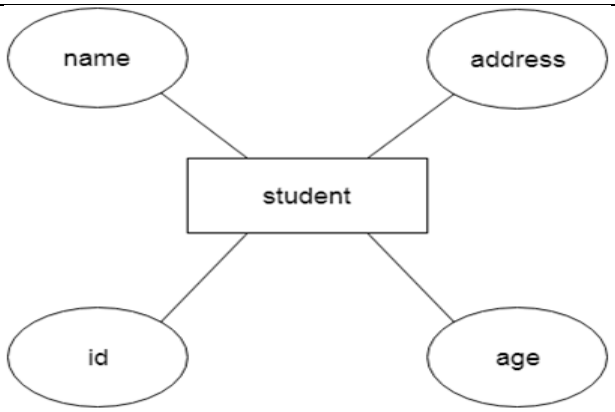
Assignment Number – A1	
Title & Problem Statement	**ER Modeling and Normalization:** Decide a case study related to real time application in group of 2-3 students and formulate a problem statement for application to be developed. Propose a Conceptual Design using ER features using tools like ERD plus, ER Win etc. (Identifying entities, relationships between entities, attributes, keys, cardinalities, generalization, specialization etc.) Convert the ER diagram into relational tables and normalize Relational data model.
Objectives	1. To study ER Modeling to design the database. 2. To study Normalization to reduce data redundancy and functional dependency
Outcomes	Students will be able to: 1. Understand ER Modeling to design the database. 2. Understand and apply normalization to reduce data redundancy and functional dependency.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Tool- ERD plus

Theory

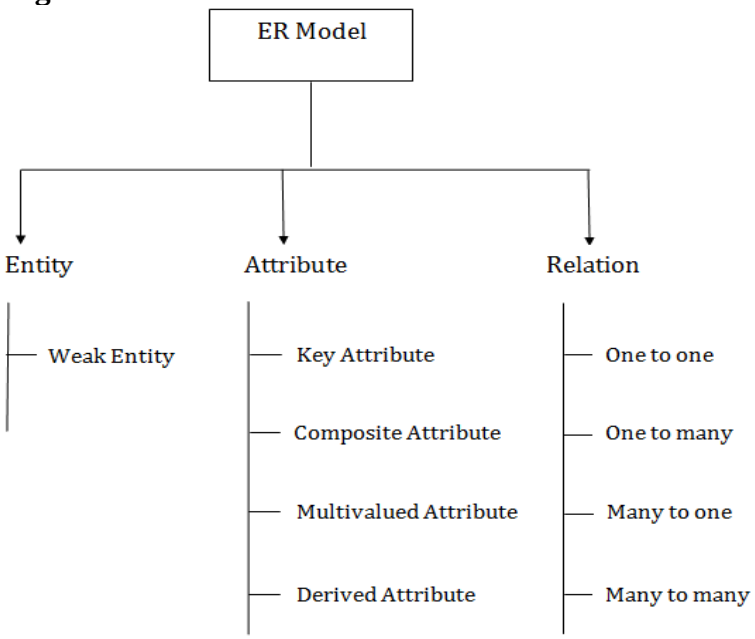
ER (Entity Relationship) Modeling –

ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system. It develops a conceptual design for the database. It also develops a very simple and easy to design view of data. In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram. For example, Suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc and there will be a relationship between them.

Example:



Component of ER Diagram –

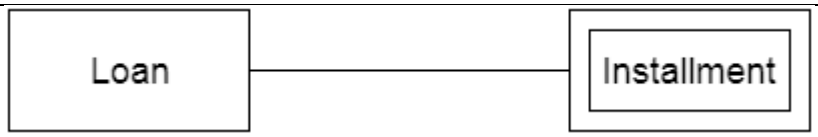


1. **Entity –**

   An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles. Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.

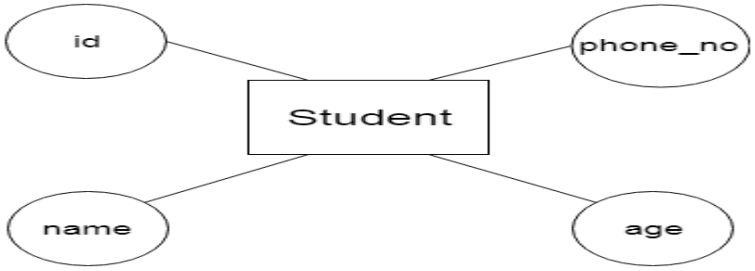
   Weak Entity –

   An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



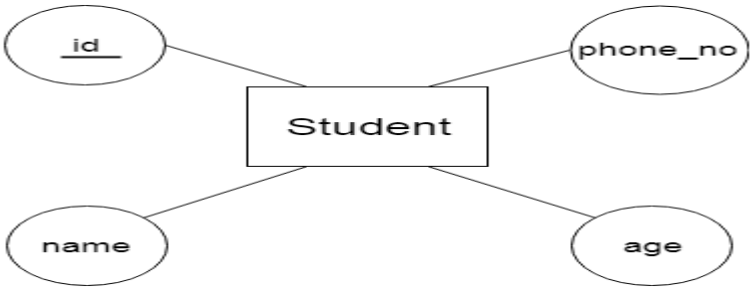
2. **Attribute –**

The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute. For example, id, age, contact number, name, etc. can be attributes of a student.



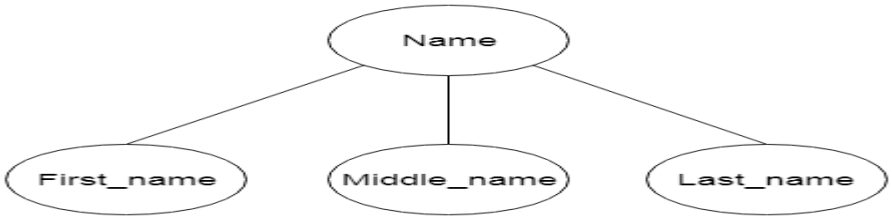
a. **Key Attribute** –

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



b. **Composite Attribute** –

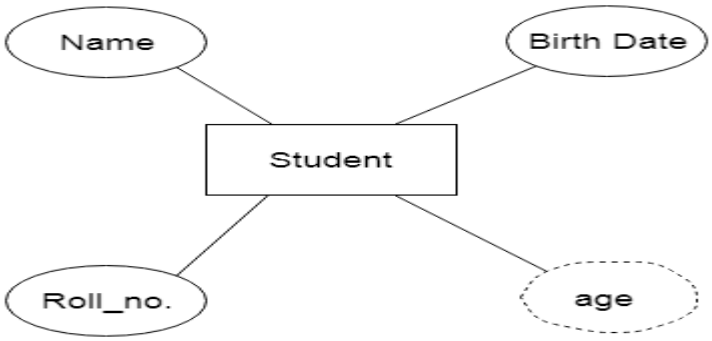
An attribute that is composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. **Multivalued Attribute** –

An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute. For example, a student can have more than one phone number.

d. **Derived Attribute** –
An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse. For example, A person's age changes over time and can be derived from another attribute like Date of birth.



3. Relationship –
A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

Types of relationship are as follows:

a. One-to-One Relationship –
When only one instance of an entity is associated with the relationship, then it is known as one to one relationship. For example, A female can marry to one male, and a male can marry to one female.

b. One-to-Many Relationship –
When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship. For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.

c. Many-to-One Relationship –

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship. For example, Student enrolls for only one course, but a course can have many students.

d. Many-to-Many Relationship –

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship. For example, Employee can assign by many projects and project can have many employees.

Normalization –

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.
- The normal form is used to reduce redundancy from the database table.

Need of Normalization –
The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide us in creating a good database structure.

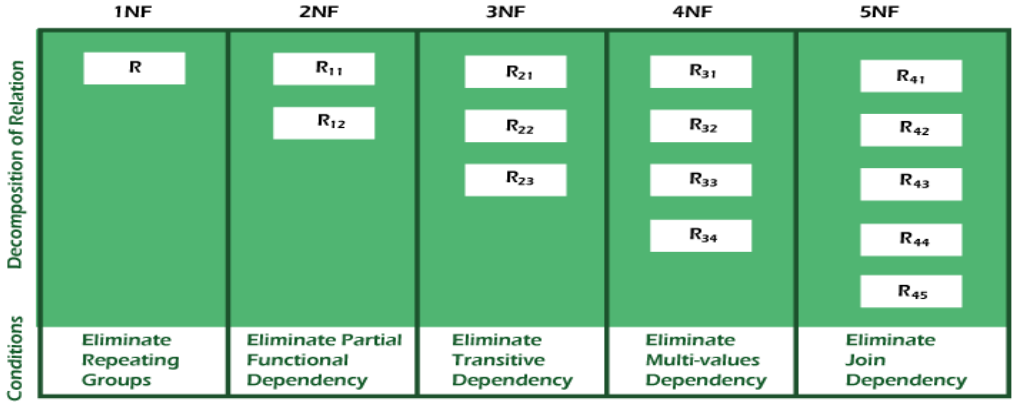
Data modification anomalies can be categorized into three types –

- **Insert Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
- **Delete Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
- **Update Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

Types of Normal Forms-

Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

Following are the various types of Normal forms:



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.
4NF	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.

Conclusion	In this assignment we are able to: 1. Understand ER Modeling to design the database. 2. Understand Normalization to reduce data redundancy and functional dependency.
Questions	1. What is ER Modeling? 2. What are the components of ER Diagram? 3. What is an entity? 4. What is a weak entity?Explain the difference between weak and strong entity. 5. What is an attribute? Explain types of attributes? 6. What is a relationship? Explain types of relationships? 7. What is Insert Anomaly, Delete Anomaly, Update Anomaly? 8. What is Normalization? Explain its different forms. 9. What is difference between 3NF and BCNF?

Assignment Number – 2A	
Title & Problem Statement	Design and Develop SQL DDL statements which demonstrate the use of **SQL objects such as Table, View, Index, Sequence, Synonym, different constraints** etc.
Objectives	1. To understand Tables in MySQL with different constraints like Not Null, Primary Key, Foreign key. 2. To understand simple View, create a view with where clause, update View, drop View. 3. To understand Index, Drop Index. 4. To understand Sequence and Synonym
Outcomes	Students will be able to: 1. Understand SQL DDL statements. 2. Perform different SQL Objects.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MySQL/SQL Server/Oracle

Theory

A **database object** is any defined object in a database that is used to store or reference data. Anything which we make from **create command** is known as Database Object. It can be used to hold and manipulate the data. Some of the examples of database objects are :

- Table
- View
- Index
- Sequence
- Synonym

Table – Basic unit of storage; composed rows and columns. Tables in SQL are created with constraints. SQL constraints are used to specify rules for the data in a table. Constraints 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 table. If there is any violation between the constraint and the data action, the action is aborted.

The following constraints are commonly used in SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different

- **PRIMARY KEY** - A combination of NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Use to create and retrieve data from the database very quickly

SQL NOT NULL Constraint

By default, a column can hold NULL values.
The NOT NULL constraint enforces a column to NOT accept NULL values.
This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

Syntax:
Create table table_name (column1 type1 not null, column2 type2 not null, …….);

Example:

```
CREATE TABLE Persons (ID int NOT NULL,
        LastNamevarchar(255) NOT NULL,
        FirstName varchar (255) NOT NULL,
        Age int);
```

The above SQL query ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values.

NOT NULL Constraint with Alter Table:

Syntax:
ALTER TABLE table_name
MODIFY column_name data_type NOT NULL;

Example:

```
ALTER TABLE Persons
        MODIFY LastNamevarchar(255) NOT NULL,
        MODIFY FirstName varchar(255) NOT NULL;
```

SQL UNIQUE Constraint

The **Unique** constraint ensures that all values in a column are different. Both the **Unique and Primary key** constraints provide a guarantee for uniqueness for a column or set of columns.

