHT JOIN, result contains all the records that match and IN ADDITION it will show extra record for each unmatched record in the right table of the join - that means that each property gets a mention even if we don't have seller details:

```
mysql> select name, phone, selling

from demo_people right join demo_property

on demo_people.pid = demo_property.pid;

+-----------+-------------+----------------------+

| name      | phone       | selling              |

+-----------+-------------+----------------------+

| MrBrown   | 01225 708225 | Old House Farm      |

| Mr Pullen | 01380 724040 | The Willows         |

| Mr Pullen | 01380 724040 | Tall Trees          |

| Mr Pullen | 01380 724040 | TheMelksham Florist |

| NULL      | NULL        | Dun Roamin           |

+-----------+-------------+----------------------+

5 rows in set (0.00 sec)

mysql>
```

Full Outer Join

The full outer join is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand side relation, and adds them to the result.
Similarly, it extends with nulls those tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

A standard SQL FULL OUTER join is like a LEFT or RIGHT join, except that it includes all rows from both tables, matching them where possible and filling in with NULLs where there is no match.Here are two tables, apples and oranges:

apples	
Variety	Price
Fuji	5.00
Gala	6.00

Join them on price. Here is the left join:

```
select * from apples as a
left outer join oranges as o on a.price = o.price
```

variety	price	variety	price
Fuji	5	Navel	5
Gala	6	NULL	NULL

And the right joins:

oranges	
Variety	Price
Valencia	4.00
Navel	5.00

```
select * from apples as a
right outer join oranges as o on a.price = o.price
```

variety	price	variety	price
NULL	NULL	Valencia	4
Fuji	5	Navel	5

The FULL OUTER JOIN of these two tables, on price, should give the following result:

variety	price	variety	price
Fuji	5	Navel	5
Gala	6	NULL	NULL
NULL	NULL	Valencia	4

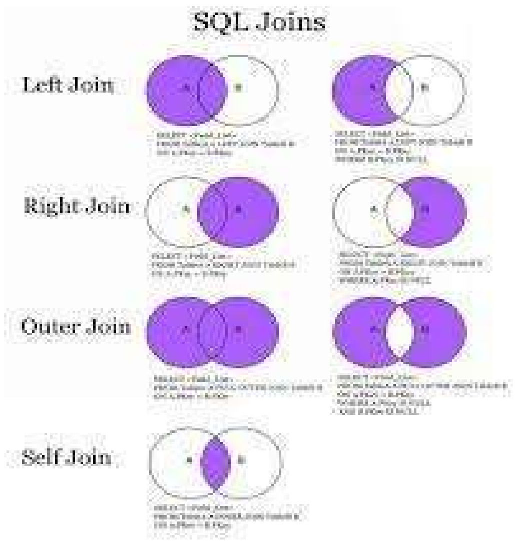
Here is a script to create and populate the example tables, so you can follow along:

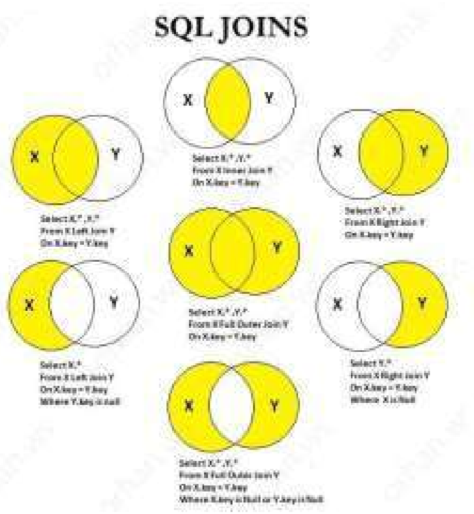
```
Mysql>create table apples (variety char(10) not null primary key, price int not null);
Mysql>create table oranges (variety char(10) not null primary key, price int not null);
Mysql>insert into apples(variety, price) values('Fuji',5),('Gala',6);
Mysql>insert into oranges(variety, price) values('Valencia',4),('Navel',5);
```

One method to simulate a full outer join is to take the union of two outer joins, for example,

```
Mysql>select * from apples as a
left outer join oranges as o on a.price = o.price
union
select * from apples as a
right outer join oranges as o on a.price = o.price
```

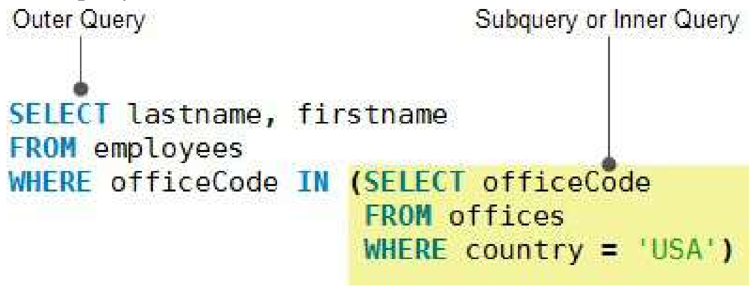
This gives the desired results in this case.





- **Sub Queries**

A MySQL subquery is a query that is nested inside another query such as SELECT, INSERT, UPDATE or DELETE. A MySQL subquery is also can be nested inside another subquery. A MySQL subquery is also called an inner query, while the query that contains the subquery is called an outer query.
For example following subquery that returns employees who locate in the offices in the USA.
The subquery returns all offices codes of the offices that locate in the USA.
The outer query selects the last name and first name of employees whose office code is in the result set returned from the subquery.



```
SELECT lastname, firstname
FROM employees
WHERE officeCode IN (SELECT officeCode
                     FROM offices
                     WHERE country = 'USA')
```

You can use a subquery anywhere an expression can be used. A subquery also must be enclosed in parentheses.
MySQL subquery within a WHERE clause
MySQL subquery with comparison operators
If a subquery returns a single value, you can use comparison operators to compare it with the expression in the WHERE clause.
For example, the following query returns the customer who has the maximum payment.

```
Mysql>SELECT customerNumber,checkNumber,amount
FROM payments
WHERE amount = (
  SELECT MAX(amount)
    FROM payments
)
```

Other comparison operators can be used such as greater than (>), less than(<), etc.
For example, you can find customer whose payment is greater than the average payment.
A subquery is used to calculate the average payment by using the AVG aggregate function.
The outer query selects payments that are greater than the average payment returned from the subquery.

```
Mysql>SELECT
      customerNumber,
      checkNumber,

      amount
FROM
payments

WHERE amount > (

 SELECT
   AVG(amount)
   FROM payments )
```

MySQL subquery with IN and NOT IN operators

If a subquery returns more than one value, you can use other operators such as IN or NOT IN operator in the WHERE clause.
For example, you can use a subquery with NOT IN operator to find customer who has not ordered any product as follows:

```
Mysql>SELECT customername
FROM customers
WHERE customerNumberNOT IN (
   SELECT DISTINCT customernumber
   FROM orders
)
```

MySQL subquery with EXISTS and NOT EXISTS
When a subquery is used with EXISTS or NOT EXISTS operator, a subquery returns a Boolean value of TRUE or FALSE. The subquery acts as an existence check. In the following example, we select a list of customers who have at least one order with total sales greater than 10K.
First, we build a query that checks if there is at least one order with total sales greater than 10K:

```
Mysql>SELECT priceEach * quantityOrdered
FROM orderdetails
WHERE priceEach * quantityOrdered> 10000
GROUP BY orderNumber
```

The query returns some records so that when we use it as a subquery, it will return TRUE; therefore the whole query will return all customers:

```
Mysql>SELECT customerName
FROM customers
WHERE EXISTS (
   SELECT priceEach * quantityOrdered
   FROM orderdetails
   WHERE priceEach * quantityOrdered> 10000
   GROUP BY orderNumber
)
```

If you replace the EXISTS by NOT EXIST in the query, it will not return any record at all.

MySQL subquery in FROM clause
When you use a subquery in the FROM clause, the result set returned from a subquery is used as a table.
This table is referred to as a derived table or materialized subquery.
The following subquery finds the maximum, minimum and average number of items in sale orders:

```
SELECT max(items),
      min(items),
      floor(avg(items))
FROM
(SELECT orderNumber,
   count(orderNumber) AS items
FROM orderdetails
GROUP BY orderNumber) AS lineitems
```

Refer Sample Database: University Database provided with assignment

- For all instructors in the university who have taught some course, find their names and the courseID of all courses they taught using natural join.
- List the names of instructors along with the titles of courses that they teach using natural join.
- Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters using subqueries.
- Find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester usingsubqueries.
- Find the total number of (distinct) students who have taken course sections taught by the instructorwith ID 110011.
- Find the names of all instructors whose salary is greater than at least one instructor in the Biologydepartment.
- Find the names of all instructors that have a salary value greater than that of each instructor in theBiology department.
- Find the departments that have the highest average salary.
- Find all students who have taken all courses offered in the Biology department.

- Compute left outer join between students and takes relation.
- Find all students who have not taken a course using left outer join.
- Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, evenif no student from the Comp. Sci. department has taken the course section.
- Create a view as faculty to access all the data from instructor relation except salary.
- Create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section.

Conclusion:

Thus we have studied and understood Joins, Sub-Query successfully.

