RWANDA POLYTECHNIC
IPRC NGOMA
Integrated Polytechnic Regional College
P.O. Box 35 KIBUNGO - RWANDA
Tel: +250 783 540 145
Email:info@iprcngoma.rp.ac.rw
www.iprcngoma.rp.ac.rw

INFORMATION AND COMMUNICATION

TECHNOLOGY DEPARTMENT

OPTION: INFORMATION TECHNOLOGY

MODULE CODE: IT205

MODULE NAME: Database Management Systems

LEVEL II

ACADEMIC YEAR 2020/2021

TRAINER: Geoffrey GAKUMBA

February 2021

CONTENTS

CHAPTER I: Overview of database

I.1 INTRODUCTION

Before DBMS was invented, Information was stored using File Processing System. Therefore, data was stored in permanent system files (secondary Storage). Different application programs are written to extract data from these files and to add record to these files.

I.2 File processing system

It is a collection of programs that store and manage files in computer hard disk.

I.3 Drawbacks of using file systems to store data:

Data redundancy and inconsistency

✓ Multiple file formats, duplication of information in different files

Difficulty in accessing data

✓ Need to write a new program to carry out each new task

Data isolation — multiple files and

formats Integrity problems

✓ Integrity constraints (e.g. account balance > 0) become part

of program code

✓ Hard to add new constraints or change existing ones

Concurrent access by multiple users

✓ Concurrent accessed needed for performance

✓ Uncontrolled concurrent accesses can lead to inconsistencies

– E.g. two people reading a balance and updating it at the same time

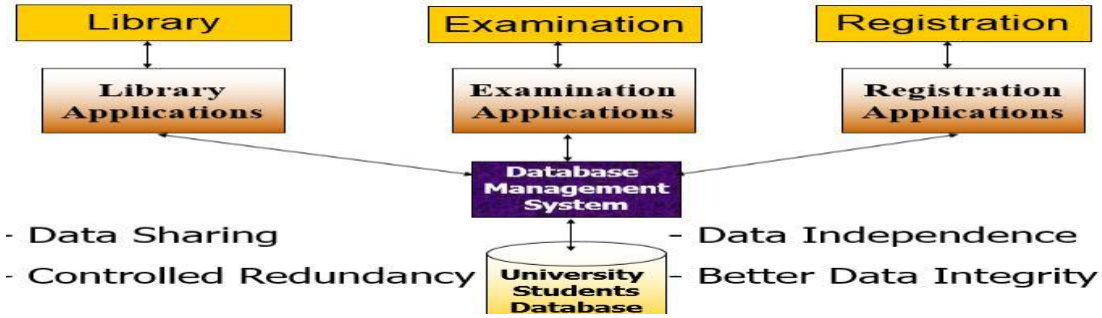
Security problems

I.4 Purpose of Database system

Database and DBMS software together is called Database **system**. In the early days, database applications were built on top of file systems. Database systems offer solutions to all the above problems. Therefore, we need database.

I.4.1 what is database?

A database is a logically coherent collection of data with some inherent meaning, representing some aspect of real world and which is designed, built and populated with data for a specific purpose. It uses a Data Definition Language.



Meta-data

Meta-data is database definition or descriptive information that is stored by the DBMS in the form of a database catalog or data dictionary.

I.4.2 COMPONENTS OF A DATABASE

A database consists of several components. Each component plays an important role in the database system environment. The major components of database are as follows:

a) Data

It is raw numbers, characters or facts represented by value. Most of the organizations generate, store and process large amount of data. The data acts as a bridge between the hardware and the software. Data may be of different types such as user data, metadata and application metadata.

b) Software

A set of programs lies between the stored data and the users of the database. It is used to control and manage the overall computerized database. It uses different types of software such as MYSQL, Oracle etc.

c) Hardware

It is the physical aspect of computer, telecommunication and database, which consists of the secondary storage devices such as magnetic disks, optical discs etc. on which data is stored.

d) Users

It is the person, who needs information from the database to carry out its primary business responsibilities.

The various types of users, which can access the database system, are as follows:

1. **Database Administrator (DBA)**

A person, who is responsible for managing or establishing policies for the maintenance and handling the overall database management system, is called DBA.

2. **Application programmers**

   The people, who write application programs in programming languages to interact and manipulate the database, are called application programmers.

3. **End-user**

   A person, who interacts with the database system to perform different operation on the database like inserting, deleting etc through menus or forms.

I.4.3 FEATURES OF A DATABASE

Features of database to let you manage your data are as follows:

a) Tables

It is the building block of any relational database model, where all the actual data is defined and entered. A database consists of many tables. Tables (relations) consists of cells at the intersection of records (rows) and fields (columns). Different types of operations are done on the tables such as storing, filtering, retrieving and editing of data. It is also known as file.

b) Fields (Data Item)

It is an area (within the record), reserved for a specific piece of data. It is the individual sub-component of one record. It contains set of characters. E.g: customer number, customer name, street address, city, state, phone number, current address etc. Field of table is also known as column or attribute.

c) Record

It is the collection of data items of all the fields (information) pertaining to one entity or a complete unit of information. i.e. a person, company, transition etc. Record of a table is also known as row, entity or tuple.

d) Queries

It is an inquiry into database. These statements give you filtered data according to your conditions and specifications indicating the fields, records and summaries which a user wants to manipulate from a database. It allows you to extract information from the database based on the conditions that you define in query.

e) Forms

In a database, a form is a window or a screen that contains numerous fields or spaces to enter data. Forms can be used to view and edit your data. It is an interface in user specified layout. E.g: user can create a data entry form that looks exactly like a paper form. People generally prefer to enter data into a well-designed form, rather than a table.

f) Report

When you want to print those records, which are fetched from your database. It is an effective way to represent data in a printed format. It allows you to represent data retrieved from one or more tables, so that it can be analyzed.

Remark: Database systems are made-up of complex data structures. To ease the user interaction with database, the developers hide internal irrelevant details from users. This process of hiding irrelevant details from user is called data abstraction.

We have three levels of abstraction:

1. **Physical level**: This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

2. **Logical level**: This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

3. **View level**: Highest level of data abstraction. This level describes the user interaction with database system.

Example: Let us say we are storing customer information in a customer table. At **physical level,** these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

At the **logical level** these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented. The programmers generally work at this level because they are aware of such things about database systems.

At **view level**, user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored; such details are hidden from them.

REVIEW QUESTIONS

1. a) What is database?

   b) What do you know about database system?

2. Differentiate file processing system from DBMS

3. What is the name given to a collection of logically related data?

4. Name the components of database.

5. What is the use of software in database?

6. What are types of users to access the database?

7. Explain the following terms:

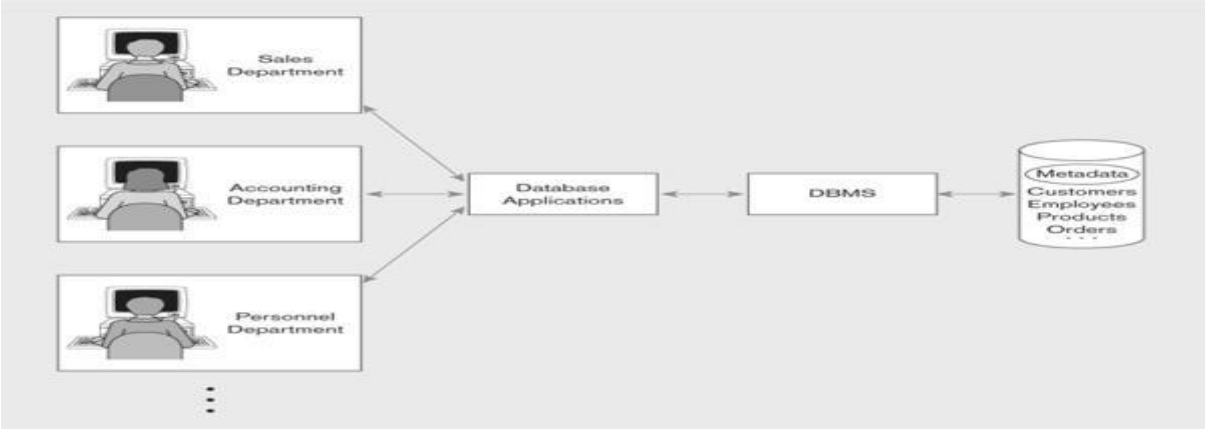
   a) Queries b) Attributes c) Tuples

8. Describe the three levels of data abstraction

Chapter II: Database Management System (DBMS)

II.1 what is DBMS?

DBMS stands for **D**atabase **M**anagement **S**ystem. A collection of programs, which enables users to create, maintain database and control all the access to the database. It is a computer based record keeping system.



The primary goal of the DBMS is to provide an environment that is both convenient and efficient for user to retrieve and store information. It acts as an interface between the application program and the data stored in the database.

DBMS is a software package that manages database: e.g : MYSQL,ORACLE, MS access.

DBMS is actually a tool that is used to perform any kind of operation on data in database. It also maintains data consistency in case of multiple users. The purpose of A DBMS is to bridge the gap between information and data. Some basic processes that are supported by a DBMS are as follows:

Specification of datatypes, structures and constraints to be considered in an

application Storing the data itself into persistent storage

Manipulation of the database

Querying the database to retrieve desired

information Updating the content of the database

II.2 A short list of database applications

Payroll

Orders

Reservation

Accounting

Etc

II.3 Features of a DBMS

Database Management Systems provide features to maintain database:

- **Data independence** - It refers to the immunity of user applications to make changes in the definition and organization of data. Data independence means that" the application is independent of the storage structure and access strategy of data". In other words, the ability to modify the schema definition in one level should not affect the schema definition of the next higher level.

There are two types of data independence:

1. **Physical data independence:** Modification in physical level should not affect the logical level.
2. **Logical data independence:** Modification in logical level should not affect the view level.

- **Integrity and security** - refers to maintaining and assuring the accuracy and consistency of data over its entire life cycle.

- **Transaction management** - A **transaction** comprises a unit of work performed within a DBMS against a database, and treated in a coherent and reliable way independent of other transactions. Transactions in a database environment have two main purposes:

  To provide isolation from other transactions.

  To have an "all or nothing" effect.

Remark: Transactions must pass the ACID test (atomic, consistent, isolated and durable).

- **Concurrency control** - ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.
- **Backup and recovery**
- **Provides a language for the creation and querying of the database**.
- **A language for writing application programs.**

II.4 Advantages of Database/DBMS

The centralized nature of database system provides several advantages, which overcome the limitations of the conventional file processing system.

These advantages are as follows:

1. **Reduce data redundancy**, redundancy means "duplication of data". This eliminates the replication of data item in different files, extra processing required to face the data item from a large database. This also ensures data consistency and saves the storage space.
2. **Enforcing data integrity**. It means that the data contained in the database is accurate and consistent. Integrity constraints or consistency rules can be applied to database, so that the correct data can be entered into the database.
3. **Data sharing.** The data stored in the database can be shared among multiple users or application programs.
4. **Data security.** The DBMS ensure that the access of database is done only through an authorized user.
5. **Ease of application development.** The application programmer needs to develop the application programs according to the user's need.
6. **Backup and recovery.** The DBMS provides backup and recovery sub-system that is responsible to recover data from hardware and software failures.
7. **Multiple views of data.** A view may be the subset of database. Various users may have different views of the database itself.
8. **Data independence.** System data descriptions are independent from the application programs.