A **Primary key** constraint automatically has a **Unique** constraint. However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

Syntax:

CREATE TABLE table_name (column_name data_type UNIQUE);

Example:

```
CREATE TABLE College(
      College_code varchar(20) UNIQUE,
      College_name varchar(50)
);
```

UNIQUE Constraint With Alter Table

We can also add the UNIQUE constraint to an existing column using the ALTER TABLE command.

Syntax:
ALTER TABLE table_name
ADD UNIQUE(column_name);

Example:

```
ALTER TABLE college
ADD UNIQUE(college_id);
```

UNIQUE Constraint for Multiple Columns

Syntax:
ALTER TABLE table_name
ADD UNIQUE unique_name(column,column2…..);

Example:

```
ALTER TABLE colleges
ADD UNIQUE unique_college(college_id,college_code);
```

SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only one primary key, which may consist of single or multiple fields.

Syntax:

```
CREATE TABLE table_name(
    Column1 data type, Column2 data type…, primary key(column1)
);
```

Example:

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50),
  PRIMARY KEY (college_id)
);
```

Add the PRIMARY KEY constraint to multiple columns:

Example:

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20),
  college_name VARCHAR(50),
  CONSTRAINT College_PK   PRIMARY KEY (college_id, college_code)
);
```

Primary Key Constraint With Alter Table

We can also add the PRIMARY KEY constraint to a column in an existing table using the ALTER TABLE command.

For a Single Column:

```
ALTER TABLE colleges
ADD PRIMARY KEY(college_id);
```

For a Multiple Column:

```
ALTER TABLE colleges
ADD CONSTRAINT college_PK PRIMARY KEY (college_id,college_code);
```

Remove Primary Key Constraint

We can remove the PRIMARY KEY constraint in a table using the DROP clause.

Syntax:
ALTER TABLE table_name DROP primary key;

Example:

```
ALTER TABLE colleges
DROP CONSTRAINTS college_pk;
```

SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Syntax:

```
CREATE TABLE table_name(
        Column1 data type,
        Column2 data type;
        …..,
        FOREIGN KEY (column_name) REFERENCES referenced_table_name
(referenced_column_name)
);
```

Example:

```
-- this table doesn't contain foreign keys
CREATE TABLE Customers (
   id INT,
   name VARCHAR(100),
   age INT,
   PRIMARY KEY(id)
 );

-- create another table named Products
-- add foreign key to the customer_id column
-- the foreign key references the id column of the Customers table

CREATE TABLE Products (
   customer_id INT ,
   name VARCHAR(100),
   FOREIGN KEY (customer_id)  REFERENCES Customers(id)
 );
```

Foreign Key with Alter Table

It is possible to add the FOREIGN KEY constraint to an existing table using the ALTER TABLE command.

Example:

```
CREATE TABLE Customers (
  id INT,
```

```
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT  Customers_PK  PRIMARY KEY (id)
);

CREATE TABLE Orders (
  order_id INT,
  item VARCHAR(40),
  amount INT,
  customer_id INT,
  CONSTRAINT Orders_PK PRIMARY KEY (order_id)
);

-- add foreign key to the customer_id field of Orders
-- the foreign key references the id field of Customers
```

```
ALTER TABLE Orders
ADD FOREIGN KEY (customer_id) REFERENCES Customers(id);
```

Multiple Foreign Keys in a Table

A database table can also have multiple foreign keys

```
--this table doesn't have a foreign key
CREATE TABLE Users (
  id INT PRIMARY KEY,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10)
);

--add foreign key to buyer and seller fields
--foreign key references the id field of the Users table
```

```
CREATE TABLE Transactions (
  transaction_id INT PRIMARY KEY,
  amount INT,
  seller INT,
  buyer INT,
  CONSTRAINT  fk_seller  FOREIGN KEY (seller) REFERENCES Users(id),
  CONSTRAINT  fk_buyer  FOREIGN KEY (buyer) REFERENCES Users(id)
);
```

Remove Foreign Key Constraint

Syntax:
ALTER TABLE table_name
DROP FOREIGN KEY foreign_key_constraint_name;

Example:

```
ALTER TABLE Transactions
DROP FOREIGN KEY fk_seller;
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to set a default value for a column.
The default value will be added to all new records, if no other value is specified.

Syntax:
CREATE TABLE table_name(
column1 datatype1 DEFAULT 'default_value1',
column2 datatype2 DEFAULT 'default_value2',
column3 datatype3 DEFAULT 'default_value3',
…….);

Example:

```
CREATE TABLE Person(
        Fname VARCHAR(20),
        Lname VARCHAR(20),
        city VARCHAR(20),
        country VARCHAR(20) DEFAULT 'India'
);
```

DEFAULT with ALTER TABLE:

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'India';
```

Remove DEFAULT Constraint

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

View:

View is a data object which does not contain any data. Contents of the view are the resultant of a base table. They are operated just like base table but they don't contain any data of their own.

The difference between a view and a table is that views are definitions built on top of other tables (or views). If data is changed in the underlying table, the same change is reflected in the view. A view can be built on top of a single or multiple tables.

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

Syntax:

CREATE VIEW view_name

AS SELECT column1,column2,……. ...

FROM table_name WHERE condition;

Example

If you have the Northwind database you can see that it has several views installed by default.

The view "Current Product List" lists all active products (products that are not discontinued) from the "Products" table. The view is created with the following SQL:

```
CREATE VIEW  US_Customers  AS
SELECT CustomerName, ContactName
 FROM Customers
WHERE Country =’US’;
```

Then, we can **query the view as follows:**

SELECT * FROM US_Customers;

SQL Updating a View:

You can update a view by using the following syntax:

Syntax:

CREATE OR REPLACE VIEW *view_name*

AS SELECT *column1*, *column2*, …

FROM *table_name* WHERE *condition*;

Example:

```
CREATE OR REPLACE VIEW US_Customers AS
  SELECT CustomerName, ContactName, City
   FROM Customers
  WHERE Country ='US';
```

SQL Dropping a View

You can delete a view with the DROP VIEW command.

Syntax:

DROP VIEW view_name;

Example:

```
DROP VIEW US_Customers;
```

SQL INDEX

High system performance is of prime importance in almost all database systems. Most businesses invest heavily in hardware so that data retrievals and manipulations can be faster. But there is a limit to hardware investments a business can make.
Optimizing your database is a cheaper and better solution.

- The slowness in the response time is usually due to the records being stored randomly in database tables.
- Search queries have to loop through the entire randomly stored records one after the other to locate the desired data.
- This results in poor performance databases when it comes to retrieving data from large tables
- Indexes come in handy in such situations. Indexes sort data in an organized sequential way(alphabetically sorted list). It is easier to lookup names that have been sorted in alphabetical order than ones that are not sorted.
- INDEXes are created on the column(s) that will be used to filter the data.
- Using indexes on tables that are frequently updated can result in poor performance. This is because MySQL creates a new index block every time that data is added or updated in the table. Generally, indexes should be used on tables whose data does not change frequently but is used a lot in select search queries.

In SQL, the INDEX constraint in a column makes it faster to retrieve data when querying that column.

Create index Syntax:

CREATE INDEX index_name
ON table_name(Column1,column2,column3…..);

Example

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR (20) NOT NULL,
  college_name VARCHAR (50),
  PRIMARY KEY (college_id)
);
```

Create Index

```
CREATE INDEX college_index
ON Colleges(college_code);
```

Here, the SQL command creates an index named college_index on the Colleges table using the college_code column.

CREATE UNIQUE INDEX for Unique Values

If you want to create indexes for unique values in a column, we use the CREATE UNIQUE INDEX constraint.

Example:

```
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR (20) NOT NULL,
  college_name VARCHAR (50)
);
```

Create unique index

```
CREATE UNIQUE INDEX college_index
ON Colleges(college_code);
```

Here, the SQL command creates a unique index named college_index on the Colleges table using the college_code column.

View Index

To view the indexes defined on a particular table, the syntax is as follows

SHOW INDEXES FROM table_name;

Remove Index from Tables

To remove INDEX from a table, we can use the DROP INDEX command.

Example:

```
DROP INDEX Colleges.college_index;
```

SQL Sequences

The sequences in SQL is a database object that generates a sequence of unique integer values. They are frequently used in databases because many applications require each row in a table to contain a unique value and sequences provide an easy way to generate them. Sequences are a feature of many SQL database management systems, such as Oracle, PostgreSQL, SQL server, and IBM DB2.

A sequence is created using the CREATE SEQUENCE statement in SQL. The statement specifies the name of the sequence, the starting value, the increment, and other properties of the sequence.

Syntax:

CREATE SEQUENCE Sequence_Name
START WITH Initial_Value
INCREMENT BY Increment_Value
MINVALUE Minimum_Value
MAXVALUE Maximum_Value
CYCLE|NOCYCLE;

Here,
Sequence_Name − This specifies the name of the sequence.
Initial_Value − This specifies the starting value from where the sequence should start.
Increment_Value − This specifies the value by which the sequence will increment by itself. This can be valued positively or negatively.
Minimum_Value − This specifies the minimum value of the sequence.
Maximum_Value − This specifies the maximum value of the sequence.
Cycle − When the sequence reaches its Maximum_Value, it starts again from the beginning.
Nocycle − An exception will be thrown if the sequence exceeds the Maximum_Value.

Example

First of all, let us try to create a table “STUDENTS” using the following query −

```
CREATE TABLE STUDENTS (ID INT, NAME CHAR(20), AGE INT NOT NULL);
```

Now, let us insert some records in the table using INSERT statements as shown in the query below −

```
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Dhruv', '20');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Arjun', '23');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Dev', '25');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Riya', '19');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Aarohi','24');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Lisa', '20');
INSERT INTO STUDENTS(ID, NAME, AGE) VALUES(NULL, 'Roy', '24');
```

Let's verify whether the table STUDENTS is created or not using the following query −

```
SELECT * FROM STUDENTS;
```

Now, let us try to create a sequence in SQL using the following statement −

```
CREATE SEQUENCE My_Sequence AS INT
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 5
CYCLE;
```

Update Sequence:

```
UPDATE STUDENTS SET ID = NEXTVALUE FOR My_Sequence;
```

AUTO INCREMENT Field in MySQL for Sequencing

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

```
CREATE TABLE Persons (
    ID int NOT NULL AUTO_INCREMENT,
        LastName varchar(255) NOT NULL,
        FirstName varchar(255),
        Age int,
    PRIMARY KEY (ID)
);
```

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "ID" column (a unique value will be added automatically):

```
INSERT INTO Persons(FirstName,LastName) VALUES ('Lars','Monsen');
```

SQL statement above would insert a new record into the "Persons" table. The "ID" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

SQL Synonym

A SYNONYM provides another name for a database object, referred to as the original object, that may exist on a local or another server. A synonym belongs to a schema, the name of the synonym should be unique. A synonym cannot be the original object for an additional synonym and synonym cannot refer to a user-defined function.

There are two types to SYNONYMS they are

- PUBLIC SYNONYM
- PRIVATE SYNONYM

If you create a synonym as public then it can be accessed by any other user with qualifying the synonym name i.e. the user doesn't have to mention the owner name while accessing the synonym.

The query below results in an entry for each synonym in the database. This query provides details about synonym metadata such as the name of synonym and name of the base object.

SELECT * FROM sys.synonyms ;

Note: Synonyms are database dependent and cannot be accessed by other databases.

Syntax:

```
CREATE SYNONYM synonymname
   FOR servername.databasename.schemaname.objectname;
   GO
```

Example 1:

Let us assume Geektabtable of GFGdatabase, Geeekshschema on server named Server1. To reference this table from another server, Server2, an application would have to use four-part named

Server1.GFG.Geeeksh.Geektab. Also, if the location of table were to change, for example, to another server, application would have to be modified to reflect that change.

To address both these issues, one can create synonym, Geektable, on Server2for Geektabtable on Server1. Now, the application only has to use single-part name, Geektable, to point Geektab table. Also, if location of the Geektab table changes, you will have to modify synonym, Geektable, to point to new location of Geektab table.

Now, let us create synonym for Geektab table of GFG database, Geeeksh schema on server named Server1.