Assignment:

1. From the following tables write a SQL query to find the salesperson and customer who reside in the same city. Return Salesman, cust_name and city.
2. From the following tables write a SQL query to find those orders where the order amount exists between 500 and 2000. Return ord_no, purch_amt, cust_name, city.
3. From the following tables write a SQL query to display the customer name, customer city, grade, salesman, salesman city. The results should be sorted by ascending customer_id.

Assignment No. 3

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

Objectives: To understand the basic operation of MongoDB.

Theory:-

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:

```
>usemydb
switched to dbmydb
```

To check your currently selected database use the command db

```
>db
myd
b
```

If you want to check your databases list, then use the command show dbs.

```
>show dbs
local0.78125G
B
test0.23012GB
```

Your created database (mydb) is not present in list. To display database you need to insert atleast onedocument into it.

```
>db.movie.insert({"name":"ABC"})
>show dbs
local0.78125GB
mydb0.23012G
B
test0.23012GB
```

In mongodb default database is test. If you didn't create any database then collections will be stored in testdatabase.

dropDatabase() Method

MongoDB db.dropDatabase() command is used to drop a existing database.Syntax:
Basic syntax of dropDatabase() command is as follows:
db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test'database
Example:
First, check the list available databases by using the command show dbs

```
>show dbs
local0.78125GB
mydb0.23012G
B
test0.23012GB
>
```

If you want to delete new database <mydb>, then dropDatabase() command would be as follows:

```
>usemydb
switched to dbmydb
>db.dropDatabase()
>{"dropped":"mydb","ok":1}
>
```

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)
```

In the command, name is name of collection to be created. Options is a document and used to specify configuration of collection

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only name of the collection. Following is the list ofoptions you can use:

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a collection fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
autoIndexID	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If If cappedis true, then you need to specify this field also.
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

While inserting the document, MongoDB first checks size field of capped collection, then it checks maxfield.
Examples:
Basic syntax of createCollection() method without options is as follows

```
>use test
switched to db test
>db.createCollection("mycollection")
{"ok":1}
>
```

The drop() Method
MongoDB'sdb.collection.drop() is used to drop a collection from the database.Syntax:
Basic syntax of drop() command is as followsdb.COLLECTION_NAME.drop()

DATA TYPES OF MONGODB:-

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

Binary data : This datatype is used to store binay data.
Code : This datatype is used to store javascript code into document. Regular expression : This datatype is used to store regular expression

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB'sinsert() or save()method.Syntax
Basic syntax of insert() command is as follows:

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

Example

```
>db.mycol.insert({
  _id:ObjectId(7df78ad8902c),
title:'MongoDB Overview',
description:'MongoDB is no sql
database',
tags:['mongodb','database','NoSQL'],
likes:100
})
```

Here mycol is our collection name, as created in previous tutorial. If the collection doesn't exist in thedatabase, then MongoDB will create this collection and then insert document into it.
In the inserted document if we don't specify the _id parameter, then MongoDB assigns an unique ObjectIdfor this document.
_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided asfollows:
_id:ObjectId(4 bytes timestamp,3 bytes machine id,2 bytes process id,3 bytes incrementer)

The find() Method
To query data from MongoDB collection, you need to use MongoDB'sfind() method.
Syntax
Basic syntax of find() method is as follows

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

find() method will display all the documents in a non structured way.The pretty() Method

To display the results in a formatted way, you can use pretty() method.
Syntax:

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

OR in MongoDB
Syntax:

To query documents based on the OR condition, you need to use $or keyword. Basic syntax of OR isshown below:

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

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)
```

Example

Consider the mycolcollectioin has following data.

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

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'

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

>db.mycol.find()
{"_id":ObjectId(5983548781331adf45ec5),"title":"New MongoDB Tutorial"}
{"_id":ObjectId(5983548781331adf45ec6),"title":"NoSQL Overview"}
>
```

The remove() Method

MongoDB'sremove() method is used to remove document from the collection. remove() method acceptstwo parameters. One is deletion criteria and second is justOne flag

deletion criteria : (Optional) deletion criteria according to documents will be removed.

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)
```

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:

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

returns everything in the collection c.

Querying for a simple type is as easy as specifying the value that you are looking for. For example, to find all documents where the value for "age" is 27, we can add that key/value pair to the query document:

```
>db.users.find({"age" : 27})
```

If we have a string we want to match, such as a "username" key with the value "abc",we use that key/value pair instead:

```
>db.users.find({"username" : "abc"})
```

Multiple conditions can be strung together by adding more key/value pairs to the query document, which gets interpreted as “condition1 AND condition2 AND … AND conditionN.” For instance, to get all users who are 27-year-olds with the username “abc” we can query for the following:

```
>db.users.find({"username" : "abc", "age" : 27})
```

Query Criteria

Queries can go beyond the exact matching described in the previous section; they can match more complexcriteria, such as ranges, OR-clauses, and negation.

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, 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}})
```

These types of range queries are often useful for dates. For example, to find people who registered before January 1, 2007, we can do this:

```
> start = new Date("01/01/2007")
>db.users.find({"registered" : {"$lt" : start}})
```

An exact match on a date is less useful, because dates are only stored with millisecond precision. Often you want a whole day, week, or month, making a range query necessary. To query for documents where a key’s value is not equal to a certain value, you must use another conditional operator, "$ne", which stands for “not equal.” If you want to find all users who do not have the username “abc” you can query for them using this:
>db.users.find({"username" : {"$ne" : "abc"}})"$ne" can be used with any

type.OR Queries

There are two ways to do an OR query in MongoDB. "$in" can be used to query for a variety of values fora single key. "$or" is more general; it can be used to query for any of the given values across multiple keys. If you have more than one possible value to match for a single key, use an array of criteria with "$in". For instance, suppose we were running a train and the winning ticket numbers were 725, 542, and 390. To find all three of these documents, we can construct the following query:

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

"$in"is very flexible and allows you to specify criteria of different types as well as values. For example, if we are gradually migrating our schema to use usernames instead of user ID numbers, we can query for either by using this:

```
>db.users.find({"user_id" : {"$in" : [12345, "abc"]})
```

This matches documents with a "user_id" equal to 12345, and documents with a "user_id" equal to "abc".If"$in" is given an array with a single value, it behaves the same as directly matching the value.

For instance,
{ticket_no : {$in : [725]}} matches the same documents as {ticket_no : 725}.
The opposite of "$in" is "$nin", which returns documents that don't match any of the criteria in the array. Ifwe want to return all of the people who didn't win anything in the train, we can query for them with this:

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

This query returns everyone who did not have tickets with those numbers. "$in" gives you an OR query fora single key, but what if we need to find documents where "ticket_no" is 725 or "winner" is true? For this type of query, we'll need to use the "$or" conditional. "$or" takes an array of possible criteria. In the train case, using "$or" would look like this:

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

"$or"can contain other conditionals. If, for example, we want to match any of the three "ticket_no" values or the "winner" key, we can use this:

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

With a normal AND-type query, you want to narrow your results down as far as possible in as few arguments as possible. OR-type queries are the opposite: they are most efficient if the first argumentsmatch as many documents as possible.

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

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

"$not" can be particularly useful in conjunction with regular expressions to find all documents that don'tmatch a given pattern

Limits, Skips, and Sorts

The most common query options are limiting the number of results returned, skipping a number of results,and sorting. All of these options must be added before a query is sent to the database.

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

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

If there are fewer than three documents matching your query in the collection, only the number of matching documents will be returned; limit sets an upper limit, not a lower limit.skipworks similarly to limit:

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

This will skip the first three matching documents and return the rest of the matches. Ifthere are less than three documents in your collection, it will not return any documents.sorttakes an object: a set of key/value pairs where the keys are key names and thevalues are the sort directions. Sort direction can be 1 (ascending) or -1 (descending). IfCursors | 57multiple keys are given, the results will be sorted in that order. For instance, to sort theresults by "username" ascending and "age" descending, we do the following:

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

These three methods can be combined. This is often handy for pagination. For example,suppose that you are running an online store and someone searches for mp3. If youwant 50 results per page sorted by price from high to low, you can do the following:

```
>db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

If they click Next Page to see more results, you can simply add a skip to the query,which will skip over the first 50 matches (which the user already saw on page 1):

```
>db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

However, large skips are not very performant, so there are suggestions on avoidingthem in a moment.

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. For example, if the array is a list of fruits, like this:

```
>db.food.insert({"fruit" : ["apple", "banana",

"peach"]})
```
the following query:

```
>db.food.find({"fruit" : "banana"})
```

will successfully match the document. We can query for it in much the same way as though we had a document that looked like the (illegal) document: {"fruit" : "apple", "fruit" : "banana", "fruit" : "peach"}.

$all
If you need to match arrays by more than one element, you can use "$all". This allows you to match a listof elements. For example, suppose we created a collection with three elements:
```
>db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
>db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
>db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

Then we can find all documents with both "apple" and "banana" elements by querying with "$all":

```
>db.food.find({fruit : {$all : ["apple", "banana"]}})
{"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
{"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

Order does not matter. Notice "banana" comes before "apple" in the second result. Using a one-element array with "$all" is equivalent to not using "$all". For instance, {fruit : {$all : ['apple']} will match thesame documents as {fruit : 'apple'}. You can also query by exact match using the entire array. However, exact match will not match a document if any elements are missing or superfluous. For example, this will match the first document shown previously:
```
>db.food.find({"fruit" : ["apple", "banana",
"peach"]})
```
But this will not:
```
>db.food.find({"fruit" : ["apple",
"banana"]})
```
and neither will this:
```
>db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