II.5 Disadvantages of DBMS

1. **Cost**

DBMS requires high initial investment for hardware, software and trained staff. A significant investment based upon size and functionality of organization if required. Also organization has to pay concurrent annual maintenance cost.

2. **Complexity**

A DBMS fulfill lots of requirement and it solves many problems related to database. But all these functionality has made DBMS an extremely complex software. Developer, designer, DBA and End user of database must have complete skills if they want to user it properly. If they don't understand this complex system then it may cause loss of data or database failure.

3. **Technical staff requirement**

Any organization have many employees working for it and they can perform many others tasks too that are not in their domain but it is not easy for them to work on DBMS. A team of technical staff is required who understand DBMS and company have to pay handsome salary to them too.

4. **Database Failure**

As we know that in DBMS, all the files are stored in single database so chances of database failure become more. Any accidental failure of component may cause loss of valuable data. This is really a big question mark for big firms.

5. **Extra Cost of Hardware**

A DBMS requires disk storage for the data and sometimes you need to purchase extra space to store your data. Also sometimes you need to a dedicated machine for better performance of database. These machines and storage space increase extra costs of hardware.

6. **Size**

As DBMS becomes big software due to its functionalities, so it requires lots of space and memory to run its application efficiently. It gains bigger size as data is fed in it.

7. **Cost of Data Conversion**

Data conversion may require at any time and organization has to take this step. It is unbelievable that data conversion cost is more than the costs of DBMS hardware and machine combined. Trained staff is needed to convert data to new system. It is a key reason that most of the organizations are still working on their old DBMS due to high cost of data conversion.

8. **Currency Maintenance**

As new threats comes daily, so DBMS requires to updates itself daily. DBMS should be updates according to the current scenario.

9. **Performance**

Traditional files system was very good for small organizations as they give splendid performance. But DBMS gives poor performance for small scale firms as its speed is slow.

II.6 Relational Database Management System (RDBMS)

RDBMS is a type of database management system that stores data in the form of relations (Table).

Relational databases are powerful, so they require few assumptions about how data is related or how, it will be extracted from the database. An important feature of relational database system is that a single database can be spread across several tables. Base, oracle, MySQL, etc are the example of RDMS.

II.6.1 Types of Key Field

The key is defined as the column or the set of columns of the database table, which is used to identify each record uniquely in a relation. E.g: if a table has id, name and address as the column names then each one is known as the key for that table. The key field is a unique identifier for each record. E.g: in student table, you could use a combination of the lastname and fistname (or perhaps lastname, firstname to ensure you to identify each student uniquely) as a key field.

The following are the types of the fields available in the DBMS system:

1. **Primary key**

A field or a set of fields that uniquely identify each record in a table is known as a primary key. Relation has at least one column for which each row that must have a unique value. Only one column attributes can be defined as a primary key for each table.

A primary key must possess the following properties:

It does not allow null values.

It has a unique index

It allows numbers and text both

E.g : In the student table, studentId works as a primary key because it contains Ids which are unique for each student.

2. Candidate key

The set of all attributes which can uniquely identity each tuple of a relation are known as candidate's keys. Each table may have one or more candidate keys and one of them will become the primary key. The candidate key of a relation is always **a minimal key.**

Eg: Column studentId and the combination of firstname and lastname work as the candidate keys for the student table.

A candidate key must possess the following properties:

For each row, the value of the key must uniquely identify that row

No attribute in the key can be discarded without destroying the property of unique identification.

3. **Alternate key**

From the set of candidate keys after selecting one of the key as a primary key, all other remaining keys are known as alternate keys.

e.g: From the candidate keys (studentId, Fistaname,lastname). If studentId is chosen as a primary, then the firstname and lastname columns work as alternate keys.

4. **Foreign key**

A field of a table (relation) that references the primary key of another table is referred to as foreign key. The relationship between two tables is established with the help of foreign key. A table may have multiple foreign keys and each foreign key can have a different referenced table. Foreign keys play an essential role in database design, when tables are broken apart, then foreign keys make it possible for them to be reconstructed.

E.g: courseId column of student table (reference table) works as a foreign key as well as a primary key for course table (referenced table).

II.7 Basic Database Tools

It is a collective term for tool, utilities and assistants to perform database administrator, development and performance tuning for all major DBMS platforms.

Some database tools are as follows:

1. **MYSQL:** MySQL is an open-source relational database management system. Its name is a combination of "My", the name of co-founder Michael Widenius's daughter, and "SQL", the abbreviation for Structured Query Language.
2. **ORACLE:** Oracle database (Oracle DB) is a relational database management system (RDBMS) from the Oracle Corporation. Originally developed in 1977 by Lawrence Ellison and other developers, Oracle DB is one of the most trusted and widely used relational database engines.

The system is built around a relational database framework in which data objects may be directly accessed by users (or an application front end) through structured query language (SQL). Oracle is a fully scalable relational database architecture and is often used by **global enterprises, which manage and process data across wide** and **local area networks**. The Oracle database has its own network component to allow communications across networks.

Oracle DB is also known as Oracle RDBMS and, sometimes, just Oracle.

Review questions

1. What do you know about DBMS?
2. The purpose of DBMS is to bridge the gap between … and …
3. In which type of system, data is stored in the form of relations.
4. Which key of a relation is always a minimal key?
5. Define the term of candidate key.
6. What are the basis database tools? Write any names of database tools.
7. What is the difference between DBMS and RDBMS
8. What do you know about data independence?

Chapter III: Database design

Database design is the process of producing a detailed **data model (A collection of tools for describing data, data relationships, data semantics, and data constraints)** of a database. It describes many different parts of design of an overall database system. It is important to take time while designing a database because good database design is a keystone for creating a database that performs every task effectively, accurately and efficiently.

Phases for designing a database:

• Requirements specification and analysis

• Conceptual design

e.g., using the Entity-Relationship model

• Logical design

e.g., using the relational model

• Physical design

III.1 Steps to design a database

There are various steps to design a database, which are as follows:

Step 1: -Determine the purpose of your database. The first step of designing a database is to determine the purpose and mechanism to design and use it.

Step 2:- Determine the tables. Tables are one of the most important elements of a database. Consist of rows and columns. To create a well-defined database, you have to keep some conditions, which are as follows:

A table should not contain duplicate information

Each table should contain information about one subject

E.g: One table is used to contain the personal information of the students and the other is used to contain the marks scored by the student.

Step 3:- Determine the fields after creating a table; you need to describe the type and number of fields required for the tables in your database. Each field in a table contains individual facts about the table's subject.

Step 4: - Identify the primary key in a table from the fields of table; you need to identify a primary key, which uniquely identifies each individual record of the table. The primary key helps you to reduce data duplication in the table.

Step 5: -Determine the relationship between tables. In this step, you need to determine relationship between two or more tables in your database. You can set up a relationship between tables based on common field between them. Establishing a relationship allows you to fetch any information from both the tables.

Step 6:- Refine the design after you have designed the tables, fields and relationships, it is time to study the design and detect any faults that might remain.

Step 7:- Enter data and create other database objects when you are satisfied, that database structure meets the goals you need, add all your existing data to a table.

III.2 Types of data model

1. Entity-Relationship model

2. Relational model

3. Other models:

Older models: network model and hierarchical model

III.2.1 Entity Relationship Diagram (ERD)

An ERD is the conceptual structure, which is often represented as a schema. A schema describes the database structure in a shorthand notation. One example is the entity-relationship (ER) diagram, which consists of basic objects called entities and relationship among these objects. Entities are described in a database by a set of attributes.

a. **Entity**

It is a thing in the real world with an independent existence. Therefore, we have strong and weak entity.

b. **Entity type**

It is a collection (set) of entities that have same attributes.

c. **Entity set**

It is a collection of all entities of particular entity type in the database

d. **Weak entity set**

It is an entity that cannot be uniquely identified by its attributes alone; therefore, it must use a foreign identifier (primary key). The foreign key is typically a primary key of an entity that is related to. Ex: Loan and payment.

e. **Attribute**

It is a particular property, which describes the entity. There are five types of attributes:

1. **Simple:** can't be divided into other attributes
2. **Composite:** can be divided. Ex: Phone number(MTN, AIRTEL)
3. **Single valued:** only contains one value. Ex :Customer_Id
4. **Multivalued:** can contain many value. Ex: Customer-names
5. **Derived attributes:** Derived from another attributes. Ex. Birthday=Age

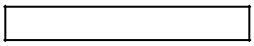
f. **Cardinality**

Cardinalities are used when you are creating an ERD and show the association between relations/entities. We have four types of cardinalities:

1. 1-1(one row in relation A relates to one row in relation B)
2. 1-many (one row in relation A relates to many rows in relation B)
3. Many-1 (Many rows in relation A relates to one row in relation B)
4. Many-many (Many rows in relation A relates to many rows in relation

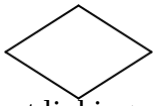
B) Here are symbols, which is useful in ERD:

1. Rectangle represents an entity set



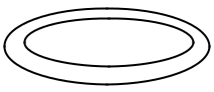
2. Ellipse represents an attribute (property)

3. Diamond represents an association/relationship



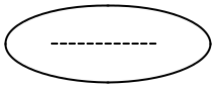
4. Lines represent linking of properties to an entity sets

5. Double ellipses represent multivalued attributes

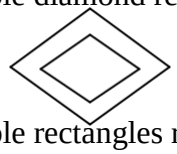


6. Dashed ellipses denote derived attributes

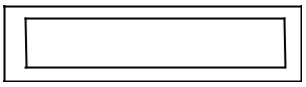
7. Underline indicates primary key/identifier



8. Double diamond represents association between weak and strong entity



9. Double rectangles represent weak entity set



Review questions

1. Sketch out the distinctions among the various types of attributes
2. Draft the differences between a weak and a strong entity set
3. A university registrar's office maintains data about the following entities;
    1. Courses, including number, title, credits, syllabus, and prerequisites.
    2. Course offerings, including course number, year, semester, section number, instructors, timings, and classrooms.
    3. Students, including student_id, name, and program
    4. Instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.

    Construct an E-R diagram for the registrar's office.

III.2.2 Relational database model

III.2.2.1 Basic Terms

An understanding of relational databases requires an understanding of some of the basic terms.

➢ **Data** are the values stored in table of the database. On its own, data means very little. "43156" is an example.

➢ **Information** is data that is processed to have a meaning. For example, "43156" is the population of the town of Littlewoods.

➢ **A database** is a collection of tables.

✓ Each table contains **records**, which are the horizontal rows in the table. These are also called **tuples**.

✓ Each record contains **fields**, which are the vertical columns of the table. These are also called **attributes**. An example would be a product record.

✓ **Fields** can be of many different types. There are many standard types, and each DBMS (database management system, such as **Oracle** or **MySQL**) can also have their own specific types, but generally they fall into at least three kinds - **character**, **numeric** and **date**. For example, a product description would be a character field, a product release date would be a date field, and a product quantity in stock would be a numeric field.