```
CREATE SYNONYM Geektable
FOR Server1.GFG.Geeeksh.Geektab;
GO
```

Example 2:
CREATING SYNONYMS. To create a synonym for SCOTT emp table give the following command.

```
Create synonym employee
    for scott.emp
```

Dropping Synonyms

To drop a synonym use the DROP SYNONYM statement.

```
drop synonym employee;
```

Conclusion	In this assignment we are able to: 1. Understand Tables in MySQL with different constraints like Not Null, Primary Key, Foreign key. 2. Understand simple View, create a view with where clause, update View, drop View. 3. Understand Index, Drop Index. 4. Understand Sequence and Synonym
Questions	1. What are Constraints? 2. What is Primary Key and Foreign Key? 3. What is the difference between Primary Key and Foreign Key? 4. What is the view? How to create and remove view in SQL? 5. What is the use of Indexes? 6. What are the sequences? How does it work? 7. What are synonyms? Explain the different types of Synonyms.

Assignment Number – 2B	
Title & Problem Statement	**Design at least 10 SQL queries for suitable database application using SQL DML statements:** Insert, Select, Update, delete with operators, functions, and set operator.
Objectives	1. To Understand and execute various DML statements. 2. To understand aggregation commands like sum, avg, etc 3. To understand set operators
Outcomes	Students will be able to: 1. Execute Insert, Select, Update, Delete with operators 2. Understand the use of Aggregate functions and different clauses in SQL 3. Understand different SET operations.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MySQL/SQL Server/Oracle

Theory

A database is an organized collection of data. A DBMS is a complex set of software programs that controls the organization, storage, management, and retrieval of data in a database. DBMS contains information about a particular enterprise

- Collection of interrelated data
- Set of programs to access the data
- An environment that is both convenient and efficient to use.

A **relational database** is a database that has a collection of tables of data items, all of which is formally described and organized according to the relational model. Data in a single table represents a relation. In typical solutions, tables may have additionally defined relationships with each other.

SQL (STRUCTURED QUERY LANGUAGE)

SQL is an ANSI and ISO standard computer language for creating and manipulating databases. SQL allows the user to create, update, delete, and retrieve data from a database.

DDL

Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:

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

DML

Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain

DCL

Data Control Language (DCL) statements. Some examples:

- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

TCL

Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT
- SET TRANSACTION - Change transaction options like isolation level and what rollbacksegment to use.

To start working with mysql
$ mysql –u root –p

Enter password : *******

Mysql>

Try some basic commands like version (), current_date, Now () etc

- To get started on your own database, first check which databases currently exists.

mysql> show databases;

- To create a new database, issue the "create database" command:

mysql> create database webdb;

- To the select a database, issue the "use" command:

mysql> use webdb;

Data types:

- **VARCHAR(size)**
  - A variable-length string between 1 and 255 characters in length; for exampleVARCHAR(25). You must define a length when creating a VARCHAR field.
- **CHAR (size)**
  - A fixed-length string between 1 and 255 characters in length (for example CHAR (5)), right-padded with spaces to the specified length when stored. Defining a length is not required, but the default is 1.
- **DATE** -- from 4712BC to 4714 AD
  - A date in YYYY-MM-DD format, between 1000-01-01 and 9999-12-31. For example, December 30th, 1973 would be stored as 1973-12-30
- **INT** -A normal-sized integer that can be signed or unsigned. If signed, the allowable range is from -2147483648 to 2147483647. If unsigned, the allowable range is from 0 to 4294967295. You can specify a width of up to 11 digits.
- **FLOAT (M, D)** - A floating-point number that cannot be unsigned. You can define the display length (M) and the number of decimals (D). This is not required and will default to 10,2, where 2 is the number of decimals and 10 is the total number of digits (including decimals). Decimal precision can go to 24 places for a FLOAT.

DDL & DML Commands

To create a table, use the CREATE TABLE command:

create table r (A_1D_1, A_2D_2, ..., A_nD_n,
(integrity-constraint$_1$),
...,
(integrity-constraint$_k$));

- r is the name of the relation
- each A_i is an attribute name in the schema of relation r
- D_i is the data type of attribute A_i

Example:

create table *branch*(
branch_name **char**(15)**,**
branch_city **char(**30),
assets **int);**

Check the design/structure of the table by issuing **DESCRIBE / DESC command**

DESCRIBE Table_name;
Or
DESC Table_name;

To change the table structure or add/modify constraints on a table, use the **ALTER TABLE** command

- To add new column to the existing table
  ALTER TABLE tablename
  ADD Columnnamedatatype;

- To delete a column from the existing table
  ALTER TABLE tablename
  DROP Columnname ;

- To modify the data type/ size of the existing field
  ALTER TABLE tablename
  MODIFY ColumnNameNewDatatype;

- To change the name of the existing field

  ALTER TABLE tablename

  RENAME ColumnName NewColmnName Datatype;

To delete/ remove a table use DROP command

DROP TABLE Table_Name;

To rename a table

RENAME TABLE tablename1 TO tablename2;

DML Commands

Data Manipulation Language (DML) statements are used for managing data tables.

Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain

1. **To insert records into a table, use the INSERT command**

Syntax:

INSERT INTO *tablename* **VALUES** (V1, V2, V3);

Example:

INSERT INTO *account* **VALUES** ('A-9732', 'Perryridge', 1200)

To insert multiple records::

INSERT INTO *tablename* **VALUES** (V1, V2,V3), (V4,V5,V6)

where Vi are column values.

2. **To Update records of a table, use the UPDATE command**

a. To update all records of table

UPDATE tablename

SET columnname = new value

b. To UPDATE specific records, use where clause**.**

Example,

```
UPDATE dept
SET dpt_name = 'ACCOUNTS'
WHERE dpt_no = 10;
```

3. **To delete records of a table, use the DELETE command**
   a. To delete all records from a table

      DELETE FROM tablename;

   b. To delete a particular record from a table

      DELETE FROM tablename
      WHERE columnName = value;

4. **To see the records/tuples of the table use SELECT statement**

   a. The data returned is stored in a result table, called the result-set.

      SELECT * FROM *table_name*;

   b. Will display all records and all fields of the table.

      SELECT *column1*, *column2*,...FROM *table_name*;

   c. Will display all records with only specified columns of the table.

      SELECT *column1*, *column2*,...
      FROM *table_name*
      WHERE ColumnName = values;

      Will display records which satisfies the above conditions.

TRUNCATE Command

Will delete all the records from the specified table. This being the DDL command cannot be rolled back.

Syntax:

TRUNCATE TABLE Table_name;

Example:

TRUNCATE TABLE customer;

DROP Command

To delete or remove a table permanently from database, we use DROP command

Syntax:

DROP TABLE TableName;

Example:

DROP TABLE customer;

- **Aggregate functions**

  An aggregate function performs a calculation on a set of values and returns a single value. MySQL provides many aggregate functions that include AVG, COUNT, SUM, MIN, MAX, etc. An aggregate function ignores NULL values when it performs calculation except for the COUNT function.

Name	Description
AVG()	Return the average value of the argument
COUNT()	Return a count of the number of rows returned
COUNT(DISTINCT)	Return the count of a number of different values
MAX()	Return the maximum value
MIN()	Return the minimum value
SUM()	Return the sum

Example:

1) SELECT COUNT(ProductID) FROM Products;
   Above SQL statement finds the number of products.
2) SELECT AVG(Price) FROM Products;
3) SELECT SUM(Quantity) FROM OrderDetails;
4) SELECT MIN(Price) AS SmallestPrice FROM Products;

- **The SQL GROUP BY Statement:**

  The GROUP BY clause, which is an optional part of the SELECT statement, groups a set of rows into a set of summary rows by values of columns or expressions. The GROUP BY clause returns one row for each group. In other words, it reduces the number of rows in the result set.

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

GROUP BY Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example
The following SQL statement lists the number of customers in each country:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

- **The SQL HAVING Clause**

  HAVING clause is used in the SELECT statement to specify filter conditions for a group of rows or aggregates. HAVING clause is often used with the GROUP BY clause. When using with the GROUP BY clause, we can apply a filter condition to the columns that appear in the GROUP BY clause. The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

  HAVING Syntax

  ```
  SELECT column_name(s)
  FROM table_name
  WHERE condition
  GROUP BY column_name(s)
  HAVING condition
  ORDER BY column_name(s);
  ```

  The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

  Example

  ```
  SELECT COUNT(CustomerID), Country
  FROM Customers
  ```

```
GROUP BY Country
HAVING COUNT(CustomerID)
```

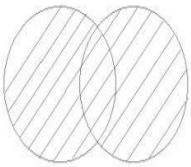
- **Set Operations**

The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, −. Each of the above operations automatically eliminates duplicates; To retain all duplicates use the corresponding multi set versions **union all, intersect all** and **except all.**

1. **UNION Operator**

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

   The syntax of the UNION is as follows:

   SELECT column1, column2
   UNION [DISTINCT | ALL]
   SELECT column1, column2
   UNION [DISTINCT | ALL

   Example:

   ```
   select * from First
   UNION
   select * from second
   ```

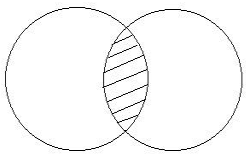
   Union All:

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

2. **INTERSECT operator**

   Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same.

   MySQL does not support INTERSECT operator.

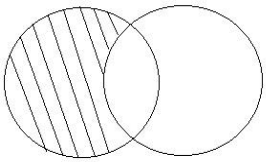
   

   Example:

select * from First
INTERSECT
select * from second

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

   Example:

   select * from First
   MINUS
   select * from second

- **Simple CASE statement**
  Besides the IF statement, MySQL provides an alternative conditional statement called 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. Let's take a look at the syntax of the simple CASE statement:

```
CASE  case_expression
  WHEN when_expression_1 THEN commands
  WHEN when_expression_2 THEN commands
       ...
  ELSE commands
END CASE;
```

  The *case_expression* can be any valid expression. We compare the value of the case_expression with when_expression in each WHEN clause e.g., when_expression_1, when_expression_2, etc. If the value of the case_expression and when_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 the case_expression , the commands in the ELSE clause will execute.

Example: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account set balance =  case
                                when balance <= 10000 then balance *1.05
                                else   balance * 1.06
                              end
```

- **LIKE operator**

The LIKE operator is commonly used to select data based on patterns. Using the LIKE operator in the right way is essential to increase the query performance.

The LIKE operator allows you to select data from a table based on a specified pattern. Therefore, the LIKE operator is often used in the WHERE clause of the SELECT statement.

MySQL provides two wildcard characters for using with the LIKE operator, the percentage % and underscore _.

- The percentage (%) wildcard allows you to match any string of zero or more characters.
- The underscore (_) wildcard allows you to match any single character.

Example: Suppose you want to search for employee whose first name starts with character 'a':

SELECT employeeNumber, lastName, firstName FROM employees WHERE firstName LIKE 'a%';

employeeNumber	lastName	firstName
1143	Bow	Anthony
1611	Fixter	Andy

- **BETWEEN Operator**

The BETWEEN operator allows you to specify a range to test. We often use the BETWEEN operator in the WHERE clause of the SELECT, INSERT, UPDATE, and DELETE statements.