$size
A useful conditional for querying arrays is "$size", which allows you to query for arrays of a given size.Here’s an example:

```
>db.food.find({"fruit" : {"$size" : 3}})
```

One common query is to get a range of sizes. "$size" cannot be combined with another $ conditional (in this example, "$gt"), but this query can be accomplished by adding a "size" key to the document. Then, every time you add an element to the array, increment the value of "size". If the original update looked likethis:

```
>db.food.update({"$push" : {"fruit" :
"strawberry"}})
```

it can simply be changed to this:

```
>db.food.update({"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

Incrementing is extremely fast, so any performance penalty is negligible. Storing documents like thisallows you to do queries such as this:

```
>db.food.find({"size" : {"$gt" : 3}})
```

Unfortunately, this technique doesn’t work as well with the "$addToSet" operator. The $slice operator As mentioned earlier in this chapter, the optional second argument to find specifies the keys to be returned.The special "$slice" operator can be used to return a subset of elements for an array key.
For example, suppose we had a blog post document and we wanted to return the first 10 comments:

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

Alternatively, if we wanted the last 10 comments, we could use -10:

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

"$slice" can also return pages in the middle of the results by taking an offset and the number of elements toreturn:

```
>db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

This would skip the first 23 elements and return the 24th through 34th. If there are fewer than 34 elementsin the array, it will return as many as possible.

Conclusion:

Thus we have studied and understood Mongodb CRUD operations successfully.

Questions:

1. Create a collection named as Restaurant.
2. Write a query to insert multiple documents into the collection Restaurant.
3. Write a query to update many documents into the collection restaurant.
4. Write a query to delete the inserted document from the collection.

Assignment No.4

Title of Assignment: Write a PL/SQL stored procedure and function.

Objective:

To learn about MySQL stored procedures, their advantages and disadvantages.
To learn about variables in stored procedure,its scope, how to declare and use variables.To learn how to write MySQL stored procedures with parameters.
To learn how to use MySQL IF statement to execute a block of SQL code based on conditions.To learn how to use MySQL CASE statements to construct complex conditionals.
TO learn how to use various loop statements in MySQL including WHILE, REPEAT and LOOP to run ablock of code repeatedly based on a condition.
To learn how to create stored functions using CREATE FUNCTION statement

Relevant Theory

- **Definition of stored procedures**

A stored procedure is a segment of declarative SQL statements stored inside the database catalog. A storedprocedure can be invoked by triggers, other stored procedures or applications such as Java, C#, PHP, etc.
A stored procedure that calls itself is known as a recursive stored procedure. Most database managementsystem supports recursive stored procedures.
However MySQL does not support it very well. You should check your version of MySQL database beforeimplementing recursive stored procedures in MySQL.

- Stored Procedures in MySQL

MySQL is known as the most popular open source RDBMS which is widely used by both community and enterprise. However, during the first decade of its existence, it did not supportstored procedures, stored functions, triggers and events. Since MySQL version 5.0, those features were added to MySQL database engine to make it more flexible and powerful.

- MySQL stored procedures advantages

Typically stored procedures help increase the performance of the applications. Once created, stored procedures are compiled and stored in the database. However MySQL implements the stored procedures slightly different. MySQL stored procedures are compiled on demand. After compiling a storedprocedure, MySQL puts it to a cache. And MySQL maintains its own stored procedure cache for every single connection. If an application uses a stored procedure multiple times in a single connection, the compiled version is used, otherwise the stored procedure works like a query.
A stored procedure helps reduce the traffic between application and database server because instead of sending multiple lengthy SQL statements, the application has to send only name and parameters of the stored procedure.
Stored procedures are reusable and transparent to any applications. Stored proceduresexpose the database interface to all applications so that developers don’t have to develop functions that are already supportedin stored procedures.
Stored procedures are secure. Database administrator can grant appropriate permissions to applications thataccess stored procedures in the database without giving any permission on the underlying database tables. Besides those advantages, stored procedures have their own disadvantages, which you should be aware ofbefore using the store procedures.

- MySQL stored procedures disadvantages

If you use a lot of stored procedures, the memory usage of every connection that is using those stored procedures will increase substantially. In addition, if you overuse a large number of logical operations

inside store procedures, the CPU usage will also increase because database server is not well-designed for logical operations.
A constructs of stored procedures make it more difficult to develop stored procedures that have complicated business logic.
It is difficult to debug stored procedures. Only few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.

It is not easy to develop and maintain stored procedures. Developing and maintainingstored procedures are often required specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance phases.

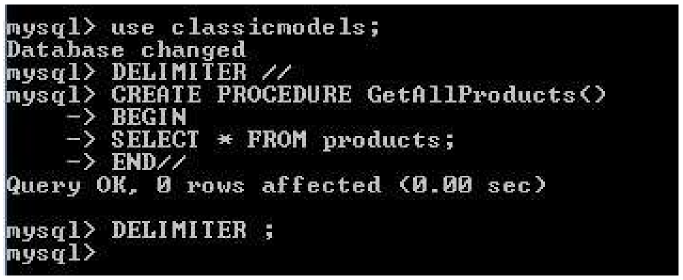
Writing the first MySQL stored procedure
We are going to develop a simple stored procedure named GetAllProducts() to help you get familiar with the syntax. The GetAllProducts() stored procedure selects all products from the products table.
Launch the mysql client tool and type the following commands:

```
DELIMITER //

 CREATE PROCEDURE GetAllProducts()
   BEGIN

   SELECT * FROM products;END
   //

DELIMITER ;
```



```
mysql> use classicmodels;
Database changed
mysql> DELIMITER //
mysql> CREATE PROCEDURE GetAllProducts()
    -> BEGIN
    -> SELECT * FROM products;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
mysql>
```

Let's examine the stored procedure in greater detail:
The first command is DELIMITER //, which is not related to the stored procedure syntax. The DELIMITER statement changes the standard delimiter which is semicolon (;) to another. In this case, the delimiter is changed from the semicolon(;) to double-slashes //.
Why do we have to change the delimiter? Because we want to pass the stored procedure to the server as a whole rather than letting mysql tool to interpret eachstatement at a time. Following the END keyword, we use the delimiter // to indicate the end of the stored procedure. The last command (DELIMITER;) changes the delimiter back to the standard one.
We use the CREATE PROCEDURE statement to create a new stored procedure. We specify the name of stored procedure after the CREATE PROCEDURE statement.
In this case, the name of the stored procedure is GetAllProducts. We put the parentheses after the name of the stored procedure.
The section between BEGIN and END is called the body of the stored procedure. You put the declarative SQL statements in the body to handle business logic.
In this stored procedure, we use a simple SELECT statement to query data from the products table.

- Calling stored procedures

In order to call a stored procedure, you use the following SQL command:

```
CALL STORED_PROCEDURE_NAME()
```

You use the CALL statement to call a stored procedure e.g., to call the GetAllProductsstored procedure, you use the following statement:

```
CALL GetAllProducts();
```

If you execute the statement above, you will get all products in the products table.

MySQL Stored Procedure Variables

A variable is a named data object whose value can change during the stored procedure execution. We typically use the variables in stored procedures to hold the immediate results. These variables are local to the stored procedure.

You must declare a variable before you can use it.

- Declaring variables

To declare a variable inside a stored procedure, you use the DECLARE statement as follows:

```
DECLARE variable_namedatatype(size) DEFAULT default_value;
```

Let's examine the statement above in more detail:
First, you specify the variable name after the DECLARE keyword. The variable name must follow the naming rules of MySQL table column names.
Second, you specify the data type of the variable and its size. A variable can have anyMySQL data types such as INT, VARCHAR, DATETIME, etc.
Third, when you declare a variable, its initial value is NULL. You can assign the variable a default value by using DEFAULT keyword.
For example, we can declare a variable named total_sale with the data type INT and default value 0 as follows:

```
DECLARE total_sale INT DEFAULT 0
```

MySQL allows you to declare two or more variables that share the same data type using a single DECLARE statement as following:

```
DECLARE x, y INT DEFAULT 0
```

We declared two INT variables x and y , and set their default values to zero.

Assigning variables
Once you declared a variable, you can start using it. To assign a variable another value, you use the SET statement, for example:

```
DECLARE total_count INT DEFAULT 0
SET total_count = 10;
```

The value of the total_count variable is 10 after the assignment.
Besides the SET statement, you can use SELECT INTO statement to assign the result of a query to a variable. Notice that the query must return a scalar value.

```
DECLARE total_products INT DEFAULT 0

SELECT COUNT(*) INTO total_products
FROM products
```

In the example above:

First, we declare a variable named total_products and initialize its value to 0.
Then, we used the SELECT INTO statement to assign the total_products variable the number of products that we selected from the products from the products table.

Variables scope

A variable has its own scope, which defines its life time. If you declare a variable inside a stored procedure, it will be out of scope when the END statement of stored procedure reached.
If you declare a variable inside BEGIN END block, it will be out of scope if the END is reached. You can declare two or more variables with the same name in different scopes because a variable is only effective in its own scope. However, declaring variables with the same name in different scopes is not good programming practice.
A variable that begins with the @ sign at the beginning is session variable. It is available and accessible until the session ends.