✓ **The domain** refers to the possible values each field can contain (it's sometimes called a field specification). For example, a field entitled "marital_status" may be limited to the values "Married" and "Unmarried".

✓ **A field** is said to contain **a null value** when it contains nothing at all. Fields can create complexities in calculations and have consequences for data accuracy. For this reason, many fields are specifically set not to contain NULL values.

✓ **A key** is a logical way to access a record in a table. For example, in the product table, the product_id field could allow us to uniquely identify a record. A key that uniquely identifies a record is called a primary key.

✓ **An index** is a physical mechanism that improves the performance of a database. Indexes are often confused with keys. However, strictly speaking they are part of the physical structure, while keys are part of the logical structure.

✓ **A view** is a virtual table made up of a subset of the actual tables.

✓ **A one-to-one (1:1) relationship** occurs where, for each instance of table A, only one instance of table B exists, and vice-versa. For example, each vehicle registration is associated with only one engine number, and vice-versa

✓ **A one-to-many (1:m) relationship** is where, for each instance of table A, many instances of the table B exist, but for each instance of table B, only once instance of table A exists. For example, for each artist, there are many paintings. Since it is a one-to-many relationship, and not many-to-many, in this case each painting can only have been painted by one artist.

✓ **A many to many (m:n) relationship** occurs where, for each instance of table A, there are many instances of table B, and for each instance of table B, there are many instances of the table A. For example, a poetry anthology can have many authors, and each author can appear in many poetry anthologies.

✓ **A mandatory relationship** exists where, for each instance of table A, one or more instances of table B must exist. For example, for a poetry anthology to exist, there must exist at least one poem in the anthology. The reverse is not necessarily true though, as for a poem to exist, there is no need for it to appear in a poetry anthology.

✓ **An optional relationship** is where, for each instance of table A, there may exist instances of table B. For example, a poet does not necessarily have to appear in a poetry anthology. The reverse isn't necessarily true though, for example for the anthology to be listed, it must have some poets.

III.2.2.2 Data Normalization

✓ **Data Normalization:** is the process of organizing the columns (attributes) and tables (relations) of a relational database to minimize data redundancy (occurs in database systems which have a field that is repeated in two or more tables) an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places.

✓ For instance, when customer data are duplicated and attached with each product bought, then redundancy of data is a known source of inconsistency since customer might appear with different values for given attribute.

✓ The process of modeling data into relational tables is known as normalization. There are commonly said to be three levels of normalization: the first, second, and third normal forms.

Level	Rule

First normal form (1NF)	An entity type is in 1NF when it contains no repeating groups of data
Second normal form (2NF)	An entity type is in 2NF when it is in 1NF and when all of its no-key attributes are fully dependent on its primary key.
Third normal form (3NF)	An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.

✓ It is possible for a SQL application to address un-normalized data, but this will usually be inefficient as that is not what the language is designed to do. If the systems analyst gets this wrong, the implications can be serious for performance, storage needs, and development effort. In most cases, data stored in a relational database and accessed with SQL should be normalized to the third normal form.

➢ As an example of normalization, consider an un-normalized table called BOOKS that stores details of books, authors, and publishers, using the ISBN number as the primary key. A primary key is the one attribute (or attributes) that can uniquely identify a record. There are two entries:

ISBN	Title	Authors	Publisher
12345	Oracle 11*g* OCP SQL Fundamentals 1 Exam Guide	John Watson, Roopesh Ramklass	McGraw-Hill, Spear Street, San Francisco, CA 94105
67890	Oracle 11*g* New Features Exam Guide	Sam Alapati	McGraw-Hill, Spear Street, San Francisco, CA 94105

✓ Storing the data in this table gives rise to several anomalies. First, here is the insertion anomaly: it is impossible to enter details of authors who are not yet published, because there will be no ISBN number under which to store them. Second, a book cannot be deleted without losing the details of the publisher: a deletion anomaly. Third, if a publisher's address changes, it will be necessary to update the rows for every book he has published: an update anomaly.

✓ Furthermore, it will be very difficult to identify every book written by one author. The fact that a book may have several authors means that the "author" field must be multivalued, and a search will have to search all the values. In addition, the storage is very inefficient due to replication of address details across rows, and the possibility of error as this data is repeatedly entered is high. Normalization should solve all these issues.

➢ **The first normal form** is to remove the repeating groups, in this case, the multiple authors: pull them out into a separate table called AUTHORS. The data structures will now look like the following.

Two rows in the BOOKS table:

ISBN	TITLE	PUBLISHER
12345	Oracle 11g OCP SQL Fundamentals 1 Exam Guide	McGraw-Hill, Spear Street, San Francisco, California
67890	Oracle 11g New Features Exam Guide	McGraw-Hill, Spear Street, San Francisco, California

And three rows in the AUTHOR table:

NAME	ISBN
John Watson	12345
Roopesh Ramklass	12345
Sam Alapati	67890

➢ The one row in the BOOKS table is now linked to two rows in the AUTHORS table. This solves the insertion anomaly (there is no reason not to insert as many unpublished authors as necessary), the retrieval problem of identifying all the books by one author (one can search the AUTHORS table on just one name) and the problem of a fixed maximum number of authors for any one book (simply insert as many or as few AUTHORS as are needed). This is the **first normal form**: no repeating groups.

➢ **The second normal form** removes columns from the table that are not dependent on the primary key. In this example, that is the **publisher's address details**: these are dependent on the publisher, not the ISBN. The BOOKS table and a new PUBLISHERS table will then look like this:

BOOKS

ISBN	TITLE	PUBLISHER
12345	Oracle 11g OCP SQL Fundamentals 1 Exam Guide	McGraw-Hill
67890	Oracle 11g New Features Exam Guide	McGraw-Hill

PUBLISHERS

PUBLISHER	STREET	CITY	STATE
McGraw-Hill	Spear Street	San Francisco	California

➢ All the books published by one publisher will now point to a single record in PUBLISHERS. This solves the problem of storing the address many times, and solves the consequent update anomalies and the data consistency errors caused by inaccurate multiple entries.

➢ **Third normal form** removes all columns that are interdependent. In the PUBLISHERS table, this means **the address columns:** the street exists in only one city, and the city can be in only one state; one column should do, not three. This could be achieved by adding an address code, pointing to a separate address table:

PUBLISHERS

PUBLISHER	ADDRESS CODE
McGraw-Hill	123

ADDRESSES

ADDRESS CODE	STREET	CITY	STATE
123	Spear Street	San Francisco	California

➢ One characteristic of normalized data that should be emphasized now is the use of primary keys and foreign keys.

➢ **A primary key** is the unique identifier of a row in a table. Every table should have a primary key defined. This is a requirement of the relational paradigm. Note that the relational database deviates from this standard: it is possible to define tables without a primary key—though it is usually not a good idea, and some other RDBMSs do not permit this.

➢ **A foreign key** is a column (or a concatenation of several columns) that can be used to identify a related row in another table. A foreign key in one table will match a primary key in another table. This is the basis of the many-to-one relationship.

➢ **A many-to-one relationship** is a connection between two tables, where many rows in one table refer to a single row in another table.

Review questions

1) a) When is a table in 1NF?

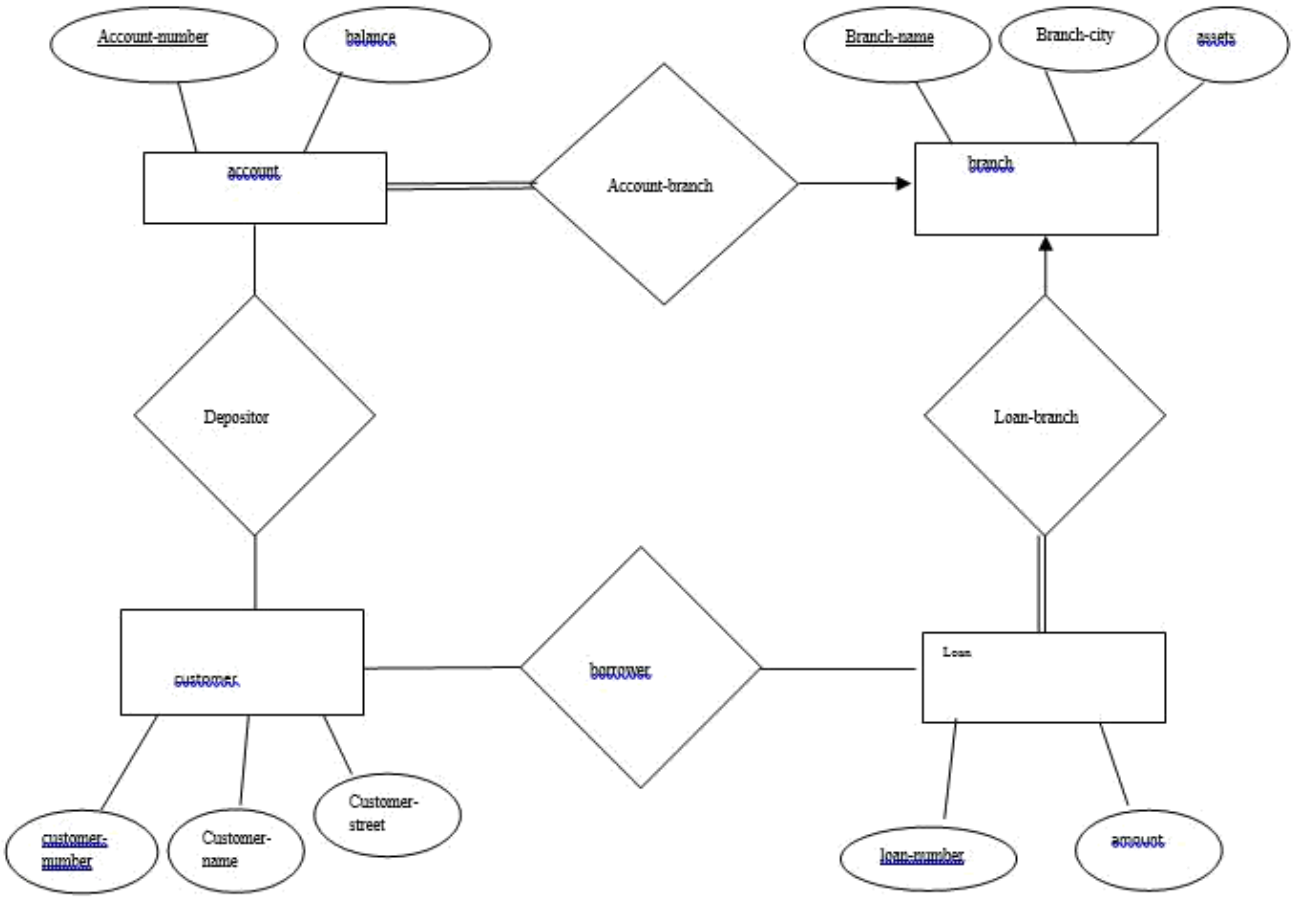
b) When is a table in 2NF? c)

When is a table in 3NF?