The following illustrates the syntax of the BETWEEN operator:

```
expr [NOT] BETWEEN begin_expr AND end_expr;
```

Example: Suppose you want to find products whose buy prices are within the ranges of $90 and $100

SELECT productCode, productName, buyPrice

FROM products

WHERE buyPrice BETWEEN 90 AND 100;

- **OPERATORS**
  1. **MySQL Comparison Functions and Operators:**
     Some Comparison Functions and Operators are –
     - Between and
     - Equal operator(=)
     - Greater than or equal operator(>=)
     - Greater than operator(>)
     - GREATEST()
     - IN()
     - INTERVAL()
     - IS NOT NULL
     - IS NOT
     - IS NULL
     - IS
     - LESS THAN OR EQUAL OPERATOR(<=)
     - LESS THAN OPERATOR(<)
     - LIKE
     - NOT EQUAL OPERATOR(<>, !=)

  2. **MySQL Logical Operators:**
     MySQL logical AND operator compares two expressions and returns true if both of the expressions are true.

     Some Logical Operators are –
     - And operator
     - Not operator
     - Or operator
     - Xor operator

     MySQL logical AND operator compares two expressions and returns true if both of the expressions are true. Some Logical Operators are - And operator Not operator Or operator Xor operator

Conclusion	In this assignment we are able to: 1. Understand and execute various DML statements. 2. Understand aggregation commands like sum, avg, etc 3. Understand various operators, clauses and set operators in SQL.

Questions	1. What are aggregate Commands in SQL 2. When do we make use of having Clause. 3. hats the difference between where and having clause. 4. 4.What are the clauses used in SQL 5. What is the use of CASE 6. Difference between union and union all. 7. What are set operators

Assignment Number – A3	
Title & Problem Statement	**SQL Queries** - all types of Joins, Sub-Query and View: Write at least 10 SQL queries for suitable database application using SQL DML statements
Objectives	To study SQL Queries all types of joins, sub-queries and view
Outcomes	Students will be able to: 1. To understand different types of joins in MySql 2. To understand the use of nested sub queries 3. To understand creation and use of View.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MySQL/SQL Server/Oracle

Theory

SQL JOINS

One of the most important features of SQL is the ability to define relationships between multiple tables and draw information from them in terms of these relationships, all within a single command. With joins, the information from any number of tables can be accessed. Joins are used to combine columns from different tables. The connection between tables is established through the WHERE clause.

Joins: The ability of relational 'join' operator is an important feature of relational systems. A join makes it possible to select data from more than table by means of a single statement. This joining of tables may be done in a many ways.

Types of JOIN:

Inner Join: Also known as equi join.

Statements generally compare two columns from two columns with the equivalence operator =. This type of join can be used in situations where selecting only those rows that have values in common in the columns specified in the ON clause, is required.

Syntax:

(ANSI style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1> INNER JOIN <tablename2>ON <tablename1>.<columnname> = <tablename2>.<columnname> WHERE <condition> ORDER BY <columnname1>;

(Theta style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1>, <tablename2> WHERE <tablename1>.<columnname> = <tablename2>.<columnname> AND <condition> ORDER BY <columnname1>;

Outer Join
Outer joins are similar to inner joins, but give a little bit more flexibility when selecting data from related tables. This type of joins can be used in situations where it is desired, to select all rows from the table on left (or right, or both) regardless of whether the other table has values in common & (usually) enter NULL where data is missing.

Syntax:

(ANSI style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1> LEFT OUTER JOIN <tablename2>ON <tablename1>.<columnname> = <tablename2>.<columnname> WHERE <condition> ORDER BY <columnname1>;

(Theta style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1>, <tablename2> WHERE <tablename1>.<columnname> = <tablename2>.<columnname> AND <condition> ORDER BY <columnname1>;

Right outer Join
List the employee details with contact details (if any using right outer join. Since the RIGHT JOIN returns all the rows from the second table even if there are no matches in the first table.

Syntax:

(ANSI style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1> RIGHT OUTER JOIN <tablename2>ON <tablename1>.<columnname> = <tablename2>.<columnname> WHERE <condition> ORDER BY <columnname1>;

(Theta style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1>, <tablename2> WHERE <tablename1>.<columnname> = <tablename2>.<columnname> AND <condition> ORDER BY <columnname1>;

Cross Join
A cross join returns what known as a Cartesian product. This means that the join combines every row from the left table with every row in the right table. As can be imagined, sometimes this join produces a mess, but under the right circumstances, it can be very useful. This type of join can be used in situation where it is desired, to select all possible combinations of rows & columns

from both tables. The kind of join is usually not preferred as it may run for a very long time & produce a huge result set that may not be useful.

Syntax:

(ANSI style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1> CROSS JOIN <tablename2>ON <tablename1>.<columnname> = <tablename2>.<columnname> ;

(Theta style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1>, <tablename2>;

Self Join
In some situation, it is necessary to join to itself, as though joining 2 separate tables.
This is referred to as self join

Syntax:
(ANSI style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1> INNER JOIN <tablename1>ON <tablename1>.<columnname> = <tablename2>.<columnname> ;

(Theta style)
SELECT<columnname1>, <columnname2><columnNameN> FROM <tablename1>, <tablename2> WHERE <tablename1>. <columnname> = <tablename1>. <columnname>;

Cartesian Product

When no join condition clause is specified in WHERE clause, each row of one table matches every row of the other table. This results in a Cartesian Product.

Select cz.price_list_id,cz.customer_name,ra.party_id,ra.customer_name
from cz_imp_customer cz , ra_customer ra;

Nested Sub Queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow :-

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns. · An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.

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

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

Example

```
SELECT *
 FROM CUSTOMERS
 WHERE ID IN (SELECT ID
 FROM CUSTOMERS
 WHERE SALARY > 4500);
```

As we have used IN, we can even use NOT IN to exclude the records that obey the given condition.

```
SELECT NAME FROM EMPLOYEES
WHERE ID NOT IN (SELECT NAME FROM EMPLOYEES WHERE
DEPT='MANUFACTURING');
```

This query will give you all the Employees except the Employees working in Manufacturing Department.

There is another operator all that can be used with Nested Subqueries.

ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list or results from a query. The ALL must be preceded by the comparison operators and evaluated to TRUE if the query returns no rows. For example, ALL means greater than every value, means greater than the maximum value. Suppose ALL (1, 2, 3) means greater than 3.

Also, we can use ANY operator. The ANY operator returns true if any of the subquery values meet the condition.

Syntax for ALL:

SELECT [column_name... | expression1]
FROM [table_name]
WHERE expression2 comparison_operator {ALL | ANY | SOME} (subquery)

Syntax for ANY:

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ANY
(SELECT *column_name* FROM *table_name* WHERE *condition*);

EXAMPLE OF ANY:

SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

>all	Greater than all
>=all	Greater than or equal to all
<all	Less than all
<=all	Less than or equal to all
= all	Equal to all / equal to the greatest value

Table 1. Meaning of different variations of all operator

Conclusion	In this assignment we are able to: 1. Understand different types of joins in MySql. 2. Understand and implement Subqueries 3. Implement the complex Views

Questions	1. What is SQL JOINS? 2. Explain different types of JOINS in SQL 3. What is the importance of SQL JOINS in database management? 4. Is SELF JOIN an INNER JOIN or OUTER JOIN? 5. What is the difference between FULL JOIN and CARTESIAN JOIN? 6. What are nested Subqueries? 7. Explain different operators used for subquery. 8. What is the difference between a simple and complex view?

Assignment Number – A4	
Title & Problem Statement	**Unnamed PL/SQL code block**: Use of Control structure and Exception handling is mandatory. Suggested problem statement: Consider Tables: 1. Borrower(Rollno, Name, DateofIssue, NameofBook, Status) 2. Fine(Roll_no, Date, Amt) a. Accept roll_no & name of book from user. b. Check the number of days (from date of issue) c. If days are between 15 to 30 then the fine amount will be Rs 5 per day. d. If no. of days>30, per day fine will be Rs 50 per day & for days less than 30, Rs. 5 per day. e. After submitting the book, status will be changed from I to R. f. If the condition of fine is true, then details will be stored into a fine table. g. Also handle the exception by named exception handler or user defined exception handler.
Objectives	1. To study PL/SQL Block Writing
Outcomes	Students will be able to: 1. To understand the use of Control structure and Exception handling
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- SQL server
Theory	

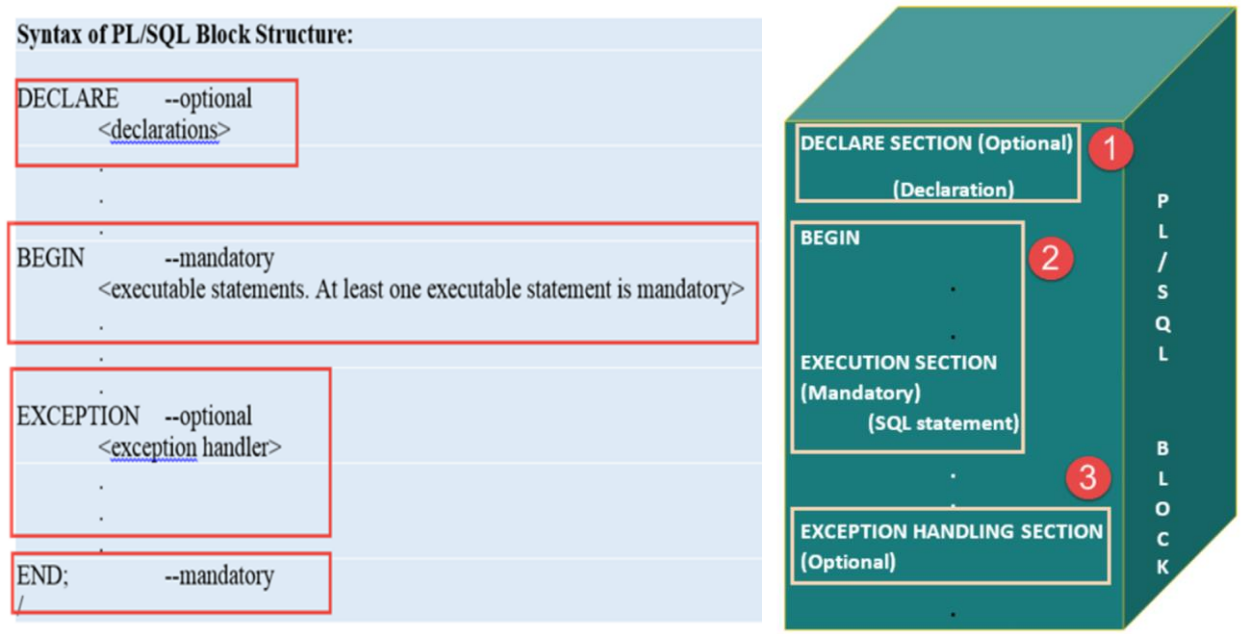
What is PL/SQL block?
PL/SQL is the procedural approach to SQL in which a direct instruction is given to the PL/SQL engine about how to perform actions like storing/fetching/processing data. These instructions are grouped together called Blocks.

Blocks contain both PL/SQL as well as SQL instruction. All these instructions will be executed as a whole rather than executing a single instruction at a time.

Block Structure
PL/SQL blocks have a predefined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks

- Declaration section
- Execution section
- Exception-Handling section

The below picture illustrates the different PL/SQL blocks, section order and the syntax —



Declaration Section
This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are a few more characteristics of this part.

- This particular section is optional and can be skipped if no declarations are needed.
- This should be the first section in a PL/SQL block, if present.
- This section starts with the keyword 'DECLARE' for triggers and anonymous block. For other subprograms this keyword will not be present, instead the part after the subprogram name definition marks the declaration section.
- This section should always be followed by the execution section.

Execution Section

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Below are a few more characteristics of this part.