Introduction to MySQL stored procedure parameters

Almost stored procedures that you develop require parameters. The parameters make the stored procedure more flexible and useful. In MySQL, a parameter has one of three modes IN, OUTor INOUT.

IN – is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure. In addition, the value of an IN parameter is protected. It means that even the value of the IN parameter is changed inside the stored procedure, its original value is retained

after the stored procedure ends. In other words, the stored procedure only works on the copy of the IN parameter.
OUT – the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program. Notice that the stored procedure cannot access the initial value of the OUT parameter when it starts.
INOUT – an INOUT parameter is the combination of IN parameter and OUT parameter. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter and pass the new value back to the calling program.
The syntax of defining a parameter in the stored procedures is as follows:

```
MODE param_nameparam_type(param_size)
```

The MODE could be IN, OUT or INOUT , depending on the purpose of parameter in the stored procedure.
The param_name is the name of the parameter. The name of parameter must follow the naming rules of the column name in MySQL.
Followed the parameter name is its data type and size. Like a variable, the data type of the parameter can by any MySQL data type.
Each parameter is separated by a comma (,) if the stored procedure has more than one parameter.
Let’s practice with some examples to get a better understanding.

MySQL stored procedure parameter examples
IN parameter example
The following example illustrates how to use the IN parameter in the GetOfficeByCountrystored procedure that selects offices located in a specified country.

```
DELIMITER //
CREATE PROCEDURE GetOfficeByCountry(IN countryName VARCHAR(255))
    BEGIN
        SELECT *
         FROM offices
         WHERE country = countryName;
    END //
DELIMITER ;
```

The countryName is the IN parameter of the stored procedure. Inside the stored procedure, we select all offices that locate in the country specified by the countryName parameter.
Suppose, you want to get all offices in the USA, you just need to pass a value (USA) to the stored

procedure as follows:

```
CALL GetOfficeByCountry('USA')
```

To get all offices in France, you pass the France literal string to the GetOfficeByCountrystored procedure as follows:

```
CALL GetOfficeByCountry('France')
```

OUT parameter example
The following stored procedure returns the number of orders by order status. It has two parameters:

orderStatus: IN parameter that is the order status which you want to count the orders.
total: OUT parameter that stores the number of orders for a specific order status.
The following is the source code of the CountOrderByStatus stored procedure.

```
DELIMITER $$

CREATE PROCEDURE
      CountOrderByStatus(IN
      orderStatusVARCHAR(25),


  OUT total
INT)BEGIN

   SELECT
   count(orderNumber)
   INTO total


FROM orders

   WHERE status =
orderStatus;END$$

DELIMITER ;
```

To get the number of shipped orders, we call the CountOrderByStatus stored procedure and pass the orderstatus as Shipped, and also pass an argument (@total) to get the return value.

```
CALL

CountOrderByStatus('Shipped',@total)

;SELECT @total;
```

To get the number of orders that are in process, we call the CountOrderByStatus stored

procedure as follows:CALL CountOrderByStatus('in process',@total);

```
SELECT @total AS  total_in_process;
```

INOUT parameter example

The following example demonstrates how to use INOUT parameter in the stored procedure.

```
DELIMITER $$

CREATE PROCEDURE set_counter(INOUT count
INT(4),IN inc INT(4))BEGIN

   SET count = count +
inc;

END$$

DELIMITER ;
```

How it works

The set_counter stored procedure accepts one INOUT parameter (count) and one IN parameter (inc).Inside the stored procedure, we increase the counter (count) by the value of the incparameter.
See how we call the set_counter stored procedure:

```
SET @counter = 1;
CALL set_counter(@counter,1); --
2 CALL  set_counter(@counter,1);
--                3                CALL
set_counter(@counter,5);     --     8
SELECT @counter; -- 8
```

MySQL IF Statement

The MySQL IF statement allows you to execute a set of SQL statements based on a certain condition or value of an expression. To form an expression in MySQL, you can combine literals,variables, operators, and even functions. An expression can return three value TRUE, FALSE or NULL.

MySQL IF statement syntax
The following illustrates the syntax of the IF statement:

```
IF if_expression THEN commands

  [ELSEIF elseif_expression THEN
  commands][ELSE commands]
  END IF;
```

If the if_expression evaluates to TRUE the commands in the IF branch will execute. If it evaluates to FALSE, MySQL will check the elseif_expression and execute the commands in ELSEIF branch if the elseif_expression evaluates to TRUE.
The IF statement may have multiple ELSEIF branches to check multiple expressions. If no expression evaluates to TRUE, the commands in the ELSE branch will execute.

MySQL IF statement examples
Let’s take a look at an example of how to use MySQL IF statements.

```
DELIMITER $$
```

```
    CREATE PROCEDURE
    GetCustomerLevel(in
    p_customerNumberint(11),
    outp_customerLevel  varchar(10))
BEGIN
    DECLARE creditlim double;

    SELECT creditlimit INTO
    creditlimFROM customers
    WHERE customerNumber = p_customerNumber;

    IF creditlim> 50000 THEN
    SET p_customerLevel = 'PLATINUM';
    ELSEIF (creditlim<= 50000 AND creditlim>= 10000)

    THEN
    SET p_customerLevel = 'GOLD';
    ELSEIF creditlim< 10000 THEN
        SET p_customerLevel =
        'SILVER';
    END IF;

END$$
```

We pass customer number to the stored procedure to get customer level based on credit limit. We use IF ELSEIF and ELSE statement to check customer credit limit against multiple values.

MySQL CASE Statement
Besides the IF statement, MySQL also provides an alternative conditional statement called MySQL CASE.The MySQL CASE statement makes the code more readable and efficient.
There are two forms of the CASE statements: simple and searched CASE statements.

Simple CASE statement
Let's take a look at the syntax of the simple CASE statement:

```
ASE  case_expression
   WHEN when_expression_1 THEN
   commandsWHEN when_expression_2
   THEN commands
   ...
   ELSE
commandsEND
CASE;
```

You use the simple CASE statement to check the value of an expression against a set of unique values.

The case_expression can be any valid expression. We compare the value of thecase_expression with when_expression in each WHEN clause
e.g., when_expression_1, when_expression_2, etc.
If the value of the case_expressionandwhen_expression_n are equal, the commands in the corresponding WHEN branch executes.
In case none of the when_expression in the WHEN clause matches the value of thecase_expression, the commands in the ELSE clause will execute. The ELSE clause is optional. If you omit the ELSE clause andno match found, MySQL will raise an error.

The following example illustrates how to use the simple CASE statement:

```
DELIMITER $$

CREATE PROCEDURE
    GetCustomerShipping(in
    p_customerNumberint(11),
    outp_shiping    varchar(50))

BEGIN

  DECLARE customerCountryvarchar(50);

  SELECT country INTO
  customerCountryFROM customers

  WHERE customerNumber = p_customerNumber;

  CASE
    customerCountry
    WHEN  'USA'
    THEN

      SET p_shiping = '2-day
    Shipping';WHEN 'Canada'
    THEN

      SET p_shiping = '3-day
    Shipping';ELSE

      SET p_shiping = '5-day
  Shipping';END CASE;


END$$


SET @customerNo = 112;

SELECT country into
@countryFROM customers
WHERE customernumber = @customerNo;

CALL

GetCustomerShipping(@customerNo,@shipping);

SELECT @customerNo AS Customer,
    @country   AS Country,
    @shipping  AS Shipping;
```

Searched CASE statement

The simple CASE statement only allows you match a value of an expression against a set of distinct values. In order to perform more complex matches such as ranges you use the searchedCASE statement. The searched CASE statement is equivalent to the IF statement, however its construct is much more readable.

The following illustrates the syntax of the searched CASE statement:

```
CASE
   WHEN condition_1 THEN
   commandsWHEN condition_2
   THEN commands
   ...
   ELSE
commandsEND
CASE;
```

MySQL evaluates each condition in the WHEN clause until it finds a condition whose value isTRUE, then corresponding commands in the THEN clause will execute.
If no condition is TRUE , the command in the ELSE clause will execute. If you don't specify theELSE clause and no condition is TRUE, MySQL will issue an error message.
MySQL does not allow you to have empty commands in the THEN or ELSE clause. If you don't want to handle the logic in the ELSE clause while preventing MySQL raise an error, you can put an empty BEGIN END block in the ELSE clause.

The following example demonstrates using searched CASE statement to find customerlevelSILVER, GOLD or PLATINUM based on customer's credit limit.

```
DELIMITER $$


CREATE PROCEDURE
   GetCustomerLevel(in
   p_customerNumberint(11),


outp_customerLevel varchar(10))
BEGIN

   DECLARE creditlim double;


   SELECT creditlimit INTO
   creditlimFROM customers

WHERE customerNumber = p_customerNumber;


   CASE

      WHEN creditlim> 50000 THEN

        SET p_customerLevel = 'PLATINUM';

      WHEN (creditlim<= 50000 AND creditlim>= 10000)
        THENSET p_customerLevel = 'GOLD';

      WHEN creditlim< 10000 THEN
        SET p_customerLevel =
        'SILVER';
```

```
    END CASE;


  END$$
```

If the credit limit is greater than 50K, then the customer is PLATINUM customer less than 50K and greaterthan 10K, then the customer is GOLD customer less than 10K, then the customer is SILVER customer.
We can test our stored procedure by executing the following test script:

```
CALL
GetCustomerLevel(112,@level);
SELECT @level AS 'Customer
Level';
```

Loop in Stored Procedures
MySQL provides loop statements that allow you to execute a block of SQL code repeatedly based on acondition. There are three loop statements in MySQL: WHILE, REPEAT and LOOP.