2) Demonstrate various level of Normalization for the following staffBranch table:

staffNo	name	position	salary	branchNo	branchAddress	telNo
S1500	Tom Daniels	Manager	46000	B001	8 Jefferson Way, Portland, OR 97201	503-555-3618
S0003	Sally Adams	Assistant	30000	B001	8 Jefferson Way, Portland, OR 97201	503-555-3618
S0010	Mary Martinez	Manager	50000	B002	City Center Plaza, Seattle, WA 98122	206-555-6756
S3250	Robert Chin	Supervisor	32000	B002	City Center Plaza, Seattle, WA 98122	206-555-6756
S2250	Sally Stern	Manager	48000	B004	16 – 14th Avenue, Seattle, WA 98128	206-555-3131
S0415	Art Peters	Manager	41000	B003	14 – 8th Avenue, New York, NY 10012	212-371-3000

3) Design a relational database(logical level) corresponding to the E-R diagram (conceptual level)


Account-number
balance
account
Account-branch
Branch-name
Branch-city
assets
branch
Depositor
Loan-branch
customer
borrower
Loan
customer-number
Customer-name
Customer-street
loan-number
amount

Chapter IV: An overview of SQL

IV.1 what is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in relational database.
SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server use SQL as standard database language.

IV.2 Why SQL?

5. Allows users to access data in relational database management systems.
6. Allows users to describe the data.
7. Allows users to define the data in database and manipulate that data.
8. Allows embedding within other languages using SQL modules, libraries & pre-compilers.
9. Allows users to create and drop databases and tables.
10. Allows users to create view, stored procedure, functions in a database.
11. Allows users to set permissions on tables, procedures and views.

IV.3 SQL Constraints

A constraint allows the database designer to enforce business rules about the data stored in the database's tables and the relationships between tables These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.
The following are commonly used constraints available in SQL:
NOT NULL Constraint: A value must be supplied for this column, but values do not have to be unique.
DEFAULT Constraint: Provides a default value for a column when none is specified.
UNIQUE Constraint: Every value in this column must be unique, but null values are allowed
PRIMARY Key: Every value in the column must be unique and cannot be null.
FOREIGN Key: Every value in the column must match a value in another column in this table or some other table; otherwise, the value is null.
CHECK Constraint: The value entered in the table must match one of the specified values for this column
INDEX: Use to create and retrieve data from the database very quickly.

IV.4 Data Integrity

The following categories of the data integrity exist with each RDBMS:

1. **Entity Integrity:** There are no duplicate rows in a table.

2. **Domain Integrity:** Enforces valid entries for a given column by restricting the type, the format, or the range of values.

3. **Referential Integrity:** Rows cannot be deleted which are used by other records.

4. **User-Defined Integrity:** Enforces some specific business rules that do not fall into entity, domain, or referential integrity.

IV.5 SQL DATATYPE

SQL data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQL. You would use these data types while creating your tables. You would choose a particular data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use: beauty

IV.5.1 Exact Numeric Data Types:

DATA TYPE	FROM	TO
Bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
Int	-2,147,483,648	2,147,483,647
Smallint	-32,768	32,767
Tinyint	0	255
Bit	0	1
Decimal	-10^38 +1	10^38 -1
Numeric	-10^38 +1	10^38 -1
Money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
Smallmoney	-214,748.3648	+214,748.3647

IV.5.2 Approximate Numeric Data Types:

DATA TYPE	FROM	TO
Float	-1.79E + 308	1.79E + 308
Real	-3.40E + 38	3.40E + 38

IV.5.3 Date and Time Data Types:

DATA TYPE	FROM	TO
Datetime	Jan 1, 1753	Dec 31, 9999
Smalldatetime	Jan 1, 1900	Jun 6, 2079
Date	Stores a date like June 30, 1991	
Time	Stores a time of day like 12:30	

Note: Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

IV.5.4 Character Strings Data Types:

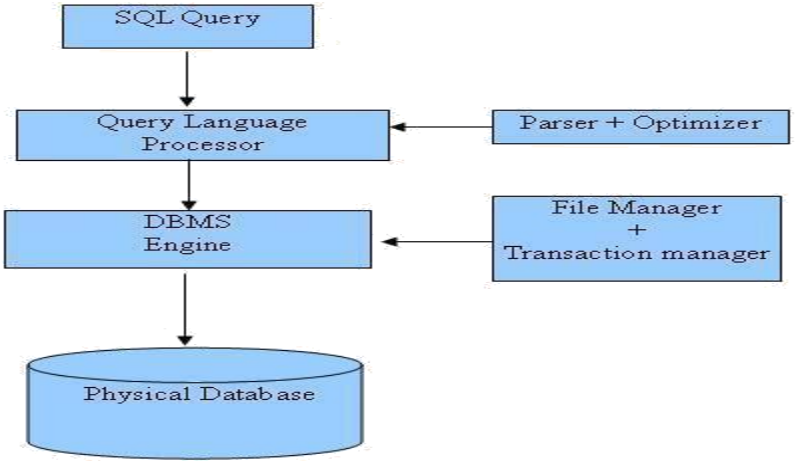
DATA TYPE	FROM	TO
Char	Char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
Varchar	Varchar	Maximum of 8,000 characters.(Variable-length non-Unicode data).
varchar(max)	varchar(max)	Maximum length of 231characters, Variable-length non-Unicode data (SQLServer2005 only).
Text	text	Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

IV.6 SQL Process:

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries, but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture:



IV.7 SQL Commands:

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into groups based on their nature:

IV.7.1 DDL - Data Definition Language:

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.

IV.7.2 DML - Data Manipulation Language:

Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

IV.7.3 DCL - Data Control Language:

Command	Description
GRANT	Gives a privilege to user
REVOKE	Takes back privileges granted from user

IV.7.4 TCL: Transaction Control Language

A **Transaction Control Language** (**TCL**) is a computer language and a subset of SQL, used to control transactional processing in a database. A transaction is logical unit of work that comprises one or more SQL statements, usually a group of Data Manipulation Language (DML) statements.

COMMANDS	DESCRIPTION
COMMIT	To apply the transaction by saving the database changes.
ROLLBACK	To undo all changes of a transaction.
SAVEPOINT	To divide the transaction into smaller sections. It defines breakpoints for a transaction to allow partial rollbacks.

CHAPTER V: SQL Simulation

V.1 CREATE DATABASE

The SQL **CREATE DATABASE** statement is used to create new SQL database.

Syntax:

Basic syntax of CREATE DATABASE statement is as follows:

```
CREATE DATABASE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

If you want to create new database <testDB>, then CREATE DATABASE statement would be as follows:

```
MYSQL> CREATE DATABASE testDB;
```

Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases as follows:

```
MYSQL> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| AMROOD             |
| TUTORIALSPOINT     |
| mysql              |
| orig               |
| test               |
| testDB             |
+--------------------+
7 rows in set (0.00 sec)
```

V.2 DROP DATABASE

The SQL **DROP DATABASE** statement is used to drop an existing database in SQL schema.

Syntax:

Basic syntax of DROP DATABASE statement is as follows:

```
DROP DATABASE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

If you want to delete an existing database <testDB>, then DROP DATABASE statement would be as follows:

```
MYSQL> DROP DATABASE testDB;
```

NOTE: Be careful before using this operation because by deleting an existing database would result in loss of complete information stored in the database.

Make sure you have admin privilege before dropping any database. Once a database is dropped, you can check it.

```
MYSQL> in the list of databases as follows: SHOW DATABASES;
  ---------------------+
| Database             |
+--------------------  +
| information_schema |
| AMROOD               |
| TUTORIALSPOINT   |
| mysql                |
| orig                 |
| test                 |
+--------------------  +
6 rows in set (0.00 sec)
```

V.3 SQL SELECT DATABASE

When you have multiple databases in your SQL Schema, then before starting your operation, you would need to select a database where all the operations would be performed.

The SQL **USE** statement is used to select any existing database in SQL schema.

Syntax:

Basic syntax of USE statement is as follows:

```
USE DatabaseName;
```

Always database name should be unique within the RDBMS.

Example:

You can check available databases as follows:

```
MYSQL> SHOW DATABASES;
+--------------------        +
| Database             |
+--------------------        +
| information_schema |
| AMROOD               |
| TUTORIALSPOINT   |
| mysql                |
| orig                 |
| test                 |
+--------------------        +
6 rows in set (0.00 sec)
```

Now, if you want to work with AMROOD database, then you can execute the following SQL command and start working with AMROOD database:

```
MYSQL> USE AMROOD;
```

V.4 SQL CREATE TABLE

Creating a basic table involves naming the table and defining its columns and each column's data type. The SQL **CREATE TABLE** statement is used to create a new table.

Syntax:

Basic syntax of CREATE TABLE statement is as follows:

```
CREATE            TABLE
table_name(                                                                    column1
    datatype,       column2
    datatype,       column3
    datatype,
    .....
    columnN datatype,PRIMARY KEY( one or more columns )
);
```

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check complete details at **Create Table Using another Table.**

V.5 CREATE TABLE USING ANOTHER TABLE

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.
The new table has the same column definitions. All columns or specific columns can be selected. When you create a new table using existing table, new table would be populated using existing values in the old table.

Syntax:

The basic syntax for creating a table from another table is as follows:

```
CREATE TABLE NEW_TABLE_NAME AS
    SELECT [column1, column2...columnN]
    FROM EXISTING_TABLE_NAME
    [WHERE]
```

Here, column1, column2...are the fields of existing table and same would be used to create fields of new table.

Example:

Following is an example, which would create a table SALARY using CUSTOMERS table and having fields customer ID and customer SALARY:

```
MYSQL> CREATE TABLE
    SALARY AS SELECT ID,
    SALARY
    FROM CUSTOMERS;
```

This would create new table SALARY, which would have the following records:

```
+----  +----------  +
 | ID | SALARY  |
----+   ----------+
| 1 |    2000.00 |
| 2 |    1500.00 |
| 3 |    2000.00 |
| 4 |    6500.00 |
| 5 |    8500.00 |
| 6 |    4500.00 |
| 7 | 10000.00 |
+---  +---------  +
```

Example:

Following is an example, which creates a CUSTOMERS table with ID as primary key and

NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

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

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use **DESC** command as follows:

```
MYSQL> DESC CUSTOMERS;
+---------  +--------------      +------  +-----+---------       +-------   +
| Field    | Type          | Null | Key | Default | Extra |
                                          +------
+---------                      +------  +-----     ---          +-------       +
| ID       | int(11)       | NO       |PRI|       |       |
| NAME     | varchar(20)     | NO    |   |       |          |
| AGE      | int(11)           |NO  |     |       |       |
| ADDRESS | char(25)          |YES |    | NULL      |       |
| SALARY   | decimal(18,2) | YES  |     | NULL     |          |
+---------  +--------------      +------  +-----+---------       +-------   +
5 rows in set (0.00 sec)
```

Now, you have CUSTOMERS table available in your database, whi ch you can use to store required information related to customers.