- This can contain both PL/SQL code and SQL code.
- This can contain one or many blocks inside it as nested blocks.
- This section starts with the keyword 'BEGIN'.
- This section should be followed either by 'END' or Exception-Handling section (if present)

Exception-Handling Section:

The exceptions are unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

Below are a few more characteristics of this part.

- This is the section where the exception raised in the execution block is handled.
- This section is the last part of the PL/SQL block.
- Control from this section can never return to the execution block.
- This section starts with the keyword 'EXCEPTION'.
- This section should be always followed by the keyword 'END'.

The Keyword 'END' marks the end of the PL/SQL block.

There are two types of exceptions –

1. System-defined exceptions
2. User-defined exceptions

A list of predefined exceptions in PL/SQL:

- DUP_VAL_ON_INDEX
- ZERO_DIVIDE,
- NO_DATA_FOUND
- TOO_MANY_ROWS
- CURSOR_ALREADY_OPEN
- INVALID_NUMBER
- INVALID_CURSOR
- PROGRAM_ERROR
- TIMEOUT _ON_RESOURCE
- STORAGE_ERROR
- LOGON_DENIED
- VALUE_ERROR, etc.

Syntax for Exception Handling:

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using **WHEN others THEN** –

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
      exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

Example: Table customer(id, Name,address, salary)

```
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

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

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION** block.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Here is the syntax for raising an exception –>

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

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using a RAISE statement as follows –

```
DECLARE
   my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name customerS.Name%type;
   c_addr customers.address%type;
   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
      FROM customers
      WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;

EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

Output:

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

Types of PL/SQL block

PL/SQL blocks are of mainly two types.

- Anonymous blocks
- Named Blocks

Anonymous blocks:

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as a database object.

Since they need not be stored in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

Below are a few more characteristics of Anonymous blocks.

- These blocks don't have any reference name specified for them.
- These blocks start with the keyword 'DECLARE' or 'BEGIN'.
- Since these blocks are not having any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
- They can call the other named blocks, but calling an anonymous block is not possible as it is not having any reference.
- It can have nested blocks in it which can be named or anonymous. It can also be nested to any blocks.
- These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

Named blocks:

Named blocks have a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present in the server. The compilation process for named blocks happens separately while creating them as database objects.

Below are a few more characteristics of Named blocks.

- These blocks can be called from other blocks.
- The block structure is the same as an anonymous block, except it will never start with the keyword 'DECLARE'. Instead, it will start with the keyword 'CREATE' which instructs the compiler to create it as a database object.
- These blocks can be nested within other blocks. It can also contain nested blocks.

Named blocks are basically of two types:

1. Procedure 2. Function

Conclusion	In this assignment we are able to: 1. Understand PL/SQL blocks and its types, different sections of blocks and their usages. 2. Perform Exception Handling

Questions	1. What is PL/SQL? 2. What are the features of PL/SQL? 3. State difference between SQL and PL/SQL. 4. List out any four statements of PL/SQL. 5. Explain PL/SQL block structure. 6. Explain types of PL/SQL blocks. 7. What are PL/SQL exceptions? 8. How do you declare a user-defined exception? 9. Name some predefined exceptions in PL/SQL? 10. What is the difference between Error and Exception?

Assignment Number – A5	
Title & Problem Statement	**PL/SQL Stored Procedure and Stored Function**. Create the following tables. • Stud_Marks(name, total_marks) • Result(Roll,Name, Class) Write a Stored Procedure namely proc_Grade for the categorization of students. If marks scored by a student in examination is <=1500 and marks>=990 then a student will be placed in distinction category if marks scored are between 989 and 900 category is first class, if marks 899 and 825 category is Higher Second Class otherwise fail. 1. Write a PL/SQL block to use the procedure created with the above requirement. Insert the data in both the tables by calling the above procedure. 2. Write a function which will return the total students in a given class.
Objectives	1. To understand the concept of stored Procedure and function. 2. To study syntax of creating and calling the stored procedure. 3. To create Function and different ways of calling it
Outcomes	Students will be able to: 1. Understand the concept of stored Procedure and function. 2. Understand syntax of creating and calling the stored procedure. 3. How to create Function and different ways of calling it
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database - SQL server

Theory

Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Terminologies in PL/SQL Subprograms

Before we learn about PL/SQL subprograms, we will discuss the various terminologies that are part of these subprograms. Below are the terminologies that we are going to discuss.

Parameter:

The parameter is a variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchanges the values with the main code. This parameter allows input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included in the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and in the calling statement should be the same.
- The size of the data type should not be mentioned at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as

IN Parameter:

- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the sub programs, their values cannot be changed inside the subprogram.
- In the calling statement these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5*8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

OUT Parameter:

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

IN OUT Parameter:

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter types should be mentioned at the time of creating the subprograms.

RETURN

RETURN is the keyword that actually instructs the compiler to switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keyword in the subprogram, the code after this will be skipped.

Normally, parent or main block will call the subprograms, and then the control will shift from those parent blocks to the called subprograms. RETURN in the subprogram will return the control back to their parent block. In the case of functions RETURN statement also returns the value. The

datatype of this value is always mentioned at the time of function declaration. The data type can be of any valid PL/SQL data type.

Procedure

Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object. Below are the characteristics of this subprogram unit.

Note: Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

- Procedures are standalone blocks of a program that can be stored in the database.
- Call to these procedures can be made by referring to their name, to execute the PL/SQL statements.
- It is mainly used to execute a process in PL/SQL.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains the declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the procedure or fetched from the procedure through parameters.
- These parameters should be included in the calling statement.

• Procedures cannot be called directly from SELECT statements; they can be called from another block or through EXEC keyword.

Syntax:

```
        DELIMITER //
CREATE PROCEDURE Procedure_Name (IN|OUT|INOUT variable name data type )
        BEGIN
        Statements 1;
        Statements 2:
        ……..
        END //
        DELIMITER  ;
        call procedure_name();
```

Function

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions has a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

- Functions are a standalone block that is mainly used for the calculation purpose.

- Function uses RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
- Function should either return a value or raise the exception, i.e. return is mandatory in functions.
- Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains the declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the function or fetched from the procedure through the parameters.
- These parameters should be included in the calling statement.
- Function can also return the value through OUT parameters other than using RETURN.
- Since it will always return the value, in the calling statement it always accompanies with the assignment operator to populate the variables.

```
CREATE FUNCTION function_name(param1,param2,…)
  RETURNS datatype
 BEGIN
      Statements
  END
```

Syntax Explanation:

- CREATE FUNCTION instructs the compiler to create a new function.
- Function name should be unique.
- RETURN datatype should be mentioned.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Similarities between Procedure and Function

- Both can be called from other PL/SQL blocks.
- If the exception raised in the subprogram is not handled in the subprogram exception handling section, then it will propagate to the calling block.
- Both can have as many parameters as required.
- Both are treated as database objects in PL/SQL.

Conclusion	In this assignment we are able to: 1. Create and use PL/SQL Functions and Procedures
Questions	1. What is Stored Procedures? 2. What is a function? 3. What are different parameters that can be passed in function and procedure? 4. What is the IN-OUT Parameter? 5. What is the OUT parameter? 6. What are the similarities between Procedure and Function? 7. What is the difference between Procedure and Function? **8.** Can a function is called inside a stored procedure and vice-versa?

Assignment Number – A6	
Title & Problem Statement	Write a PL/SQL block to create a cursor to copy contents of one table into another. Avoid redundancy. Cursors: (All types: Implicit, Explicit, Cursor FOR Loop, Parameterized Cursor) Write a PL/SQL block of code using parameterized Cursor, that will merge the data available in the newly created table N_RollCall with the data available in the table O_RollCall. If the data in the first table already exists in the second table then that data should be skipped.
Objectives	1. To learn and understand PL/SQL in Oracle. 2. To learn and understand cursors.
Outcomes	Students will be able to: 1. Implement the PL/SQL blocks in Oracle. 2. Understand and implement different types of cursors.
S/W Requirement	1. Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more. 2. Windows 7 Operating System, Oracle 11g, SQL developer

Theory

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

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

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

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

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

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

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

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

Example

We will be using the CUSTOMERS table,

```
Select * from customers;

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

The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected −

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

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

Explicit Cursors

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

The syntax for creating an explicit cursor is −

```
Select * from customers;

+____+_____+_____+_____+_____+
| ID | NAME  | AGE | ADDRESS | SALARY |
+____+_____+_____+_____+_____+
| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |
| 2 | Khilan    | 25 | Delhi      | 2000.00 |
| 3 | kaushik   | 23 | Kota       | 2500.00 |
| 4 | Chaitali | 25 | Mumbai    | 7000.00 |
| 5 | Hardik    | 27 | Bhopal    |  9000.00 |
| 6 | Komal     | 22 | MP         |  5000.00 |
```

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

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

Declaring the Cursor

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

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

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

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the

above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
  c_id customers.id%type;
  c_name customerS.No.ame%type;
  c_addr customers.address%type;
  CURSOR c_customers is
     SELECT id, name, address FROM
customers; BEGIN
  OPEN c_customers;
  LOOP
       FETCH c_customers into c_id, c_name,
    c_addr; EXIT WHEN
    c_customers%notfound;
       dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
   CLOSE
c_customers; END;
```

Conclusion	In this assignment we are able to: 1. Implement the PL/SQL block in Oracle 2. Understand and implement Cursor and its different types in Oracle.
Questions	1. What is PL/SQL block? 2. What is Cursor? Explain its different types 3. What is an Implicit Cursor? 4. What is % found ? 5. What is an Explicit Cursor? 6. What is %ROWCOUNT? 7. Explain the steps to implement Cursor. 8. Explain the difference between Implicit and Explicit Cursor. 9. What is the importance of %TYPE and %ROWTYPE data types in PL/SQL?

Assignment Number –A7	
Title & Problem Statement	**Design the Database Trigger. For All Types Row level and Statement level Triggers, before and After Triggers:** Write a database trigger on the Library table. The System should keep track of the records that are being updated or deleted. The old value of updated or deleted records should be added in the Library_Audit table.
Objectives	1. To understand the concept of Events & Triggers. 2. To study the syntax of triggers. 3. Understand the difference between trigger and procedure
Outcomes	Students will be able to: 1. Understand what are triggers & the usage of triggers in the database. 2. Write triggers on table with insert, update & delete queries.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- Oracle/ MySQL

Theory

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers :

Triggers can be written for the following purposes

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Types of Triggers in Oracle :

1. Row-Level Triggers
Row-level triggers execute once for each row in a transaction. Row-level triggers are created using the for each row clause in the create trigger command. For instance if we insert in a single transaction 20 rows to the table EMPLOYEE, the trigger is executed 20 times.

2. Statement-Level Triggers
Statement-level triggers execute once for each transaction. When we insert in one transaction 20 rows to the EMPLOYEE table , then the statement-level trigger is executed only once.