WHILE loop
The syntax of the WHILE statement is as follows:

```
WHILE expression
  DOStatements
END WHILE
```

The WHILE loop checks the expression at the beginning of each iteration. If theexpression valuates to TRUE, MySQL will executes statements between WHILE and END WHILE untilthe expression evaluates to FALSE. The WHILE loop is called pretest loop because it checksthe expression before the statements execute.

Here is an example of using the WHILE loop statement in stored procedure:

```
DELIMITER $$

 DROP PROCEDURE IF EXISTS
 WhileLoopProc$$CREATE PROCEDURE
 WhileLoopProc()

     BEGIN

          DECLARE x  INT;

          DECLARE str
          VARCHAR(255);SET x = 1;

          SET str =  '';

WHILE x <= 5 DO

                 SET  str = CONCAT(str,x,',');


                 SET  x = x +
          1;END WHILE;
          SELECT str;
```

```
    END$$

  DELIMITE

  R ;
```

In the stored procedure above:
First, we build str string repeatedly until the value of the x variable is greater than 5. Then, we display the final string using the SELECT statement. Notice that if we don't initialize x variable, its default valueis NULL. Therefore the condition in the WHILE loop statement is always TRUE and you will have a indefinite loop, which is not expected.

REPEAT loop
The syntax of the REPEAT loop statement is as follows:

```
REPEAT
Statements;
UNTIL
expressionEND
REPEAT
```

First MySQL executes the statements, and then it evaluates the expression. If the expression evaluatesto TRUE, MySQL executes the statements repeatedly until the expression evaluates to FALSE.
Because the REPEAT loop statement checks the expression after the execution of statements thereforethe REPEAT loop statement is also known as post-test loop.
We can rewrite the stored procedure that uses WHILE loop statement above using theREPEAT loop statement:

```
DELIMITER $$
 DROP PROCEDURE IF EXISTS
 RepeatLoopProc$$CREATE PROCEDURE
 RepeatLoopProc()
     BEGIN
          DECLARE x  INT;
          DECLARE str VARCHAR(255);
          SET x = 1;
          SET str = '';
          REPEAT
                  SET  str =
                  CONCAT(str,x,',');SET x =
                  x + 1;
          UNTIL x > 5
          END
          REPEAT;
          SELECT str;
     END$$
DELIMITER
;
```

It is noticed that there is no delimiter semicolon (;) in the UNTIL

expression.LOOP, LEAVE and ITERATE Statements
The LEAVE statement allows you to exit the loop immediately without waiting for checking the condition.The LEAVE statement works like the break statement in other languages such as PHP,

C/C++, Java, etc.
The ITERATE statement allows you to skip the entire code under it and start a new iteration.The ITERATE statement is similar to the continue statement in PHP, C/C++, Java, etc.
MySQL also gives you a LOOP statement that allows you to execute a block of code repeatedly with anadditional flexibility of using a loop label.
The following is an example of using the LOOP loop statement.

```
DELIMITER $$

 DROP PROCEDURE IF EXISTS
 LOOPLoopProc$$CREATE PROCEDURE
 LOOPLoopProc()

    BEGIN

        DECLARE x  INT;


DECLARE str
        VARCHAR(255);SET
        x = 1;

        SET str = '';
        loop_label:
        LOOP

              IF  x> 10 THEN

                LEAVE
                 loop_label;


                 END  IF;


                 SET x = x + 1;

              IF (x mod 2) THEN
                ITERATE
                loop_label;

              ELSE

                SET  str =
              CONCAT(str,x,',');END
              IF;


        END
        LOOP;

        SELECT
        str;
    END$$
DELIMITE
R ;
```

The stored procedure only constructs string with even numbers e.g., 2, 4, 6, etc.
We put a loop_label loop label before the LOOP statement. If the value of x is greater than 10, the loop is terminated because of the LEAVE statement. If the value of the x is an odd number, the ITERATE statement ignores everything below it and starts a new iteration. If the value of the x is an even number, the block in the ELSE statement will build the string with even numbers.

MySQL Stored Function

A stored function is a special kind stored program that returns a single value. You use stored functions to encapsulate common formulas or business rules that may be reusable among SQL statements or stored programs.
Different from a stored procedure, you can use a stored function in SQL statements wherever an expressionis used. This helps improve the readability and maintainability of the procedural code.

MySQL stored function syntax
The following illustrates the simplest syntax for creating a new stored function:

```
CREATE FUNCTION function_name(param1,param2,…)
   RETURNS datatype
  [NOT] DETERMINISTIC
statements
```

- First, you specify the name of the stored function after CREATE FUNCTION keywords.
- Second, you list all parameters of the stored function. By default, all parameters are implicitly INparameters. You cannot specify IN, OUT or INOUT modifiers to the parameters.
- Third, you must specify the data type of the return value in the RETURNS statement. It can be anyvalid MySQL data types.
- Fourth, for the same input parameters, if the stored function returns the same result, it is considereddeterministic and not deterministic otherwise.
  You have to decide whether a stored function is deterministic or not. If you declare it incorrectly, the stored function may produced an unexpected result, or the available optimization is not used which degrade the performance.
- Fifth, you write the code in the statements section. It can be a single statement or a compound statement. Inside the statements section, you have to specify at least one RETURN statement.

The RETURN statement returns a value to the caller. Whenever the RETURN statement is reached, thestored function's execution is terminated immediately.

MySQL stored function example
The following example is a function that returns level of customer based on credit limit.

```
DELIMITER $$


CREATE FUNCTION CustomerLevel(p_creditLimit double) RETURNS
  VARCHAR(10)DETERMINISTIC

BEGIN

  DECLARE lvlvarchar(10);
```

```
 IF p_creditLimit> 50000
       THENSET lvl =
       'PLATINUM';

   ELSEIF (p_creditLimit<= 50000 AND p_creditLimit>=
       10000) THENSET lvl = 'GOLD';

   ELSEIF p_creditLimit< 10000
       THENSET lvl = 'SILVER';

   END IF;


   RETURN
(lvl);END
```

Now we can call the CustomerLevel() in an SQL SELECT statement as

```
follows:SELECT customerName,
       CustomerLevel(creditLimit)
FROM customers;
```

We also rewrite the GetCustomerLevel() stored procedure that we developed in theMySQL IF statement as follows:

```
DELIMITER $$

CREATE PROCEDURE
    GetCustomerLevel(IN
    p_customerNumber INT(11),
    OUT p_customerLevel  varchar(10)
)
BEGI
N
    DECLARE creditlim DOUBLE;

    SELECT creditlimit INTO
    creditlimFROM customers
    WHERE customerNumber = p_customerNumber;

    SELECT
    CUSTOMERLEVEL(creditlim)INTO
    p_customerLevel;
 END
```

As you can see, the GetCustomerLevel() stored procedure is much more readable when usingthe CustomerLevel() stored function.
Notice that a stored function returns a single value only. If you include a SELECT statement without INTO clause, you will get an error.
In addition, if a stored function contains SQL statements, you should not use it inside other SQL statements; otherwise the stored function will cause the performance of the SQL statements to degrade.

Conclusion:

Thus we have studied and created procedure and functions successfully.

Assignment No. 4

Problem statement: Aggregation with suitable example using

MongoDB.Objectives: To understand the basic Aggregation queries

of MongoDB.

Theory:

Aggregation Pipeline

Aggregation pipeline gives you a way to transform and combine documents in your collection. You do itby passing the documents through a pipeline that's somewhat analogous to the Unix "pipe" where yousend output from one command to another to a third, etc.

The simplest aggregation you are probably already familiar with is the SQL group by expression. We already saw the simple count() method, but what if we want to see how many student are male and how many are female?

```
db.student.aggregate ({$group :{_id:'$gender',total: {$sum:1}}})
```

In the shell we have the aggregate helper which takes an array of pipeline operators. For a simple count grouped by something, we only need one such operator and it's called $group. This is the exact analog of GROUP BY in SQL where we create a new document with _id field indicating what field we are grouping by (here it's gender) and other fields usually getting assigned results of some aggregation, in this case we
$sum 1 for each document that matches
aparticular gender.

You probably noticed that the _id field was assigned '$gender' and not 'gender' - the '$' before a field nameindicates that the value of this field from incoming document will be substituted. What are some of the other pipeline operators that we can use? The most common one to use before (and frequently after)
$group would be $match - this is exactly like the find method and it allows us to aggregate only a matchingsubset of our documents, or to exclude some documents from our result.

```
db.student.aggregate ({$match: {weight :{$lt:600}}},
                            {$group: {_id:'$gender', total :{$sum:1},
avgVamp :{$avg:'$vampires'}}}, {$sort:{avgVamp:-1}} )
```

Here we introduced another pipeline operator $sort which does exactly what you would expect, along with it we alsoget $skip and $limit. We also used a $group operator $avg.