V.6 SQL DROP TABLE

The SQL **DROP TABLE** statement is used to remove a table definition and all data, indexes, triggers, constraints, and permission specifications for that table.
NOTE: You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

Syntax:

Basic syntax of DROP TABLE statement is as follows:

```
DROP TABLE table_name;
```

Example:

Let us first verify CUSTOMERS table and then we would delete it from the database:

```
MYSQL> DESC CUSTOMERS;
+---------  +--------------- +------ +-----+---------     +-------       +
| Field    | Type           | Null | Key | Default | Extra |
+---------  +--------------- +------ +-----+---------     +-------       +
| ID       | int(11)         | NO     |PRI|         |       |
| NAME    | varchar(20)       | NO    |   |       |          |
| AGE      | int(11)           |NO  |     |       |       |
```

```
| ADDRESS | char(25)     | YES  |     | NULL    |       |
| SALARY  | decimal(18,2) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

This means CUSTOMERS table is available in the database, so let us drop it as follows:

```
MYSQL> DROP TABLE CUSTOMERS;
Query OK, 0 rows affected (0.01 sec)
```

Now, if you would try DESC command, then you would get error as follows:

```
MYSQL> DESC CUSTOMERS;
ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist
```

Here, TEST is database name, which we are using for our examples.

V.7 SQL INSERT QUERY

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

Syntax:

There are two basic syntaxes of INSERT INTO statement as follows:

```
INSERT INTO TABLE_NAME (column1, column2,
column3,...columnN)] VALUES (value1, value2, value3,...valueN);
```

Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQL INSERT INTO syntax would be as follows:

```
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN);
```

Example:

Following statements would create six records in CUSTOMERS table:

```
INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (1,
'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (2,
'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (3,
'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY) VALUES (4,
'Chaitali', 25, 'Mumbai', 6500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

You can create a record in CUSTOMERS table using second syntax as follows:

```
INSERT INTO CUSTOMERS
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

All the above statements would produce the following records in CUSTOMERS table:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

V.8 POPULATE ONE TABLE USING ANOTHER TABLE

You can populate data into a table through select statement over another table provided another table has a set of fields, which are required to populate first table. Here is the syntax:

```
INSERT INTO first_table_name [(column1, column2, ... columnN)]
   SELECT column1, column2, ...columnN
   FROM second_table_name
   [WHERE condition];
```

V.9 SQL SELECT QUERY

SQL **SELECT** Statement is used to fetch the data from a database table, which returns data in the form of result table. These result tables are called result-sets.

Syntax:

The basic syntax of SELECT statement is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2...are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax:

```
SELECT * FROM table_name;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

The Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

```
MYSQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  1 | Ramesh   |  2000.00 |
|  2 | Khilan   |  1500.00 |
|  3 | kaushik  |  2000.00 |
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

If you want to fetch all the fields of CUSTOMERS table, then use the following query:

```
MYSQL> SELECT * FROM CUSTOMERS;
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

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

V.10 SQL WHERE CLAUSE

The SQL **WHERE** clause is used to specify a condition while fetching the data from single table or joining with multiple tables.

If the given condition is satisfied, then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause is not only used in SELECT statement, but it is also used in UPDATE, DELETE statement, etc., which we would examine in subsequent chapters.

Syntax:

The basic syntax of SELECT statement with WHERE clause is as follows:

```
SELECT column1, column2,
columnN FROM table_name
WHERE [condition]
```

You can specify a condition using comparison or logical operators like >, <, =, LIKE, NOT etc. Below examples would make this concept clear.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

```
MYSQL> SELECT ID,
NAME, SALARY FROM
CUSTOMERS
WHERE SALARY > 2000;
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  4 | Chaitali |  6500.00 |
|  5 | Hardik   |  8500.00 |
|  6 | Komal    |  4500.00 |
|  7 | Muffy    | 10000.00 |
+----+----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table for a customer with name **Hardik**. Here, it is important to note that all

the strings should be given inside single quotes (') where as numeric values should be given without any quote as in above example:

```
MYSQL> SELECT ID,
NAME, SALARY FROM
CUSTOMERS
WHERE NAME = 'Hardik';
```

This would produce the following result:

```
+----+----------+----------+
| ID | NAME     | SALARY   |
+----+----------+----------+
|  5 | Hardik   | 8500.00  |
+----+----------+----------+
```

V.11 SQL AND\ OR OPERATORS

The SQL **AND** and **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called conjunctive operators.
These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

V.11.1 The AND Operator

The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

Syntax:

The basic syntax of AND operator with WHERE clause is as follows:

```
SELECT column1, column2,
columnN FROM table_name
WHERE [condition1] AND [condition2]...AND [conditionN];
```

You can combine N number of conditions using AND operator. For an action to be taken by the SQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 AND age is less tan 25 years:

```
MYSQL> SELECT ID,
NAME, SALARY FROM
CUSTOMERS
WHERE SALARY > 2000 AND age < 25;
```

This would produce the following result:

```
+----+-------+----------+
| ID | NAME  | SALARY   |
+----+-------+----------+
|  6 | Komal |  4500.00 |
|  7 | Muffy | 10000.00 |
+----+-------+----------+
```

V.11.2 The OR Operator:

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

Syntax:

The basic syntax of OR operator with WHERE clause is as follows:

```
SELECT column1, column2,
columnN FROM table_name
WHERE [condition1] OR [condition2]...OR [conditionN]
```

You can combine N number of conditions using OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 OR age is less tan 25 years:

```
MYSQL> SELECT ID,
NAME, SALARY FROM
CUSTOMERS
WHERE SALARY > 2000 OR age < 25;
```

This would produce the following result:

ID	NAME	SALARY
3	kaushik	2000.00
4	Chaitali	6500.00
5	Hardik	8500.00
6	Komal	4500.00
7	Muffy	10000.00

V.12 SQL UPDATE QUERY

The sql update query is used to modify the existing records in a table.You can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

Syntax:

The basic syntax of UPDATE query with WHERE clause is as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would update ADDRESS for a customer whose ID is 6:

```
MYSQL> UPDATE
CUSTOMERS SET
ADDRESS = 'Pune'
WHERE ID = 6;
```

Now, CUSTOMERS table would have the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | Pune      | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

If you want to modify all ADDRESS and SALARY column values in CUSTOMERS table, you do not need to use
WHERE clause and UPDATE query would be as follows:

```
MYSQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune', SALARY = 1000.00;
```

Now, CUSTOMERS table would have the following records:

```
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----  +----------  +-----+---------    +----------  +
|  1| Ramesh   |  32 | Pune      | 1000.00 |
|  2| Khilan    | 25 | Pune    | 1000.00 |
|  3| kaushik   | 23 | Pune     | 1000.00 |
|  4| Chaitali |  25 | Pune    | 1000.00 |
|  5| Hardik    | 27 | Pune     | 1000.00 |
|  6| Komal     |  22 | Pune      | 1000.00 |
|  7| Muffy     | 24 | Pune     | 1000.00 |
+----  +----------  +-----+---------    +----------  +
```

V.13 SQL DELETE QUERY

The sql delete query is used to delete the existing records from a table.You can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

Syntax:

The basic syntax of DELETE query with WHERE clause is as follows:

```
DELETE FROM table_name
WHERE [condition];
```

You can combine N number of conditions using AND or OR operators.

Example:

Consider the CUSTOMERS table having the following records:

```
+----  +----------  +-----+-----------+----------    +
| ID | NAME     | AGE | ADDRESS      | SALARY   |
----+    ----------+-----  ----------+----------+   +
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan    | 25 | Delhi        | 1500.00 |
|  3 | kaushik   | 23 | Kota        | 2000.00 |
|  4 | Chaitali |    25 | Mumbai          | 6500.00 |
|  5 | Hardik    | 27 | Bhopal     |   8500.00   |
|  6 | Komal     |  22|MP           |   4500.00 |
|  7 | Muffy     | 24 | Indore     | 10000.00   |
+----  +----------  +-----+-----------+----------    +
```

Following is an example, which would DELETE a customer, whose ID is 6:

```
MYSQL> DELETE FROM
CUSTOMERS WHERE ID
= 6;
```

Now, CUSTOMERS table would have the following records:

```
                -----+---------   +-------
+----------  +            --        ---      +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
+----  +----------+      +-----------+----------  +
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
|  2| Khilan    | 25 | Delhi       | 1500.00 |
|  3| kaushik   | 23 | Kota        | 2000.00 |
|  4| Chaitali |  25 | Mumbai       | 6500.00 |
|  5| Hardik    | 27 | Bhopal     |  8500.00  |
|  7| Muffy     | 24 | Indore     | 10000.00  |
+----  +----------+      +-----------+----------  +
```

If you want to DELETE all the records from CUSTOMERS table, you do not need to use WHERE clause and DELETE query would be as follows:

```
MYSQL> DELETE FROM CUSTOMERS;
```

Now, CUSTOMERS table would not have any record.

V.14 SQL LIKE CLAUSE

The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

Syntax:

The basic syntax of % and _ is as follows:

```
SELECT FROM table_name
WHERE column LIKE 'XXXX%'

or
SELECT FROM table_name
WHERE column LIKE '%XXXX%'

or

SELECT FROM table_name
WHERE column LIKE 'XXXX_'or SELECT FROM table_name
WHERE column LIKE '_XXXX' or SELECT FROM table_name WHERE column LIKE
'_XXXX_'
```

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '_' operators:

Statement	**Description**
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY LIKE '2_%_%'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY LIKE ‘ 2_____3’	Finds any values in a five-digit number that start with 2 and end with 3

Let us take a real example, consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would display all the records from CUSTOMERS table where SALARY starts with 200:

```
MYSQL> SELECT * FROM
CUSTOMERS WHERE
SALARY LIKE '200%';
```

This would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
+----+----------+-----+-----------+----------+
```

V.15 SQL TOP CLAUSE

The SQL **TOP** clause is used to fetch a TOP N number or X percent records from a table.

Note: All the databases do not support TOP clause. For example MySQL supports **LIMIT** clause to fetch limited number of records and Oracle uses **ROWNUM** to fetch limited number of records.