Syntax for Oracle:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
DECLARE
 Declaration-statements
BEGIN
 Executable-statements
EXCEPTION
 Exception-handling-statements
END;
```

Where,

- **CREATE [OR REPLACE] TRIGGER trigger_name** − Creates or replaces an existing trigger with the trigger_name.
- **{BEFORE | AFTER }** − This specifies when the trigger will be executed.
- **{INSERT [OR] | UPDATE [OR] | DELETE}** − This specifies the DML operation.
- **[OF col_name]** − This specifies the column name that will be updated.
- **[ON table_name]** − This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n]** − This allows you to refer to new and old

values for various DML statements, such as INSERT, UPDATE, and DELETE.

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

We can define a maximum of six triggers for each table.

- **BEFORE INSERT** – activated before data is inserted into the table.
- **AFTER INSERT** – activated after data is inserted into the table.
- **BEFORE UPDATE** – activated before data in the table is updated.
- **AFTER UPDATE** – activated after data in the table is updated.
- **BEFORE DELETE** – activated before data is removed from the table.
- **AFTER DELETE** – activated after data is removed from the table.

Example (in MySQL) :

To start with, we will be using the CUSTOMERS table

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
```

The following program creates a **row-level trigger** for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE UPDATE ON customers
FOR EACH ROW
DECLARE
 sal_diff integer;

BEGIN
 sal_diff := NEW.salary - OLD.salary;
```

```
 select sal_diff;
END;
/
```

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

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any UPDATE operation on the table.

Conclusion	In this assignment we are able to: 1. Understand the events and triggers and their different types. 2. Understand how to create triggers with DML statements.
Questions	1. What is Trigger? 2. What are the different types of triggers? 3. What is the use of For Each row? 4. Explain NEW and OLD Keywords in trigger? 5. What is the difference between stored procedure and trigger?

Assignment Number –A8	
Title & Problem Statement	**MySQL - Database Connectivity:** Write a program to implement MySQL/Oracle database connectivity with any front end language to implement Database navigation operations (add, delete, edit etc.)
Objectives	1. To understand and establish connection of the database with the front end.
Outcomes	Students will be able to: 1. Understand database connectivity with front end language. 2. Perform database operations link insert, delete, update etc. by using front end language.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MySQL Programming Language- Python

Theory

Server handles Query and transaction functionality related to SQL processing while Client handles User interface programs and application programs.

In Python, We can use the following modules to communicate with MySQL:

- MySQL Connector Python
- PyMySQL
- MySQLDB
- MySqlClient
- OurSQL

You can choose any of the above modules as per your requirements. The way of accessing the MySQL database remains the same.
Mostly used any of the following two modules:-

1. MySQL Connector Python
2. PyMySQL

How to connect MySQL database in Python:
Let's see how to connect the MySQL database in Python using the 'MySQL Connector Python module.

Arguments required to connect:
You need to know the following details of the MySQL server to perform the connection from Python.

Argument	Description
Username	The username that you use to work with MySQL Server. The default username for the MySQL database is a **root**.
Password	Password is given by the user at the time of installing the MySQL server. If you are using root then you won't need the password.
Host name	The server name or Ip address on which MySQL is running. if you are running on localhost, then you can use **localhost** or its IP 127.0.0.0
Database name	The name of the database to which you want to connect and perform the operations.

How to Connect to MySQL Database in Python:

1. **Install MySQL connector module**

   Use the pip command to install MySQL connector Python.
   pip install mysql-connector-python

2. **Import MySQL connector module**

   Import using an import **mysql.connector** statement so you can use this module's methods to communicate with the MySQL database.

3. **Use the connect() method**

   Use the **connect()** method of the MySQL Connector class with the required arguments to connect MySQL. It would return a MySQLConnection object if the connection established successfully

4. **Use the cursor() method**

   Use the **cursor()** method of a MySQLConnection object to create a cursor object to perform various SQL operations.

5. **Use the execute() method**

The **execute()** methods run the SQL query and return the result.

6. **Extract result using fetchall()**

Use **cursor.fetchall()** or **fetchone()** or **fetchmany()** to read query results.

7. **Close cursor and connection objects**

use **cursor.close()** and **connection.close()** methods to close open connections after your work completes.

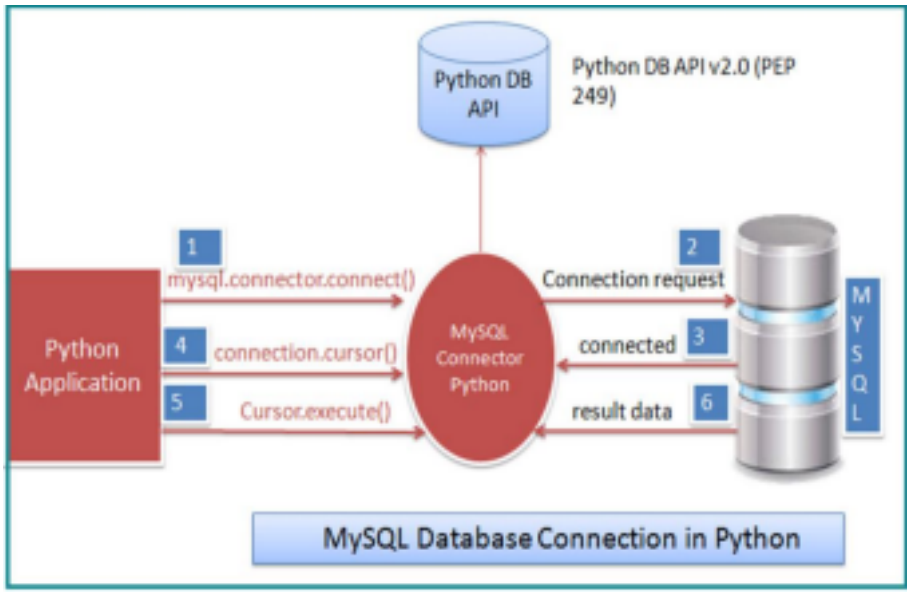


Figure 1: MySQL database connection in Python

Example to connect to MySQL Database in Python :

```
import mysql.connector
from mysql.connector import Error

try:
        connection = mysql.connector.connect(host='localhost',
                                             database='Electronics',
                                             user='pynative',  password='pynative@#29')
```

```
    if connection.is_connected():
        db_Info = connection.get_server_info()
        print("Connected to MySQL Server version ", db_Info)
        cursor = connection.cursor()
        cursor.execute("select database();")
        record = cursor.fetchone()
        print("You're connected to database: ", record)

except Error as e:
    print("Error while connecting to MySQL", e)

finally:
    if connection.is_connected():
        cursor.close()
        connection.close()
        print("MySQL connection is closed")
```

Conclusion	In this assignment we are able to: 1. Understand the database connectivity between Python and Mysql. 2. Understand the steps in establishing the database connectivity to perform database operations like insert, delete, update etc.
Questions	1. What is 2 tier & 3 tier Architecture application. 2. What are the parameters required to set the connection? 3. What are connection and statement objects used for database connectivity. 4. How we can you use Python with MySQL 5. Explain how to connect to MySQL using Python? 6. How to Extract result? 7. Difference between fetchall() or fetchone() or fetchmany(). 8. What is the use of the cursor() method? 9. How to run SQL query using python?

Assignment Number – B1	
Title & Problem Statement	**Design and Develop MongoDB Queries using CRUD operations. (Use CRUD operations, SAVE method, logical operators.** Create an Articles Collection and perform basic operations (CRUD) like Insert, Display, Update and Delete operations. Eg. Articles(Tittle, Content, Author, Author_age, Dop, Category, Comment). Comment is an embedded document and also an array. Comment may consist of the Name, Remarks keys.
Objectives	1. To understand Basic CRUD operations. 2. To understand the use of the SAVE method. 3. To understand logical operators $and, $not and $or.
Outcomes	Students will be able to: 1. Implement a **CRUD** with suitable example using MongoDB
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MongoDB

Theory

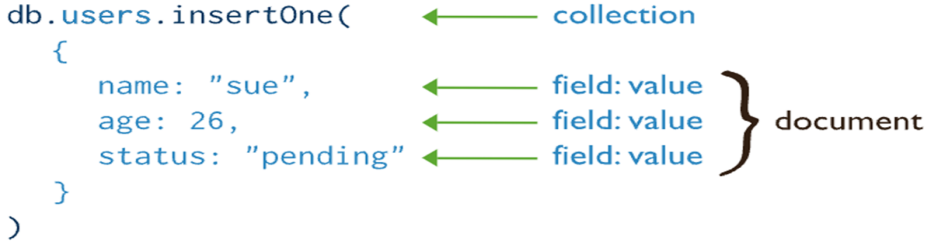
Mongodb is a document-oriented database program widely classified as a NoSQL database program.

In Mongodb the CRUD operation refers to the creating, reading, updating, and deleting of documents.

Here is an explanation of the operations in detail:

1. **CREATE**
   Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection. MongoDB provides the following methods to insert documents into a collection:
   - db.collection.insertOne() *New in version 3.2*
   - db.collection.insertMany() *New in version 3.2*

   In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.



- **db.collection_name.insertOne()** inserts a *single* document into a collection and returns a document that includes the newly inserted document's `_id` field value.

  Example:

  db.inventory.insertOne(

  { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }

)

- **db.collection_name.insertMany()** can insert *multiple* documents into a collection. Pass an array of documents to the method. It returns a document that includes the newly inserted documents `_id` field values.

  Example:

  db.inventory.insertMany([

  { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },

  { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },

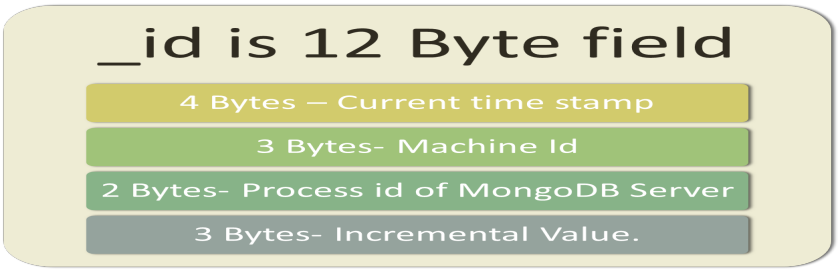
  { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }

  }

  ])

  where, _id Field:

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



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

SAVE method:

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

Syntax: db.collection.save()

Parameters:

Name	Description	Required / Optional
document	A document to save to the collection.	Required
writeConcern	A document expressing the write concern. Omit to use the default write concern.	Optional

Returns: A WriteResult object that contains the status of the operation.

Example: Save a New Document without Specifying an _id Field
In the following example, save()method performs an insert since the document passed to the method does not contain the _id field:

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

Output:

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

WriteResult({ "nInserted" : 1 })
```

During the insert, the shell will create the _id field with a unique ObjectId value, as verified by the inserted document:

Example: Replace an Existing Document

The products collection contains the following document:

```
{ "_id" : 1001, "inv_no" : "I00001", "inv_date" : "10/10/2012", "ord_qty" : 200 }
```

The save() method performs an update with upsert: true since the document contains an _id field

```
db.invoice.save( {  _id: 1001,inv_no: "I00015", inv_date: "15/10/2012", ord_qty:500 } );
```

2. READ

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

db.collection_name.find()

You can specify query filters or criteria that identify the documents to return.

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

Syntax: **db.collection_name.find().pretty()**

You can specify query filters or criteria that identify the documents to return.

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

- **Specify Equality Condition**

  To specify equality conditions, use <field>:<value> expressions in the query filter document:

  Example:

  db. users. find({name: "Jiya"})

Comparison Operators:

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

db.collection_name.findOne() - Find First document Only.

3. UPDATE

Update operations modify existing documents in a collection.

Syntax:

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

MongoDB provides the following methods to update documents of a collection:

- `db.collection.updateOne()` *New in version 3.2*
- `db.collection.updateMany()` *New in version 3.2*
- `db.collection.replaceOne()` *New in version 3.2*

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document. You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

```
db.users.updateMany(                          <-- collection
   { age: { $lt: 18 } },                      <-- update filter
   { $set: { status: "reject" } }             <-- update action
)
```

4. DELETE

Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

- `db.collection.deleteOne()` *New in version 3.2*
- `db.collection.deleteMany()` *New in version 3.2*

To delete multiple documents, use db.collection.deleteMany().

To delete a single document, use db.collection.deleteOne().

Delete All Documents:

To delete all documents from a collection, pass an empty filter document {} to the db.collection_name.deleteMany() method as follows:

db.users.deleteMany({})

Delete All Documents that Match a Condition:

You can specify criteria, or filters, that identify the documents to delete.To specify equality conditions, use <field>:<value> expressions in the query filter document.