MongoDB arrays are powerful and they don't stop us from being able to aggregate on values that are stored inside ofthem. We do need to be able to "flatten" them to properly count everything:

```
db.student.aggregate ({$unwind: 'like'},
{$group: {_id:'$like, total :{$sum:1},student:{$addToSet:'$like'}}},
{$sort :{ total:-1}},{$limit:1} )
```

Here we will find out which food item is loved by the most student and we will also get the list of names ofall thestudent that like it. $sort and $limit in combination allow you to get answers to "top N" types of questions.

$sort:-

The $sort stage has the following prototype form:

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

$sort takes a document that specifies the field(s) to sort by and the respective sort order. <sort order> can have one of the following values:
1 to specify ascending order.
-1 to specify descending order.
{ $meta: "textScore" } to sort by the computed textScore metadata in descending order.

To ascending order for a field or fields to sort by and a value of 1 or -1 to specify an ascending Or descending sort respectively, as in the following example:

```
  db.users.aggregate (
  [

{$sort: {age: -1, posts: 1}}
  ]
  )
```

$limit:-
Limits the number of documents passed to the next stage in the *pipeline*.
The $limit stage has the following prototype form:
{$limit:<positiveinteger>}
$limit takes a positive integer that specifies the maximum number of documents to pass along.
Example
Consider the following example:

```
db.article.aggregate(
{$limit:5}
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit has no effect on the content of the documents it passes.

There is another powerful pipeline operator called $project (analogous to the projection we can specify to find) whichallows you not just to include certain fields, but to create or calculate new fields based on values in existing fields. For23
example, you can use math operators to add together values of several fields before finding out the average, or youcan use string operators to create a new field that's a concatenation of some
existing fields.

$project:-
The $project takes a document that can specify the inclusion of fields, the suppression of the _id field, the addition of new fields, and the resetting the values of existing fields. The specifications have the following forms:

Syntax	Description
<field>: <1 or true>	Specify the inclusion of a field.
_id: <0 or false>	Specify the suppression of the _id field.
<field>: <expression>	Add a new field or reset the value of an existing field.

The following $project stage includes only the _id, title, and the author fields in its output documents:
db.books.aggregate([{ $project : { title : 1 , author : 1 } }])
To include only the title field in the embedded document in the stop field, you can use the dot notation:
db.bookmarks.aggregate([{ $project: { "stop.title": 1 } }])
Or, you can nest the inclusion specification in a document:
db.bookmarks.aggregate([{ $project: { stop: { title: 1 } } }])

$group :-
Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an_id field which contains the distinct group by key. The output

documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group‘s _id field. $group does *not* order its output documents.

The $group stage has the following prototype form:
{$group:{_id:<expression>,<field1>:{<accumulator1>:<expression1>},...}}
The _id field is *mandatory*; however, you can specify an_id value of null to calculate accumulated values for all the input documents as a whole.
The remaining computed fields are *optional* and computed using the <accumulator> operators.
The _id and the <accumulator> expressions can accept any valid *expression*. For more information on expressions, see *Expressions*.
Accumulator Operator

The <accumulator> operator must be one of the following accumulator operators:

Name	Description
$addToSet	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.
$avg	Returns an average for each group. Ignores non-numeric values.
$first	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
$last	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max	Returns the highest expression value for each group.
$min	Returns the lowest expression value for each group.
$push	Returns an array of expression values for each group.
$sum	Returns a sum for each group. Ignores non-numeric values.

$group Operator and Memory
The $group stage has a limit of 100 megabytes of RAM. By default, if the stage exceeds this limit, $group will produce an error. However, to allow for the handling of large datasets, set the allowDiskUse option to true to enable $group operations to write to temporary files. See db.collection.aggregate() method and the aggregate command for details.

Examples

Calculate Count, Sum, and Average

Given a collection sales with the following documents:

```
{"_id":1,"item":"abc","price":10,"quantity":2,"date":ISODate("2014-03-1T08:00:00Z")}
{"_id":2,"item":"jkl","price":20,"quantity":1,"date":ISODate("2014-03-01T09:00:00Z")}
{"_id":3,"item":"xyz","price":5,"quantity":10,"date":ISODate("2014-03-15T09:00:00Z")}
{"_id":4,"item":"xyz","price":5,"quantity":20,"date":ISODate("2014-04-04T11:21:39.736Z")}
{"_id":5,"item":"abc","price":10,"quantity":10,"date":ISODate("2014-04-
04T21:23:13.331Z")}
```

Group by Month, Day, and Year

The following aggregation operation uses the $group stage to group the documents by the month, day, and year and calculates the total price and the average quantity as well as counts the documents per each group:db.sales.aggregate(

```
[
{
$group:{
_id:{month:{$month:"$date"},day:{$dayOfMonth:"$date"},year:{$year:"$date"}},
totalPrice:{$sum:{$multiply:["$price","$quantity"]}},
averageQuantity:{$avg:"$quantity"},
count:{$sum:1}
}
}
]
)
```

The operation returns the following results:

```
{"_id":{"month":3,"day":15,"year":2014},"totalPrice":50,"averageQuantity":10,"count":1}
{"_id":{"month":4,"day":4,"year":2014},"totalPrice":200,"averageQuantity":15,"count":2}
{"_id":{"month":3,"day":1,"year":2014},"totalPrice":40,"averageQuantity":1.5,"count":2}
```

Group by null

The following aggregation operation specifies a group _id of null, calculating the total price and the average quantity as well as counts for all documents in the collection:

```
db.sales.aggregate
([
{
$group:{
_id:null,
totalPrice:{$sum:{$multiply:["$price","$quantity"]}},
averageQuantity:{$avg:"$quantity"},
count:{$sum:1}

}
]

)
```
The operation returns the following result:

```
{"_id":null,"totalPrice":290,"averageQuantity":8.6,"count":5}
```
Retrieve Distinct Values

Given a collection sales with the following documents:

```
{"_id":1,"item":"abc","price":10,"quantity":2,"date":ISODate("2014-03-01T08:00:00Z")}
{"_id":2,"item":"jkl","price":20,"quantity":1,"date":ISODate("2014-03-01T09:00:00Z")}
{"_id":3,"item":"xyz","price":5,"quantity":10,"date":ISODate("2014-03-15T09:00:00Z")}
{"_id":4,"item":"xyz","price":5,"quantity":20,"date":ISODate("2014-04-04T11:21:39.736Z")}
{"_id":5,"item":"abc","price":10,"quantity":10,"date":ISODate("2014-04-04T21:23:13.331Z")}
```
The following aggregation operation uses the $group stage to group the documents by the item to retrieve the distinct item values:

```
db.sales.aggregate([{$group:{_id:"$item"}}])
```
The operation returns the following result:

```
{"_id":"xyz"}
{"_id":"jkl"}
{"_id":"abc"}
```

$unwind:-

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

The $unwind stage has the following prototype form:

```
{ $unwind: <field path> }
```
To specify a field path, prefix the field name with a dollar sign $ and enclose in quotes.

If a value in the field specified by the field path is *not* an array, db.collection.aggregate() generates an error.

If you specify a path for a field that does not exist in an input document, the pipeline ignores the input document and will not output documents for that input document.

If the array holds an empty array ([]) in an input document, the pipeline ignores the input document and will not output documents for that input document.

Examples

Consider an inventory with the following document:

```
{"_id":1,"item":"ABC1",sizes:["S","M","L"]}
```
The following aggregation uses the $unwind stage to output a document for each element in the sizes array:

```
db.inventory.aggregate([{$unwind:"$sizes"}])
```
The operation returns the following results:

```
{"_id":1,"item":"ABC1","sizes":"S"}
{"_id":1,"item":"ABC1","sizes":"M"}
{"_id":1,"item":"ABC1","sizes":"L"}
```
Each document is identical to the input document except for the value of the sizes field which now holds a value from the original sizes array.

$skip:-

Skips over the specified number of *documents* that pass into the stage and passes the remaining documents to the next stage in the *pipeline*.

The $skip stage has the following prototype form:

```
{$skip:<positiveinteger>}
```
$skip takes a positive integer that specifies the maximum number of documents to skip.

Example

Consider the following example:

```
db.article.aggregate({$skip:5});
```
This operation skips the first 5 documents passed to it by the pipeline. $skip has no effect on the content of the documents it passes along the pipeline.

$limit:-

Limits the number of documents passed to the next stage in the*pipeline*.
The $limit stage has the following prototype form:
{$limit:<positiveinteger>}

$limit takes a positive integer that specifies the maximum number of documents to pass along.
Example

Consider the following example:
db.article.aggregate(
{$limit:5}
);
This operation returns only the first 5 documents passed to it from by the pipeline. $limit has no effect on the content of the documents it passes.

Conclusion:

Successfully we have studied and performed Mongodb Aggregation.

Questions:

1. Write an expression by using,
   (i) Group
   (ii) Limit
   (iii) Project
   (iv) Sort
   (v) Match
   (vi) Count
   (vii) Lookup

Assignment No. 4

Problem statement: Indexing with suitable example using MongoDB.

Objectives: To understand the basic indexing of MongoDB.

INDEXING:-

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These *collection scans* are inefficient because they requiremongod to process a larger volume of data than an index for each operation.

Indexes are special data structures that store a small portion of the collection's 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.

Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect. In some cases, MongoDB can use the data from the index to determine which documents match a query. The following diagram illustrates a query that selects documents using an index.

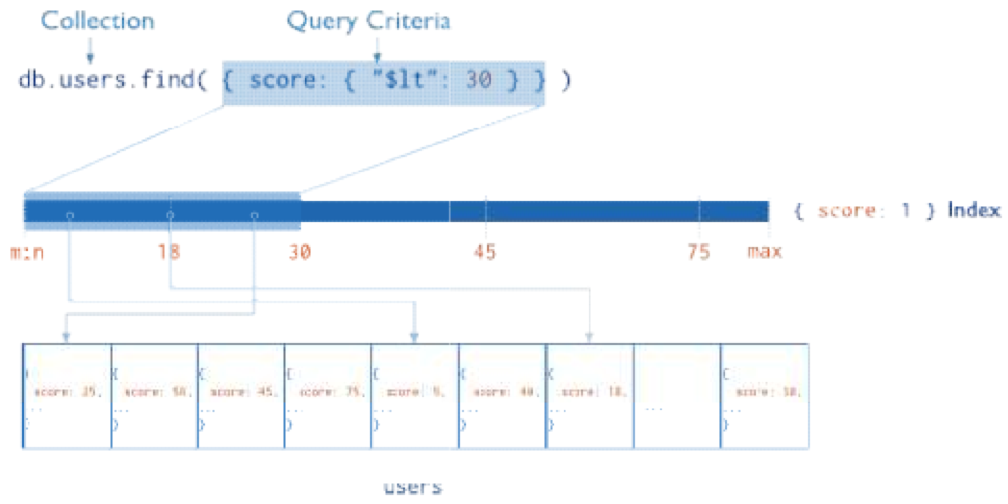


Diagram of a query selecting documents using an index. MongoDB narrows the query by scanning the range of documents with values of score less than 30.

Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

Default _id

All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an_id field with an *ObjectId*value.

The _id index is *unique*, and prevents clients from inserting two documents with the same value for the _id field.

Single Field

In addition to the MongoDB-defined _id index, MongoDB supports user-defined indexes on a *single field of a document*. Consider the following illustration of a single-field index:

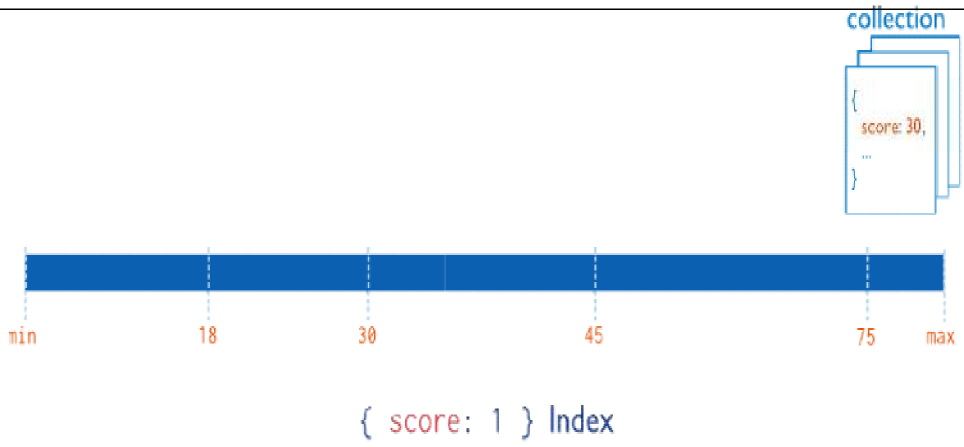


Diagram of an index on the score field (ascending).

Compound Index

MongoDB *also* supports user-defined indexes on multiple fields. These *compound indexes* behave like single-field indexes; *however*, the query can select documents based on additional fields. The order of fields listed in a compound index has significance. For instance, if a compound index consists of {userid:1,score:-1}, the index sorts first by userid and then, within each userid value, sort by score. Consider the following illustration of this compound index:

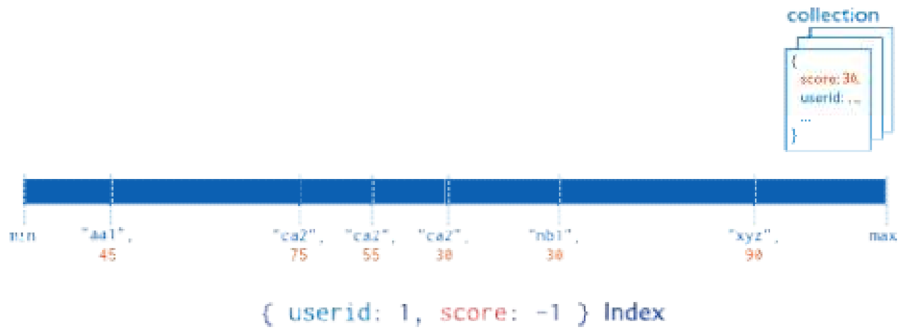


Diagram of a compound index on the userid field (ascending) and the score field (descending). The index sorts first by the userid field and then by the score field.

Multikey Index

MongoDB uses *multikey indexes*to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These *multikey indexes* allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

Consider the following illustration of a multikey index:

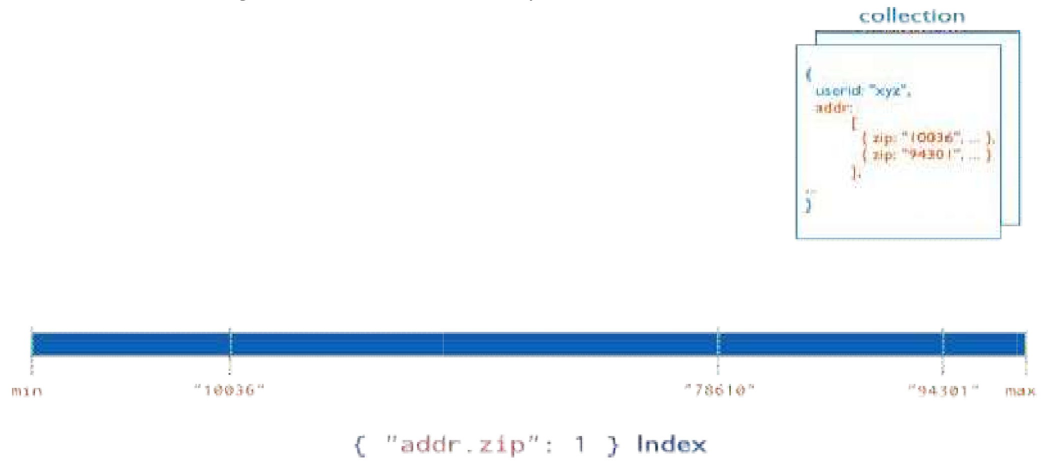


Diagram of a multikey index on the addr.zip field. The addr field contains an array of address documents. The address documents contain the zip field.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* that uses planar geometry when returning results and *2sphere indexes* that use spherical geometry to return results.
Text Indexes
MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. “the”, “a”, “or”) and *stem* the words in a collection to only store root words.
Hashed Indexes
To support *hash based sharding*, MongoDB provides a *hashed index* type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.
Index Properties
Unique Indexes
The *unique* property for an index causes MongoDB to reject duplicate values for the indexed field. To create a *unique index*on a field that already has duplicate values,. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.
Sparse Indexes
The *sparse*property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.
You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.
Create an Index
Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. Users can create indexes for any collection on any field in a *document*. By default, MongoDB creates an index on the _id field of every collection.
Create an Index on a Single Field
To create an index, use ensureIndex() or a similar method from your driver. The ensureIndex() method only creates an index if an index of the same specification does not already exist.
For example, the following operation creates an index on the userid field of the records collection:
db.records.ensureIndex({userid:1})
The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.
The created index will support queries that select on the field userid, such as the following:
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 Compound Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB supports indexes that include content on a single field, as well as *compound indexes* that include content from multiple fields. Continue reading for instructions and examples of building a compound index.
Build a Compound Index
To create a *compound index* use an operation that resembles the following prototype:
db.collection.ensureIndex({a:1,b:1,c: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*.
Example
The following operation will create an index on the item, category, and price fields of the products collection:
db.products.ensureIndex({item:1,category:1,price:1})

Create a Unique Index

MongoDB allows you to specify a *unique constraint* on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields. Additionally, if you want to

create an index on a collection that has existing data that might have duplicate values for the indexed field, you may choose to combine unique enforcement with *duplicate dropping*.
Unique Indexes
To create a *unique index*, consider the following prototype:
db.collection.ensureIndex({a:1},{unique:true})
For example, you may want to create a unique index on the "tax-id": of the accounts collection to prevent storing multiple account records for the same legal entity:
db.accounts.ensureIndex({"tax-id":1},{unique:true})
The *_id index* is a unique index. In some situations you may consider using the _id field itself for this kind of data rather than using a unique index on another field.
In many situations you will want to combine the unique constraint with the sparse option. When MongoDB indexes a field, if a document does not have a value for a field, the index entry for that item will be null. Since unique indexes cannot have duplicate values for a field, without the sparse option, MongoDB will reject the second document and all subsequent documents without the indexed field. Consider the following prototype.
db.collection.ensureIndex({a:1},{unique:true,sparse:true})
You can also enforce a unique constraint on *compound indexes*, as in the following prototype:
db.collection.ensureIndex({a:1,b:1},{unique:true})
These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.
Drop Duplicates
To force the creation of a *unique index*index on a collection with duplicate values in the field you are indexing you can use the dropDups option. This will force MongoDB to create a *unique* index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of ensureIndex():
db.collection.ensureIndex({a:1},{unique:true,dropDups:true})
Create a Sparse Index
Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings.
Prototype
To create a *sparse index* on a field, use an operation that resembles the following prototype:
db.collection.ensureIndex({a:1},{sparse:true})
Example
The following operation, creates a sparse index on the users collection that *only* includes a document in the index if the twitter_name field exists in a document.
db.users.ensureIndex({twitter_name:1},{sparse:true})
The index excludes all documents that do not include the twitter_name field.
Create a Hashed Index
To create a hashed index, specify hashed as the value of the index key, as in the following example:
Example
Specify a hashed index on _id
db.collection.ensureIndex({ _id: "hashed" })

Conclusion:

Successfully we have studied and performed Mongodb Indexing

Questions:

1. Create an index in the database and search the index by using search operator.

Assignment No. 4

Problem statement: Map-reduce with suitable example using MongoDB.

Objectives: To understand the basic map-reduce of MongoDB

MapReduce

MapReduce is the Uzi of aggregation tools. Everything described with count, distinct, and group can be done with MapReduce, and more. It is a method of aggregation that can be easily parallelized across multiple servers. It splits up a problem, sends chunks of it to different machines, and lets each machine solve its part of the problem. When all of the machines are finished, they merge all of the pieces of the solution back into a full solution.

MapReduce has a couple of steps. It starts with the map step, which maps an operation onto every document in a collection. That operation could be either "do nothing" or "emit these keys with X values." There is then an intermediary stage called the shuffle step: keys are grouped and lists of emitted values are created for each key. The reduce takes this list of values and reduces it to a single element. This element is returned to the shuffle step until each key has a list containing a single value: the result. The price of using MapReduce is speed: group is not particularly speedy, but MapReduce is slower and is not supposed to be used in "real time." You run
Map-reduceis a data processing paradigm for condensing large volumes of data into useful aggregated results. MongoDB uses mapReduce command for map-reduce operations. MapReduce is generally used for processing large data sets.
MapReduce Command:
Following is the syntax of the basic mapReduce command:

```
>db.collection.mapReduce(
function(){emit(key,value);},//map function
function(key,values){returnreduceFunction},//reduce function
{
out: collection,
query: document,
sort: document,
limit: number
}
)
```

The map-reduce function first queries the collection, then maps the result documents to emit key-value pairs which is then reduced based on the keys that have multiple values.
In the above syntax:
map is a javascript function that maps a value with a key and emits a key-valur pair
reduce is a javscript function that reduces or groups all the documents having the same key
out specifies the location of the map-reduce query result
query specifies the optional selection criteria for selecting documents
sort specifies the optional sort criteria
limit specifies the optional maximum number of documents to be returned
Emit takes a key (used to reduce the data) and a value.
It's important to note that MongoDB only calls the reduce function for those keys that have multiple values. Therefore the value you emit should mirror the return value from the reduce function.
we will use a mapReduce function on our posts collection to select all the active posts, group them on the basis of user_name and then count the number of posts by each user using the following code:

```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
    {
```

```
query:{status:"active"},
 out:"post_total"
       }
)
```
The above mapReduce query outputs the following result:
```
{
   "result" : "post_total",
   "timeMillis" : 9,
   "counts" : {
      "input" : 4,
      "emit" : 4,
      "reduce" : 2,
      "output" : 2
   },
   "ok" : 1,

}
```
The result shows that a total of 4 documents matched the query (status:"active"), the map function emitted 4 documents with key-value pairs and finally the reduce function grouped mapped documents having the same keys into 2.

To see the result of this mapReduce query use the find operator:
```
>db.posts.mapReduce(
function() { emit(this.user_id,1); },
function(key, values) {return Array.sum(values)},
       {
query:{status:"active"},
out:"post_total"
       }
).find()
```
The above query gives the following result which indicates that both userstom and mark have two posts in active states:
```
{ "_id" : "tom", "value" : 2 }
{ "_id" : "mark", "value" : 2 }
```

Conclusion:
Successfully we have studied and performed Mongodb Mapreduce.

Question:

1. Write a MongoDB query to find the name and address of the restaurants that have the word 'coffee' in their name.
2. Write a MongoDB query to find the name, address, and cuisine of the restaurants that have a cuisine that ends with the letter 'y'.

Assignment No.5

Title of Assignment: Write a PL/SQL block to calculate the grade of minimum 10 students using database cursor.

Objective:

To learn how to use MySQL cursor in stored procedures to iterate through a result set returned by a SELECT statement.

Relevant Theory
Introduction to MySQL cursor
To handle a result set inside a stored procedure, you use a cursor. A cursor allows you to iterate set of rows returned by a query and process each row accordingly.
MySQL cursor is read only, non-scrollable and asensitive.
Read only: you cannot update data in the underlying table through the cursor.
Non-scrollable: you can only fetch rows in the order determined by the SELECT statement. You cannot fetch rows in the reversed order. In addition, you cannot skip rows or jump to a specific row in the result set.
A sensitive: there are two kinds of cursors: a sensitive cursor and insensitive cursor. An asensitive cursor points to the actual data, whereas an insensitive cursor uses a temporary copy of the data. An a sensitive cursor performs faster than an insensitive cursor because it does not have to make a temporary copy of data. However, any change that made to the data from other connections will affect the data that is being used by an a sensitive cursor, therefore it is safer if you don't update the data that is being used by an a sensitive cursor. MySQL cursor is a sensitive.

You can use MySQL cursors in stored procedures, stored functions and triggers.

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;
```

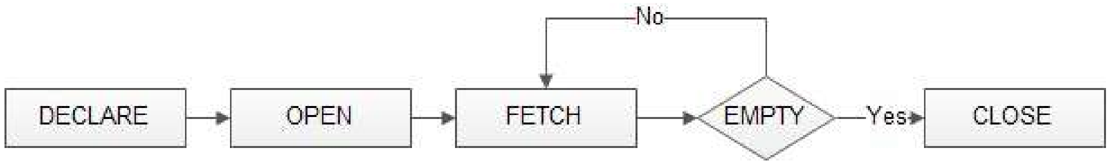
After that, you can check to see if there is any row available before fetching it.
Finally, you call the CLOSE statement to deactivate the cursor and release the memory associated with it as follows:

```
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 theemployees 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_listvarchar(4000))
BEGIN

   DECLARE v_finished INTEGER DEFAULT 0;
      DECLARE v_emailvarchar(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;
```

Conclusion:

Successfully we have studied and created Cursor.

Question:

Write a PL/SQL block to calculate the grade of minimum 10 students using database cursor on student table congaing roll no and marks for three subjects.

Assignment No. 6

Problem statement: Connectivity with MongoDB using any Java application.

Objectives: To understand Connectivity with MongoDB using any Java application

Theory:-

Download MongoDB Java Driver
The first thing you will need to do is download the MongoDB Java Driver.

Put Driver in Classpath
Once you've downloaded the Java driver, put the JAR in your Java CLASSPATH. A common location is in the <JRE_HOME>\lib\ext folder of your Java JRE installation.

Establish A Connection
To make a connection, we will need to use the MongoClient class instance. The MongoClient class is thread safe which means only one instance is needed even if multiple threads are being used.
Note: You will need to know the IP address and port of the database you wish to connect to. If the database does not exist, MongoDB will create it for you.
Use the following code to make your connection.

```
importcom.mongodb.MongoClient;
importcom.mongodb.MongoException;
importcom.mongodb.WriteConcern;
importcom.mongodb.DB;
importcom.mongodb.DBCollection;
importcom.mongodb.BasicDBObject;
importcom.mongodb.DBObject;
importcom.mongodb.DBCursor;
importcom.mongodb.ServerAddress;

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

DB db = mongoClient.getDB( "mydb" );
```

User authentication
If you are running MongoDB in secure mode you will need user authentication to complete the connection. If so, use the following sample code:

```
MongoClientmongoClient = new MongoClient();
DB db = mongoClient.getDB("test");
booleanauth = db.authenticate(myUserName, myPassword);
```

To clean up your resources and dispose of an instance call:

```
mongoClient.close();
```

Get Database
Once a connection is established you can then use the getDB() method to get a database by name.

```
DB db = mongoClient.getDB( "mydb" );
```

To display all databases:

```
List dbs = mongo.getDatabaseNames();
for(String db : dbs)
{
  System.out.println(db);
}
```

Getting Collections

Getting collections is a key step to being able to query and manipulate data. You can retrieve a collection by name using getCollection().

```
DBCollectioncoll = db.getCollection("testCollection");
To retrieve a list of collection names in the database:
Set colls = db.getCollectionNames();

for (String s : colls)
{
System.out.println(s);
}
```

Conclusion:

Successfully we have performed the database connectivity.

Question:

1. Implement an real time java application where you use the Mongodb database connectivity .