Syntax:

The basic syntax of TOP clause with SELECT statement would be as follows:

```
SELECT TOP number|percent
column_name(s) FROM table_name
WHERE [condition]
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----  +----------  + -----+----------- +----------    +
| ID | NAME     | AGE | ADDRESS     | SALARY  |
+----  +----------  + ----- +----------- +----------    +
| 1  | Ramesh   | 32 | Ahmedabad | 2000.00 |
|  2| Khilan    | 25 | Delhi       | 1500.00 |
|  3| kaushik   | 23 | Kota        | 2000.00 |
|  4| Chaitali |  25 | Mumbai      | 6500.00 |
|  5| Hardik    | 27 | Bhopal     | 8500.00   |
|  6| Komal     |  22|MP           |  4500.00 |
|  7| Muffy         | 24 | Indore             | 10000.00  |
+----  +----------  +   -----+----------- +----------    +
```

Following is an example on SQL server, which would fetch top 3 records from CUSTOMERS table:

```
MYSQL> SELECT TOP 3 * FROM CUSTOMERS;
```

This would produce the following result:

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

If you are using MySQL server, then here is an equivalent example:

```
MYSQL>  SELECT  *  FROM
CUSTOMERS LIMIT 3;
```

This would produce the following result:

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

If you are using Oracle server, then here is an equivalent example:

```
MYSQL>  SELECT  *  FROM
CUSTOMERS            WHERE
ROWNUM <= 3;
```

This would produce the following result:

```
+---- +---------+---     + ----------+---------    +
| ID | NAME      | AGE | ADDRESS    | SALARY  |
+---- +---------+---     + ----------+---------    +
```

```
| 1| Ramesh   | 32 | Ahmedabad | 2000.00 |
| 2| Khilan   | 25 | Delhi     | 1500.00 |
| 3| kaushik  | 23 | Kota      | 2000.00 |
```

V.16 SQL ORDER CLAUSE

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause is as follows:

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would sort the result in ascending order by NAME and SALARY:

```
MYSQL> SELECT * FROM
       CUSTOMERS ORDER
       BY NAME, SALARY;
```

This would produce the following result:

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
```

```
|   3| kaushik   | 23 | Kota       | 2000.00 |
|   2| Khilan    | 25 | Delhi      | 1500.00 |
|   6| Komal     |  22|MP          |  4500.00 |
|   7| Muffy     | 24 | Indore     | 10000.00 |
| 1  | Ramesh    | 32 | Ahmedabad | 2000.00 |
+----  +----------  +-----+-----------+----------  +
```

Following is an example, which would sort the result in descending order by NAME:

```
MYSQL> SELECT * FROM
       CUSTOMERS   ORDER
       BY NAME DESC;
```

This would produce the following result:

```
                    -----+--------
  +----  +----------  +             ---   +---------      +
  | ID | NAME     | AGE | ADDRESS     | SALARY  |
  +----  +----------  + -----+-----------+---------      +
  | 1  | Ramesh   | 32 | Ahmedabad | 2000.00 |
  | 7 | Muffy     | 24 | Indore     | 10000.00 |
  | 6 | Komal     |  22|MP          | 4500.00 |
  | 2 | Khilan    | 25 | Delhi      |  1500.00 |
  | 3 | kaushik   | 23 | Kota       | 2000.00 |
  | 5 | Hardik    | 27 | Bhopal     |   8500.00 |
  | 4 | Chaitali |  25 | Mumbai     | 6500.00 |
+----  _         +-----    +-----------+----------  +
```

V.17 SQL GROUP BY

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

Example:

Consider the CUSTOMERS table having the following records:

```
                    -----+---------   +-------
+----   +----------  +                --          ---       +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
----+    ----------+-----  ----------+----------+   +
| 1  | Ramesh      | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan   |  25 | Delhi       | 1500.00 |
| 3 | kaushik   | 23 | Kota       | 2000.00 |
| 4 | Chaitali |  25 | Mumbai      | 6500.00 |
| 5 | Hardik   |  27 | Bhopal    |  8500.00  |
| 6 | Komal    |  22|MP           |  4500.00 |
|  7 | Muffy       | 24 | Indore           | 10000.00  |
+---   +----------   +    -----+----------   +---------   +
```

If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

```
MYSQL>                                                    SELECT     NAME,
      FROM CUSTOMERS GROUP BY NAME;
```

This would produce the following result:

```
| NAME      | SUM(SALARY) |
  +----------+-------------+
| Chaitali |      6500.00 |
| Hardik     |    8500.00 |
| kaushik    |    2000.00 |
| Khilan     |    1500.00 |
| Komal      |    4500.00 |
| Muffy      |   10000.00 |
| Ramesh     |     2000.00 |
```

Now, let us have following table where CUSTOMERS table has the following records with duplicate names:

```
                    -----+---------
+----  +----------  +                --   +---------       +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
+----  +----------  + -----+----------- +---------       +
| 1  | Ramesh   | 32 | Ahmedabad | 2000.00 |
|  2| Ramesh   |  25 | Delhi     |   1500.00 |
|  3| kaushik   |23 | Kota        |  2000.00 |
|  4| kaushik   |25 | Mumbai       | 6500.00 |
|  5| Hardik    |27 | Bhopal     |   8500.00 |
|  6| Komal     | 22|MP           |4500.00 |
|  7| Muffy     |24 | Indore          | 10000.00 |
+----  +----------   +   -----+----------   +---------       +
```

Now again, if you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

```
MYSQL> SELECT NAME, SUM(SALARY)
       FROM CUSTOMERS GROUP BY NAME;
```

This would produce the following result:

```
+---------+                +
| NAME      | SUM(SALARY) |
+---------+                +
| Hardik   |      8500.00 |
| kaushik |      8500.00 |
| Komal    |      4500.00 |
| Muffy    |     10000.00 |
| Ramesh  |      3500.00 |
+---------+                +
```

V.18 SQL DISTINCT KEYWORD

The SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

```
SELECT DISTINCT column1, column2,.....columnN FROM table_name
WHERE [condition]
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

First, let us see how the following SELECT query returns duplicate salary records:

```
MYSQL> SELECT SALARY FROM
       CUSTOMERS ORDER BY
       SALARY;
```

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

```
+----------+
| SALARY   |
+----------+
| 1500.00  |
| 2000.00  |
| 2000.00  |
| 4500.00  |
| 6500.00  |
| 8500.00  |
| 10000.00 |
+----------+
```

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
MYSQL> SELECT DISTINCT SALARY
       FROM CUSTOMERS ORDER BY
       SALARY;
```

This would produce the following result where we do not have any duplicate entry:

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

V.19 SQL SORTING RESULTS

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort query results in ascending order by default. **Syntax:**

The basic syntax of ORDER BY clause, which would be used to sort result in ascending or descending order, is as follows:

```
SELECT column-list
FROM    table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would sort the result in ascending order by NAME and SALARY:

```
MYSQL> SELECT * FROM CUSTOMERS ORDER
```

```
BY NAME, SALARY;
```

This would produce the following result:

```
+----  +----------  +-----+-----------+----------    +
| ID | NAME     | AGE | ADDRESS    | SALARY  |
+----  +----------  +-----+-----------+----------    +
|   4| Chaitali |  25 | Mumbai       | 6500.00 |
|   5| Hardik    | 27 | Bhopal     | 8500.00    |
|   3| kaushik   | 23 | Kota        | 2000.00 |
|   2| Khilan    | 25 | Delhi       | 1500.00 |
|   6| Komal     |  22|MP          |  4500.00 |
|   7| Muffy     | 24 | Indore     | 10000.00  |
| 1  | Ramesh    | 32 | Ahmedabad | 2000.00 |
+----  +----------  +-----+-----------+----------    +
```

Following is an example, which would sort the result in descending order by NAME:

```
MYSQL>  SELECT  *  FROM
        CUSTOMERSORDER
        BY NAME DESC;
```

This would produce the following result:

```
+----  +----------  +-----+-----------    +---------        +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
+----  +----------  +-----+-----------    +---------        +
| 1  | Ramesh    | 32 | Ahmedabad | 2000.00 |
|   7| Muffy     | 24 | Indore       | 10000.00 |
|   6| Komal     | 22|MP           | 4500.00 |
|   2| Khilan    | 25 | Delhi       | 1500.00 |
|   3| kaushik   | 23 | Kota        | 2000.00 |
|   5| Hardik    | 27 | Bhopal       | 8500.00 |
|   4| Chaitali | 25 | Mumbai                | 6500.00 |
+----  +----------  + -----+-----------     +---------        +
```

To fetch the rows with own preferred order, the SELECT query would be as follows:

```
MYSQL>                                                    SELECT    *       FROM
        CUSTOMERS ORDER BY
        (CASE ADDRESS
        WHEN 'DELHI'         THEN 1
        WHEN 'BHOPAL'        THEN 2
        WHEN 'KOTA'   THEN 3
        WHEN 'AHMADABAD' THEN 4
        WHEN 'MP'    THEN 5
        ELSE 100 END) ASC, ADDRESS DESC;
```

This would produce the following result:

```
                -----+---------   +-------
+----   +----------  +             --        ---      +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
----+    ----------+-----  ----------+----------+   +
|  2 | Khilan    | 25 | Delhi        |  1500.00 |
|  5 | Hardik    | 27 | Bhopal       |   8500.00  |
|  3 | kaushik   |  23 | Kota        | 2000.00 |
|  6 | Komal     |  22|MP           |   4500.00 |
|  4 | Chaitali |  25 | Mumbai       | 6500.00 |
|  7 | Muffy     | 24 | Indore     | 10000.00  |
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
+---  +----------  +-----+-----------+----------   +
```

This will sort customers by ADDRESS in your ownoOrder of preference first and in a natural order for the remaining addresses. Also remaining Addresses will be sorted in the reverse alpha order.

V.20 SQL OPERATION

V.20.1 what is an operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation (s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

V.20.2 SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

Here are simple examples showing usage of SQL Arithmetic Operators:

```
MYSQL> select 10+ 20;
+-------      +
|10+20|
+-------      +
|      30 |
+-------      +
1 row in set (0.00 sec)

MYSQL> select 10 * 20;
+--------          +
|10*20|
+--------          +
|      200 |
+--------          +
1 row in set (0.00 sec)

MYSQL> select 10 / 5;
+--------   +
|10/5|
+--------   +
| 2.0000 |
+--------   +
1 row in set (0.03 sec)

MYSQL> select 12 % 5;
+---------    +
|12%      5 |
+---------    +
|       2 |
+---------        +
1 row in set (0.00 sec)
```

V.20.3 SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes t	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition	(a > b) is not true.

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

Consider the CUSTOMERS table having the following records:

```
MYSQL> 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  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
7 rows in set (0.00 sec)
```

Here are simple examples showing usage of SQL Comparison Operators:

```
MYSQL> SELECT * FROM CUSTOMERS WHERE SALARY > 5000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  | 6500.00  |
|  5 | Hardik   |  27 | Bhopal  | 8500.00  |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
3 rows in set (0.00 sec)

MYSQL> SELECT * FROM CUSTOMERS WHERE SALARY = 2000;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
+----+----------+-----+-----------+----------+
2 rows in set (0.00 sec)

MYSQL> SELECT * FROM CUSTOMERS WHERE SALARY != 2000;
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   | 1500.00  |
|  4 | Chaitali |  25 | Mumbai  | 6500.00  |
|  5 | Hardik   |  27 | Bhopal  | 8500.00  |
|  6 | Komal    |  22 | MP      | 4500.00  |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
5 rows in set (0.00 sec)
```

```
MYSQL> SELECT * FROM CUSTOMERS WHERE SALARY <> 2000;
```

```
| ID | NAME     | AGE | ADDRESS | SALARY     |
+----+----------+-----+---------+------------+
|  2| Khilan   | 25 | Delhi     | 1500.00 |
|  4| Chaitali | 25 | Mumbai    |  6500.00 |
|  5| Hardik   | 27 | Bhopal    | 8500.00 |
|  6| Komal    | 22|MP          | 4500.00 |
|  7| Muffy    | 24 | Indore    | 10000.00  |
+----+----------+-----+---------+------------+
5 rows in set (0.00 sec)
```

```
MYSQL> SELECT * FROM CUSTOMERS WHERE SALARY >= 6500;
+----+----------+-----+---------+------------+
| ID | NAME     | AGE | ADDRESS | SALARY     |
+----+----------+-----+---------+------------+
|  4| Chaitali | 25 | Mumbai    |  6500.00 |
|  5| Hardik   | 27 | Bhopal    | 8500.00  |
|  7| Muffy    | 24 | Indore    | 10000.00  |
+----+----------+-----+---------+------------+
3 rows in set (0.00 sec)
```