To delete all documents that match a deletion criteria, pass a filter parameter to the deleteMany() method. The Example is as follows:-

Logical Operator - $and

The MongoDB $and operator performs a logical AND operation on an array of two or more expressions and retrieves the documents which satisfy all the expressions in the array. The $and operator uses short-circuit evaluation. If the first expression (e.g. <expression1>) evaluates to false, MongoDB will not evaluate the remaining expressions.

Syntax: { $and: [{ <exp1> }, { <exp2> } , ... , { <expN> }] }

Example of MongoDB Logical Operator - $and

If we want to select all documents from the collection "student" which satisfying the condition -

1. *sex* of student is Female and

2. *class* of the student is VI and

3. *grd_point* of the student is greater than equal to 31

The following mongodb command can be used :

>db.student.find({$and:[{"sex":"Male"},{"grd_point":{ $gte: 31 }},{"class":"VI"}]}).pretty();

Logical Operator - $not

The MongoDB $not operator performs a logical NOT operation on the given expression and fetches selected documents that do not match the expression and the document that do not contain the field as well, specified in the expression.

Syntax: { field: { $not: { <expression> } } }

Example of MongoDB Logical Operator - $not

If we want to select all documents from the collection "student" which satisfying the condition - *age* of the student is at least 12.The following mongodb command can be used :
>db.student.find({"age": { $not: {$lt : 12}}}).pretty();

Example of MongoDB Logical Operator - $not with pattern matching

---

If we want to select all documents from the collection "student" which satisfying the condition - *sex* of student must not Male. The following mongodb command can be used :
>db.student.find({"sex": { $not: /^M.*/}}).pretty();

Logical operator $or and $nor.

The $or operator is used to search multiple expressions in a single query with only one matching criterion to be needed in a document. More than one keys and values can be used with the $or operator.

MongoDB conditional operator - $or example

If we want to fetch documents from the collection "testtable" which containing the value of "age " either 19 or 22 or 23, the following mongodb command can be used :
>db.testtable.find({$or : [{"age" : 19},{"age" : 22},{"age":23}]}

MongoDB $or operator with another field

If we want to fetch documents from the collection "testtable" which containing the value of "date_of_join" is "16/10/2010" and the value of "age " either 19 or 22 or 23, the following mongodb command can be used :

>db.testtable.find({ "date_of_join" : "16/10/2010" , $or : [{"age" : 19},{"age" : 22},{"age":23}] })

The $nor operator is used to search multiple expressions in a single query which does not match any of the values specified with the $nor.

MongoDB $nor (not or) operator

If we want to fetch documents from the collection "test table" which containing the value of "date_of_join" is "16/10/2010" and not containing the value of "age " either 19 or 22 or 23, the following mongodb command can be used :

>db.testtable.find({ "date_of_join" : "16/10/2010" , $nor : [{"age" : 19},{"age" :22},{"age":23}] })

Conclusion	In this assignment we are able to: 1. Understand the CRUD operations in MongoDB.
Questions	1. How to Display the first document found in db? 2. How to Modify the document in db? 3. How to Insert record with save method with and without objectID? 4. What is the difference between the INSERT and SAVE method? 5. What is _id Field? 6. What 12 bytes of ObjectID consist of? 7. How to Remove All Documents from a Collection? 8. How to Delete the collection? 9. How to Create a Database in MongoDB? 10. What is Document in MongoDB? 11. Difference between MySQL and MongoDB.

Assignment Number – B2	
Title & Problem Statement	**Implement aggregation and indexing with suitable example using MongoDB** Create an orders collection with keys order_id, cust_id, cust_name, phone_no(array field), email_id(optional field), item_name, DtOfOrder, quantity, amount, status(P :pending/D:delivered)
Objectives	1. To study indexing in mongoDb 2. To study aggregation operation in MongoDb and to use various pipeline operators such as $project, $group, $match etc
Outcomes	Students will be able to: 1. Implement aggregation and indexing with suitable example using MongoDB
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MongoDB

Theory

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 require mongod 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.

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

_id Index

The _id index is a unique index on the _id field, and MongoDB creates this index by default on all collections. You cannot delete the index on _id.

You can create indexes on any field within any document or subdocument. Additionally, you can create compound indexes with multiple fields, so that a single query can match multiple components using the index while scanning fewer whole documents.

In general, you should create indexes that support your primary, common, and user-facing queries. Doing so requires MongoDB to scan the fewest number of documents possible.

In the mongo shell, you can create an index by calling the **createIndex()** method.

Arguments to ensureIndex() resemble the following:

```
{ "field": 1 }
{ "product.quantity": 1 }
{ "product": 1, "quantity": 1 }
```

For each field in the index specify either 1 for an ascending order or -1 for a descending order, which represents the order of the keys in the index. For indexes with more than one key (i.e. compound indexes) the sequence of fields is important.

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field. To create a unique index on the user_id field of the members collection, use the following operation in the mongo

```
db.addresses.ensureIndex( { "user_id": 1 }, { unique: true } )
```

By default, unique is false on MongoDB indexes.

If you use the unique constraint on a compound index then MongoDB will enforce uniqueness on the combination of values, rather than the individual value for any or all values of the key. If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document. MongoDB will only permit one document without a unique value in the collection because of this unique constraint. You can combine with the sparse index to filter these null values from the unique index. You may not specify a unique constraint on a hashed index.

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field. Any document that is missing the field is not indexed. The index is "sparse" because of the missing documents when values are missing. By contrast, non-sparse indexes contain all documents in a collection, and store null values for documents that do not contain the indexed field. Create a sparse index on the xmpp_id field, of the members collection, using the following operation in the mongo shell:

db.addresses.ensureIndex({ "xmpp_id": 1 }, { sparse: **true** })

By default, sparse is false on MongoDB indexes.

Multikey Indexes

If you index a field that contains an array, MongoDB indexes each value in the array separately, in a "multikey index."

Given the following document:

{ "_id" : ObjectId("..."),

"name" : "Warm Weather",

"author" : "Steve",

"tags" : ["weather", "hot", "record", "april"] }

Then an index on the tags field would be a multikey index and would include these separate entries:

{ tags: "weather" }

{ tags: "hot" }

{ tags: "record" }

{ tags: "april" }

Queries could use the multikey index to return queries for any of the above values.

1. **Explain() Before Indexing:**

> db.users_million.find({"username":"user15"}).explain()

```
{
        "cursor" : "BasicCursor",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1116472,
        "nscanned" : 1116472,
```

```
        "nscannedObjectsAllPlans" : 1116472,
        "nscannedAllPlans" : 1116472,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 2,
        "nChunkSkips" : 0,
        "millis" : 3744,
        "indexBounds" : {

        },
        "server" : "hp:27017"
}
```

2. **Apply createIndex now:**

```
>db.users_million.createIndex({"username":1})
```

3. **Now check output for explain() for same query**

```
> db.users_million.find({"username":"user15"}).explain()
{
        "cursor" : "BtreeCursor username_1",
        "isMultiKey" : false,
        "n" : 1,
        "nscannedObjects" : 1,
        "nscanned" : 1,
        "nscannedObjectsAllPlans" : 1,
        "nscannedAllPlans" : 1,
        "scanAndOrder" : false,
        "indexOnly" : false,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "millis" : 0,
        "indexBounds" : {
                "username" : [
                        [
                                "user15",
                                "user15"
                        ]
                ]
        },
        "server" : "hp:27017"
```

$exists

Syntax: { field: { $exists: <boolean> } }

$exists selects the documents that contain the field if <boolean> is true. If <boolean> is false, the query only returns the documents that do not contain the field. Documents that contain the field but has the value null are *not* returned

MongoDB *$exists* does **not** correspond to SQL operator exists. For SQL exists, refer to the $in operator.

Aggregation

Once you have data stored in MongoDB, you may want to do more than just retrieve it; you may want to analyze and crunch it in interesting ways.

MongoDB provides three ways to perform aggregation:

a. **the aggregation pipeline**
b. **the map-reduce function**
c. **single purpose aggregation methods and commands**

The aggregation framework lets you transform and combine documents in a collection. Basically, you build a pipeline that processes a stream of documents through several building blocks: filtering, projecting, grouping, sorting, limiting, and skipping.

For example, if you had a collection of magazine articles, you might want to find out who your most prolific authors were. Assuming that each article is stored as a document in MongoDB, you could create a pipeline with several steps:

1. Project the authors out of each article document.
2. Group the authors by name, counting the number of occurrences.
3. Sort the authors by the occurrence count, descending.
4. Limit results to the first five.

Each of these steps maps to a aggregation framework operator:

1. {"$project" : {"author" : 1}}

This projects the author field in each document. The syntax is similar to the field selector used in querying: you can select fields to project by specifying "*fieldname*" : 1 or exclude fields with "*fieldname*" : 0.

After this operation, each document in the results looks like: {"_id" : *id*, "au

thor" : "*authorName*"}. These resulting documents only exists in memory and are not written to disk anywhere.

2. {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}}
This groups the authors by name and increments "count" for each document an author appears in. First, we specify the field we want to group by, which is "author". This is indicated by the "_id" : "$author" field. You can picture this as: after the group there will be one result document per author, so "author" becomes the unique identifier ("_id").
The second field means to add 1 to a "count" field for each document in the group. Note that the incoming documents do not have a "count" field; this is a new field created by the "$group".
At the end of this step, each document in the results looks like:
{"_id" : "*authorName*", "count" : *articleCount*}.

3. {"$sort" : {"count" : -1}}
This reorders the result documents by the "count" field from greatest to least.

4. {"$limit" : 5}
This limits the result set to the first five result documents. To actually run this in MongoDB, pass each operation to the aggregate() function:

```
> db.articles.aggregate({"$project" : {"author" : 1}},
... {"$group" : {"_id" : "$author", "count" : {"$sum" : 1}}},
... {"$sort" : {"count" : -1}},
... {"$limit" : 5})
```

Pipelining Operators
Operators can be combined in any order and repeated as many times as necessary. For example, you could "$match", "$group", and then "$match" again with different criteria.

$match filters documents so that you can run an aggregation on a subset of documents. For example, if you only want to find out stats about users in Oregon, you might add a "$match" expression such as {$match : {"state" : "Oregon"}}. "$match" can use all of the usual query operators ("$gt", "$lt", "$in", etc.).

$project
Projection is much more powerful in the pipeline than it is in the “normal” query language. "$project" allows you to extract fields from subdocuments, rename fields, and perform interesting operations on them.

Pipeline expressions
There are several expressions available with aggregation which you can combine and nest to any depth to create more complex expressions.

$group
Grouping allows you to group documents based on certain fields and combine their values. Some examples of groupings:
• If we had per-minute measurements and we wanted to find the average humidity per day, we would group by the "day" field.
{"$group" : {"_id" : "$day"}}

$unwind
Unwinding turns each field of an array into a separate document. For example, if we had a blog with comments, we could use unwind to turn each comment into its own “document”:Pipeline.

$sort
You can sort by any field or fields, using the same syntax you would with the “normal” query language. If you are sorting a non-trivial number of documents, it is highly recommended that you do the sort at the beginning of the pipeline and have an index it can use. Otherwise, the sort may be slow and take a lot of memory.

$limit
$limit takes a number, *n*, and returns the first *n* resulting documents.

$skip
$skip takes a number, *n*, and discards the first *n* documents from the result set. As with “normal” querying, it isn’t efficient for large skips, as it must find all of the matches that must be skipped and then discard them.