V.20.4 SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Consider the CUSTOMERS table having the following records:

```
MYSQL> 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 |
|  7| Muffy     | 24 | Indore     | 10000.00  |
+----  +----------  + -----+----------- +----------    +
7 rows in set (0.00 sec)
```

Here are simple examples showing usage of SQL Comparison Operators:

```
MYSQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;
+----  +----------  +-----+--------- +----------   +
| ID | NAME     | AGE | ADDRESS | SALARY  |
+----  +----------  +-----+--------- +----------   +
|  4 | Chaitali | 25 | Mumbai     | 6500.00 |
|  5 | Hardik   | 27 | Bhopal   | 8500.00 |
+----  +----------  +-----+--------- +----------   +
2 rows in set (0.00 sec)


MYSQL> SELECT * FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;
+---     +----------  +-----+----------- +----------       +
| ID | NAME     | AGE | ADDRESS     | SALARY   |
+----  +----------  +-----+----------- +----------       +
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan   | 25 | Delhi      | 1500.00 |
|  4  | Chaitali | 25 | Mumbai                  | 6500.00 |
|  5 | Hardik   | 27 | Bhopal       | 8500.00 |
|  7 | Muffy    | 24 | Indore      | 10000.00 |
+----  +----------  +-----+----------- +----------       +
5 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE IS NOT NULL;
----+    -----------+------  ------------  ----------+
| ID | NAME     | AGE | ADDRESS     | SALARY   |
+----  +----------      +-----+----------          +----------        +
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan   | 25 | Delhi      | 1500.00 |
|  3 | kaushik  | 23 | Kota         | 2000.00 |
|  4 | Chaitali | 25 | Mumbai       | 6500.00 |
```

```
|  5| Hardik   | 27 | Bhopal      | 8500.00 |
|  6| Komal    | 22|MP           |  4500.00 |
|  7| Muffy    | 24 | Indore     | 10000.00 |
+---- +----------  +----- +----------    +----------  +
7 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE NAME LIKE 'Ko%';
+----  +-------  +---------  +---------  +
| ID | NAME  | AGE | ADDRESS | SALARY  |
+----  +-------  +---------  +---------  +
|  6| Komal |  22|MP          | 4500.00 |
+----  +-------  +---------  +---------  +
1 row in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE IN ( 25, 27 );
+---- +----------  +----- +---------  +
ID | NAME     | AGE | ADDRESS | SALARY  |
+---- +----------  +----- +---------  +
 | 2 | Khilan    | 25 | Delhi      | 1500.00 |
|  4| Chaitali | 25 | Mumbai     | 6500.00 |
|  5| Hardik   | 27 | Bhopal     | 8500.00 |
+---- +----------  +----- +---------  +
3 rows in set (0.00 sec)

SQL> SELECT * FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;
+---- +----------  +----- +---------  +
| ID | NAME     | AGE | ADDRESS | SALARY  |
+---- +----------  +----- +---------  +
|  2| Khilan   | 25 | Delhi      | 1500.00 |
|  4| Chaitali | 25 | Mumbai     | 6500.00 |
|  5| Hardik   | 27 | Bhopal     | 8500.00 |
+---- +----------  +----- +---------  +
3 rows in set (0.00 sec)

SQL> SELECT AGE FROM CUSTOMERS
WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+-----+
|AGE|
+-----+
| 32|
| 25|
| 23|
| 25|
| 27|
| 22|
| 24|
+-----+
7 rows in set (0.02 sec)
```

```
MYSQL> SELECT * FROM CUSTOMERS
WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+----+---------+-----+-----------+---------+
| ID | NAME    | AGE | ADDRESS   | SALARY  |
+----+---------+-----+-----------+---------+
| 1  | Ramesh  | 32  | Ahmedabad | 2000.00 |
+----+---------+-----+-----------+---------+
1 row in set (0.02 sec)

MYSQL> SELECT * FROM CUSTOMERS
WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
|  2 | Khilan   | 25  | Delhi     | 1500.00  |
|  4 | Chaitali | 25  | Mumbai    | 6500.00  |
|  5 | Hardik   | 27  | Bhopal    | 8500.00  |
+----+----------+-----+-----------+----------+
4 rows in set (0.00 sec)
```

V.21 SQL CONSTRAITS

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Contraints could be column level or table level. Column level constraints are applied only to one column where as table level constraints are applied to the whole table.
Following are commonly used constraints available in SQL. These constraints have already been discussed in SQL - RDBMS Concepts chapter but its worth to revise them at this point.

Following are commonly used constraints available in SQL:

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

V.21.1 NOT NULL Constraint:

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

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

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

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

V.21.2 DEFAULT Constraint:

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

Example

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

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

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

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

V.21.2.1 Drop Default Constraint

To drop a DEFAULT constraint, use the following SQL:

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

V.21.3 UNIQUE Constraint:

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

Example:
For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

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

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

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

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

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

V.21.3.1 DROP a UNIQUE Constraint:

To drop a UNIQUE constraint, use the following SQL:

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

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

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

V.21.4 Primary Key:

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

Note: You would use these concepts while creating database tables.

V.21.4.1 Create Primary Key:

Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

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

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

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

NOTE: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

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

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

V.21.4.2 Delete Primary Key:

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

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

V.21.5 Foreign Key:

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

Example:

Consider the structure of the two tables as follows:

CUSTOMERS table:

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

ORDERS table:

```
CREATE TABLE ORDERS (
        ID         INT       NOT NULL,
        DATE       DATETIME,
        CUSTOMER_ID INT references CUSTOMERS(ID),
```

```
        AMOUNT     double,
        PRIMARY KEY (ID)
);
```

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

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

V.21.5.1 DROP a FOREIGN KEY Constraint:

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

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

V.21.6 INDEX:

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically and are then referred to one or more specific page numbers.

An index helps speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with **UPDATE** and **INSERT** statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the **CREATE INDEX statemen**t, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

V.21.6.1 The CREATE INDEX Command:

The basic syntax of **CREATE INDEX** is is as follows:

```
CREATE INDEX index_name ON table_name;
```

Example:

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

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

1. **Single-Column Indexes:**

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX index_name
ON table_name (column_name);
```

2. **Unique Indexes:**

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE INDEX index_name
on table_name (column_name);
```

3. **Composite Indexes:**

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single -column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions. Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

4. **Implicit Indexes:**

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

V.21.6.2 The DROP INDEX Command:

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

You can check INDEX Constraint chapter to see actual examples on Indexes.

➢ **Dropping Constraints:**

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint, and then enable it later.

V.21.6.3 When should indexes be avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

➢ **Integrity Constraints:**

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in referential integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints mentioned above.

V.22 SQL JOIN

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables in our SELECT statement as follows:

```
MYSQL> SELECT ID, NAME,
       AGE, AMOUNT FROM
       CUSTOMERS, ORDERS
       WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

V.22.1 SQL Join Types:

There are different types of joins available in SQL:

- **INNER JOIN:** returns rows when there is a match in both tables.
- **LEFT JOIN:** returns all rows from the left table, even if there are no matches in the right table.
- **RIGHT JOIN:** returns all rows from the right table, even if there are no matches in the left table.
- **FULL JOIN:** returns rows when there is a match in one of the tables.
- **SELF JOIN:** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN:** returns the Cartesian product of the sets of records from the two or more joined tables.

V.22.1.1 INNER JOIN

The most frequently used and important of the joins is the **INNER JOIN**. They are also referred to as an EQUIJOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join -predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join -predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

The basic syntax of **INNER JOIN** is as follows:

```
SELECT   table1.column1,   table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_filed = table2.common_field;
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
                    -----+---------   +-------
+---    +----------  +              --         ---      +
| ID | NAME     | AGE | ADDRESS     | SALARY  |
----+    ----------+-----  ----------+----------+   +
| 1  | Ramesh      | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan    | 25 | Delhi       | 1500.00 |
|  3 | kaushik   | 23 | Kota        | 2000.00 |
|  4 | Chaitali |  25 | Mumbai      | 6500.00 |
|  5 | Hardik    | 27 | Bhopal    |  8500.00  |
|  6 | Komal     |  22|MP           |  4500.00 |
|  7  | Muffy          | 24 | Indore               | 10000.00  |
+----   +----------  +-----+-----------+----------   +
```

(b) Another table is ORDERS as follows:

```
+-----  +--------------------+-------------  +--------         +
| OID | DATE                |         ID | AMOUNT |
+-----  +--------------------+-------------  +--------         +
| 102 | 2009-10-08 00:00:00 |          3 |   3000 |
| 100 | 2009-10-08 00:00:00 |          3 |   1500 |
```

```
| 101 | 2009-11-20 00:00:00 |          2 |  1560 |
| 103 | 2008-05-20 00:00:00 |          4 |  2060 |
+-----+---------------------+------------+--------+
```

Now, let us join these two tables using INNER JOIN as follows:

```
MYSQL> SELECT ID, NAME,
       AMOUNT, DATE FROM
       CUSTOMERS
       INNER JOIN ORDERS
       ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

V.22.1.2 LEFT JOIN

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of **LEFT JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_filed = table2.common_field;
```

Here given condition could be any given expression based on your requirement.

Example:

Consider the following two tables; (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
```

```
| 5 | Hardik |    27 | Bhopal |  8500.00 |
| 6 | Komal     |22|MP         |  4500.00 |
| 7 | Muffy     |24 | Indore     | 10000.00 |
```

(b) Another table is ORDERS as follows:

```
| OID | DATE                | CUSTOMER_ID | AMOUNT |
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
```

Now, let us join these two tables using LEFT JOIN as follows:

```
MYSQL> SELECT ID, NAME,
       AMOUNT, DATE FROM
       CUSTOMERS
       LEFT JOIN ORDERS
       ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
| ID | NAME    | AMOUNT | DATE                |
| 1  | Ramesh  |   NULL | NULL                |
| 2  | Khilan  |   1560 | 2009-11-20 00:00:00 |
| 3  | kaushik |   3000 | 2009-10-08 00:00:00 |
| 3  | kaushik |   1500 | 2009-10-08 00:00:00 |
| 4  | Chaitali|   2060 | 2008-05-20 00:00:00 |
| 5  | Hardik  |   NULL | NULL                |
| 6  | Komal   |   NULL | NULL                |
| 7  | Muffy   |   NULL | NULL                |
```

V.22.1.3 RIGHT JOIN

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of **RIGHT JOIN** is as follows:

```
SELECT table1.column1, table2.column2..FROM table1 RIGHT
JOIN table2 ON table1.common_filed = table2.common_field;
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using RIGHT JOIN as follows:

```
MYSQL> SELECT ID, NAME,
       AMOUNT, DATE FROM
       CUSTOMERS
       RIGHT JOIN ORDERS
       ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

V.22.1.4 FULL JOIN

The SQL **FULL JOIN** combines the results of both left and right outer joins.The joined table

will contain all records from both tables, and fill in NULLs for missing matches on either side.