Mathematical expressions.
Here’s the syntax for each operator:
"$add" : [*expr1[, expr2, ..., exprN]*]
Takes one or more expressions and adds them together.
"$subtract" : [*expr1, expr2*]
Takes two expressions and subtracts the second from the first.
"$multiply" : [*expr1[, expr2, ..., exprN]*]
Takes one or more expressions and multiplies them together.
"$divide" : [*expr1, expr2*] Takes two expressions and divides the first by the second.
"$mod" : [*expr1, expr2*]Takes two expressions and returns the remainder of dividing the first by the second.

```
> db.employees.aggregate(
... {
... "$project" : {
... "totalPay" : {
... "$subtract" : [{"$add" : ["$salary", "$bonus"]}, "$401k"]
... }
... }... })
```

Date expressions: "$year", "$month", "$week",
"$dayOfMonth", "$dayOfWeek", "$dayOfYear", "$hour", "$minute", and "$second".

Conclusion	In this assignment we are able to: 1. Understand the operations of aggregation and indexing in MongoDB.
Questions	1. Difference between Sql and Nosql. 2. Difference between simple index and compound index. 3. What are sparse Indexes? 4. What is a multikey index? 5. Name any five pipeline operators. 6. What are various string expressions and date expressions? 7. What is the use of $project,$match,…etc.

Assignment Number – B3	
Title & Problem Statement	**MongoDB - Map reduces operations:** Create an **orders** collection with fields **customer id, order date, status, price and items, quantity**. Execute the given queries using MapReduce. Perform map reduce operation on orders collection.
Objectives	1. To study MapReduce operation in mongodb to perform Complex Aggregation tasks.
Outcomes	Students will be able to: 1. Understand MapReduce operation in MongoDB. 2. Perform Complex Aggregation task with MapReduce.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Database- MongoDB

Theory

MapReduce is a powerful and flexible tool for aggregating data. It can solve some problems that are too complex to express using the aggregation framework's query language. MapReduce uses JavaScript as its "query language" so it can express arbitrarily complex logic. However, this power comes at a price: MapReduce tends to be fairly slow and should not be used for real-time data analysis. MapReduce 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 the machines are finished, they merge all 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. MapReduce is an incredibly useful and powerful, but also somewhat complex, tool.

Example: Finding All Keys in a Collection
MongoDB assumes that your schema is dynamic, so it does not keep track of the keys in each document. The best way, in general, to find all the keys across all the documents in a collection is to use MapReduce. In this example, we'll also get a count of how many times each key appears in the collection. This example doesn't include keys for embedded documents, but it would be a simple addition to the map function to do so. For the mapping step, we want to get every key of every document in the collection. The map function uses a special function to "return" values that we want to process later: emit. emit gives MapReduce a key (like the one used by group earlier) and a value. Inthis case, we emit a count of how many times a given key appeared in the document

(once: {count : 1}). We want a separate count for each key, so we'll call emit for every key in the document. this is a reference to the current document we are mapping:

```
> map = function() {
... for (var key in this) {
... emit(key, {count : 1});
... }};
```

Now we have a ton of little {count : 1} documents floating around, each associated with a key from the collection. An array of one or more of these {count : 1} documents will be passed to the reduce function. The reduce function is passed two arguments: key, which is the first argument from emit, and an array of one or more {count : 1} documents that were emitted for that key:

```
> reduce = function(key, emits) {
... total = 0;
... for (var i in emits) {
... total += emits[i].count;
... }
... return {"count" : total};
... }
```

reduce must be able to be called repeatedly on results from either the map phase or previous reduce phases. Therefore, reduce must return a document that can be re-sent to reduce as an element of its second argument. For example, say we have the key x mapped to three documents: {count : 1, id : 1}, {count : 1, id : 2}, and {count : 1, id : 3}. (The ID keys are just for identification purposes.) MongoDB might call reduce in the following pattern:

```
> r1 = reduce("x", [{count : 1, id : 1}, {count : 1, id : 2}])
{count : 2}
> r2 = reduce("x", [{count : 1, id : 3}])
{count : 1}
> reduce("x", [r1, r2])
{count : 3}
```

You cannot depend on the second argument always holding one of the initial documents ({count : 1} in this case) or being a certain length. reduce should be able to be run on any combination of emit documents and reduce return values.
Altogether, this MapReduce function would look like this:

```
> mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" : reduce})
{
"result" : "tmp.mr.mapreduce_1266787811_1",
"timeMillis" : 12,
"counts" : {
"input" : 6
"emit" : 14
```

```
"output" : 5
},
"ok" : true
}
```

The document MapReduce returns gives you a bunch of metainformation about the operation:
"result" : "tmp.mr.mapreduce_1266787811_1"
This is the name of the collection the MapReduce results were stored in. This is a temporary collection that will be deleted when the connection that did the Map- Reduce is closed.
"timeMillis" : 12
How long the operation took, in milliseconds.
"counts" : { ... }
This embedded document is mostly used for debugging and contains three keys:
"input" : 6
The number of documents sent to the map function.
"emit" : 14
The number of times emit was called in the map function.
"output" : 5
The number of documents created in the result collection.

If we do a find on the resulting collection, we can see all the keys and their counts from our original collection:

```
> db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

Each of the key values becomes an "_id", and the final result of the reduce step(s) becomes the "value".

Conclusion	In this assignment we are able to: 1. Understand the operations of MapReduce in MongoDB. 2. Understand the working and performance of complex Aggregation tasks in MapReduce.
Questions	1. What is MapReduce? 2. How does MapReduce work in MongoDB? 3. What is the difference between MapReduce and aggregation in MongoDB? 4. What are advantages and drawbacks of map reduce? 5. What is the use of emit? 6. What is map and reduce function in MongoDB?

Assignment Number – B4	
Title & Problem Statement	**Database Connectivity:** Write a program to implement MongoDB database connectivity with any front end language to implement Database navigation operations (add, delete, edit etc.)
Objectives	To study the concept of establishing connection of MongoDB database with front end application.
Outcomes	Students will be able to: 1. Understand Basic Connectivity steps with front end application 2. Establish Database Connectivity with MongoDB 3. Implement basic Database Navigation Operations like add, delete, update,etc.
S/W Requirement	OS – Linux Ubuntu 18 (64 bit) Python, MongoDB Python Package - pymongo

Theory

MongoDB stores data in JSON-like documents, which makes the database very flexible and scalable. To be able to connect front end with MongoDB, you need to access a MongoDB database. For this assignment we are going to use Python as front-end. Python needs a MongoDB driver "PyMongo" to access the MongoDB database.
So, it is recommended to install "PyMongo" package in your system using PIP. PIP is most likely already installed in your Python environment.
Download and install "PyMongo":

How to Connect MongoDB Database in Python

1. First start MongoDB from command prompt using
    a. sudo service mongod start
    b. mongo

2. Import MongoClient
    from pymongo import MongoClient

3. Create a connection
    The very first after importing the module is to create a MongoClient.
    from pymongo import MongoClient
    client = MongoClient()

After this, connect to the default host and port. Connection to the host and port is done explicitly. The following command is used to connect the MongoClient on the localhost which runs on port number 27017.
client = MongoClient('host', port_number)
example: - ***client = MongoClient('localhost', 27017)***

It can also be done using the following command:
client = MongoClient("mongodb://localhost:27017/")

4. Access DataBase Objects

To create a database or switch to an existing database we use:

Method 1: Dictionary-style

mydatabase = client['name_of_the_database']

Method 2:

mydatabase = client.name_of_the_database

If there is no previously created database with this name, MongoDB will implicitly create one for the user.

Note: The name of the database fill won't tolerate any dash (-) used in it. The names like my-Table will raise an error. So, underscore is permitted to use in the name.

5. Accessing the Collection

We access a collection in PyMongo in the same way as we access the Tables in the RDBMS. To access the table, say table name "myTable" of the database, say "mydatabase".

Method 1: Dictionary-style

mycollection = mydatabase['myTable']

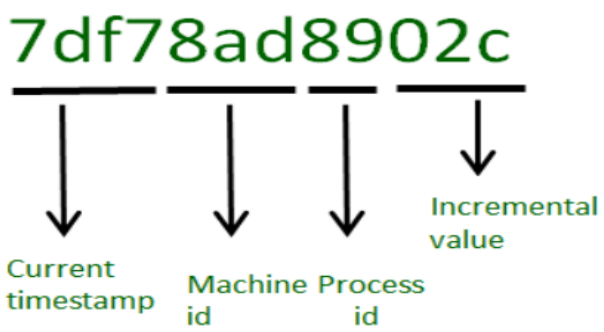
Method 2:

mycollection = mydatabase.myTable

MongoDB store the database in the form of dictionaries as shown:

```
record = {
title: 'MongoDB and Python',
description: 'MongoDB is no SQL database',
tags: ['mongodb', 'database', 'NoSQL'],
viewers: 104
}
```

'_id' is the special key which get automatically added if the programmer forgets to add explicitly. _id is the 12 bytes hexadecimal number which assures the uniqueness of every inserted document.



6. Insert the data inside a collection

   Methods to be used:
   insert_one() or ***insert_many()***

   We normally use insert_one() method document into our collections. Say, we wish to enter the data named as record into the ’myTable’ of ‘mydatabase’.

   rec = mycollection.insert_one(record)

7. Querying in MongoDB

   There are certain query functions which are used to filter the data in the database. The two most commonly used functions are:

   i. **find ()**: used to get more than one single document as a result of query.
   for i in mycollection.find({title: 'MongoDB and Python'})
   print(i)
   This will output all the documents in the myTable of mydatabase whose title is ‘MongoDB and Python’.

   ii. **count()**: used to get the numbers of documents with the name as passed in the parameters.
   print(mycollection.count({title: 'MongoDB and Python'}))

   This will output the numbers of documents in the myTable of mydatabase whose title is ‘MongoDB and Python’.
   To print all the documents/entries inside ‘myTable’ of database ‘mydatabase’:

```
from pymongo import MongoClient

try:
    conn = MongoClient()
    print("Connected successfully!!!")
except:
    print("Could not connect to MongoDB")

# database name: mydatabase
db = conn.mydatabase

# Created or Switched to collection names: myTable
collection = db.myTable

# To find() all the entries inside collection name 'myTable'
cursor = collection.find()
for record in cursor:
    print(record)
```

8. Update the data inside a collection

   You can update a record, or document as it is called in MongoDB, by using the ***update_one()*** and ***update_many()*** method.

   - The first parameter of these methods is a query object defining which document to update.
   - The second parameter is an object defining the new values of the document.

   Using method ***update_one()***: if the query finds more than one record, only the first occurrence is updated.

```
#update all the documents whose title is MSQL
result = mycollection.update_many({"title": "MSQL"},
        { $set: {"title": "MYSQL and Python"},
                {$currentDate:{"lastModified":true}
        }
        )
```

   To find number of documents or entries in the collection are updated use.

```
print(result.matched_count)
```

9. Delete the data inside a collection

   You can delete a document by using methods: ***delete_one()*** and ***delete_many()***.

   To delete one document, we use the ***delete_one()*** method and to delete more than one document, use the ***delete_many()*** method.

	The first parameter of these methods is a query object defining which document to delete. ***#Delete all the documents whose viewers less than 100*** ***deleteFilter = {"viewers": {$lt: 100}}*** ***result = mycollection.delete_many(deleteFilter)*** ***print(result.deleted_count, " documents deleted.")*** To Delete all documents in a collection ***result = mycollection.delete_many({})***
Conclusion	In this assignment we are able to: 1. Understand the basic database connectivity steps using python. 2. Established Python MongoDB database connection. 3. Implemented basic Database Navigation Operations like add, delete, update,etc.
Questions	1. What is the syntax to create a collection and to drop a collection in MongoDB? 2. Mention what is ObjectId composed of? 3. Is Python necessary for MongoDB? 4. How does Python connect to MongoDB? 5. How to insert document in a collection? 6. What are the different parameters used while updating document in collection? 7. How to delete all documents in a collection? 8. What is the significance of count ()? How to use this function?