Syntax:

The basic syntax of **FULL JOIN** is as follows:

```
SELECT   table1.column1,   table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_filed = table2.common_field;
```

Here given condition could be any given expression based on your requirements.

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----  +----------+ -----+----------  +----------   +
| ID | NAME     | AGE | ADDRESS  | SALARY   |
+----  +----------+ -----+----------  +----------   +
| 1  | Ramesh    | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan    | 25 | Delhi       |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |   6500.00 |
|  5 | Hardik    | 27 | Bhopal      |  8500.00 |
|  6 | Komal     | 22|MP          |  4500.00 |
|  7 | Muffy     |  24 | Indore     | 10000.00 |
+----  +----------+ -----+----------  +----------   +
```

(b) Another table is ORDERS as follows:

```
+      +                        +             +         +
  ----  ------------------          -----------                -------
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
     -------------
-----+     --------+    -------------+           --------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+----- +------------------    +-------------    +--------         +
```

```
MYSQL>   SELECTID,              NAME,
        AMOUNT,       DATE        FROM
        CUSTOMERS
        FULL JOIN ORDERS
        ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+------  +----------          +--------------------  +
| ID   | NAME     | AMOUNT | DATE                 |
+------  +----------          +--------------------  +
|    1| Ramesh   |     NULL | NULL                 |
|    2| Khilan    |  1560 | 2009-11-20 00:00:00 |
|    3| kaushik   |  3000 | 2009-10-08 00:00:00 |
|    3| kaushik   |  1500 | 2009-10-08 00:00:00 |
```

```
|    4| Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5| Hardik    | NULL | NULL                 |
|    6| Komal     |  NULL | NULL                |
|    7| Muffy     | NULL | NULL                 |
|    3| kaushik   | 3000 | 2009-10-08 00:00:00 |
|    3| kaushik   | 1500 | 2009-10-08 00:00:00 |
|    2| Khilan    | 1560 | 2009-11-20 00:00:00 |
|   4 | Chaitali |    2060 | 2008-05-20 00:00:00 |
+------ +---------- +-------   +--------------------  +
```

If your Database does not support FULL JOIN like MySQL does not support FULL JOIN, then you can use **UNION ALL** clause to combine two JOINS as follows:

```
MYSQL>    SELECT     ID,    NAME,
       AMOUNT,      DATE       FROM
       CUSTOMERS
       LEFT       JOIN       ORDERS       ON
CUSTOMERS.ID                              =
ORDERS.CUSTOMER_ID UNION ALL
       SELECT  ID, NAME, AMOUNT,
       DATE FROM CUSTOMERS
       RIGHT JOIN ORDERS
       ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

V.22.1.5 SELF JOIN

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

Syntax:

The basic syntax of **SELF JOIN** is as follows:

```
SELECT                        a.column_name,
b.column_name... FROM table1 a, table1 b
WHERE a.common_filed = b.common_field;
```

Here, WHERE clause could be any given expression based on your requirement.

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
                    -----+---------
+---    +----------  +              --   +---------       +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
----+    ----------+-----  ----------+    ----------+
| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan    | 25 | Delhi        |  1500.00 |
| 3 | kaushik   | 23 | Kota         |  2000.00 |
| 4 | Chaitali |  25 | Mumbai       | 6500.00 |
| 5 | Hardik    | 27 | Bhopal     |   8500.00 |
| 6 | Komal     |  22|MP           |  4500.00 |
| 7 | Muffy     | 24 | Indore      | 10000.00 |
```

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

Now, let us join this table using SELF JOIN as follows:

```
MYSQL> SELECT a.ID, b.NAME,
       a.SALARY FROM CUSTOMERS
       a, CUSTOMERS b WHERE
       a.SALARY < b.SALARY;
```

This would produce the following result:

```
+----  +---------- + ---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2| Ramesh    | 1500.00 |
|  2| kaushik   | 1500.00 |
|  1| Chaitali | 2000.00 |
|  2| Chaitali | 1500.00 |
|  3| Chaitali | 2000.00 |
|  6| Chaitali | 4500.00 |
|  1| Hardik    | 2000.00 |
|  2| Hardik    | 1500.00 |
|  3| Hardik    | 2000.00 |
|  4| Hardik    | 6500.00 |
| 6 | Hardik    | 4500.00 |
|  1| Komal     | 2000.00 |
|  2| Komal     | 1500.00 |
|  3| Komal     | 2000.00 |
|  1| Muffy     | 2000.00 |
|  2| Muffy     | 1500.00 |
|  3| Muffy     | 2000.00 |
|  4| Muffy     | 6500.00 |
|  5| Muffy     | 8500.00 |
|  6| Muffy     | 4500.00 |
+----_       +---------   +
```

V.23 SQL ALIAS SYNTAX

You can rename a table or a column temporarily by giving another name known as alias.

The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

Syntax:

The basic syntax of **table** alias is as follows:

```
SELECT column1, column2....
FROM table_name AS
alias_name WHERE [condition];
```

The basic syntax of **column** alias is as follows:

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

(b) Another table is ORDERS as follows:

```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, following is the usage of **table alias**:

```
MYSQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT FROM CUSTOMERS AS C, ORDERS AS O WHERE C.ID = O.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Following is the usage of **column alias**:

```
MYSQL> SELECT   ID AS CUSTOMER_ID, NAME AS
       CUSTOMER_NAME FROM CUSTOMERS
       WHERE SALARY IS NOT NULL;
```

This would produce the following result:

```
+------------      +--------------      +
| CUSTOMER_ID | CUSTOMER_NAME |
+-------------     +--------------      +
|           1| Ramesh         |
|           2| Khilan         |
|           3| kaushik        |
|           4| Chaitali       |
|           5| Hardik         |
|           6| Komal          |
|           7| Muffy          |
+-------------     +--------------      +
```

V.24 SQL ALTER TABLE COMMAND

The SQL **ALTER TABLE** command is used to add, delete or modify columns in an existing table. You would also use ALTER TABLE command to add and drop various constraints on an existing table.

Syntax:

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

```
ALTER TABLE table_name ADD column_name datatype;
```

The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

```
ALTER TABLE table_name MODIFY column_name datatype NOT NULL;
```

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);
```

The basic syntax of ALTER TABLE to **ADD CHECK CONSTRAINT** to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);
```

The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);
```

The basic syntax of ALTER TABLE to **DROP CONSTRAINT** from a table is as follows:

```
ALTER TABLE table_name
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name
DROP INDEX MyUniqueConstraint;
```

The basic syntax of ALTER TABLE to **DROP PRIMARY KEY** constraint from a table is as follows:

```
ALTER TABLE table_name
DROP CONSTRAINT MyPrimaryKey;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example to ADD a new column in an existing table:

```
ALTER TABLE CUSTOMERS ADD SEX char(1);
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

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

```
| 4 | kaushik | 25 | Mumbai      | 6500.00 | NULL |
|  5| Hardik  | 27 | Bhopal      | 8500.00| NULL |
| 6  | Komal   | 22|MP          | 4500.00 | NULL |
|  7| Muffy   | 24 | Indore      | 10000.00| NULL |
+---- +--------- +-----+----------  +---------- +------+
```

Following is the example to DROP sex column from existing table:

```
ALTER TABLE CUSTOMERS DROP SEX;
```

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

V.25 TRUNCATE TABLE

The SQL **TRUNCATE TABLE** command is used to delete complete data from an existing table. You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

Syntax:

The basic syntax of **TRUNCATE TABLE** is as follows:

```
TRUNCATE TABLE table_name;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+---  +---------  +-----+----------+----------  +
| ID | NAME     | AGE | ADDRESS     | SALARY  |
----+   ----------+-----  ----------+----------+   +
| 1  | Ramesh      | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan    | 25 | Delhi       | 1500.00 |
|  3 | kaushik   | 23 | Kota       | 2000.00 |
|  4 | Chaitali |   25 | Mumbai         | 6500.00 |
|  5 | Hardik    | 27 | Bhopal    |   8500.00  |
|  6 | Komal     |  22|MP          |   4500.00 |
|  7 | Muffy     | 24 | Indore    | 10000.00  |
+---  +---------  +-----+----------+----------  +
```

Following is the example to truncate:

```
MYSQL > TRUNCATE TABLE CUSTOMERS;
```

Now, CUSTOMERS table is truncated and following would be the output from SELECT statement:

```
MYSQL>  SELECT  *  FROM
CUSTOMERS; Empty set (0.00
sec)
```

V.26 SQL-USING VIEW

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

can contain all rows of a table or select rows from a table. A view can be created from one or many tables, which depends on the written SQL query to create a view. Views, which are virtual tables, allow users to do the following:

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

V.26.1 Creating Views:

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

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

The basic CREATE VIEW syntax is as follows:

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

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

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

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

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

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

```
MYSQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result:

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

V.26 .2 Updating a View:

A view can be updated under certain conditions:

1. The SELECT clause may not contain the keyword DISTINCT.
2. The SELECT clause may not contain summary functions.
3. The SELECT clause may not contain set functions.
4. The SELECT clause may not contain set operators.
5. The SELECT clause may not contain an ORDER BY clause.
6. The FROM clause may not contain multiple tables.
7. The WHERE clause may not contain subqueries.
8. The query may not contain GROUP BY or HAVING.
9. Calculated columns may not be updated.
10. All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above mentioned rules then you can update a view. Following is an example to update the age of Ramesh:

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

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

result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 35  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

V.26.3 Inserting Rows into a View:

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

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

V.26.4 Deleting Rows into a View:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.
Following is an example to delete a record having AGE= 22.

```
MYSQL > DELETE FROM
      CUSTOMERS_VIEW
      WHERE age = 22;
```

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

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 35  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
```

V.26.5 Dropping Views:

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

```
DROP VIEW view_name;
```

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

```
DROP VIEW CUSTOMERS_VIEW;
```

V.27 SQL HAVING CLAUSE

The HAVING clause enables you to specify conditions that filter which group results appear in the final results.The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

The following is the position of the HAVING clause in a query:

```
SELECT   FROM   WHERE
GROUP    BY     HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

```
SELECT column1, column2
FROM table1, table2
WHERE [ conditions ]
GROUP BY column1, column2
HAVING   [   conditions   ]
ORDER    BY     column1,
column2
```

Example:

Consider the CUSTOMERS table having the following records:

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

Following is the example, which would display record for which similar age count would be more than or equal to 2:

```
MYSQL>
SELECT      *
FROM
CUSTOMERS
GROUP     BY
age
HAVING    COUNT(age)
>= 2;
```

This would produce the following result:

```
----+--------+-----+---------+---------+
| ID | NAME   | AGE | ADDRESS | SALARY  |
+----+--------+-----+---------+---------+
| 2 | Khilan | 25 | Delhi     | 1500.00 |
 +----+--------+-----+---------+---------+
```

V.28 SQL TRANSACTION

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record, updating a record, or deleting a record from the table, then you are performing transaction on the table. It is important to control transactions to ensure data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

V.28.1Properties of Transactions:

1. Transactions have the following four standard properties, usually referred to by the acronym ACID:
    - **Atomicity:** ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
    - **Consistency:** ensures that the database properly changes states upon a successfully committed transaction.
    - **Isolation:** enables transactions to operate independently of and transparent to each other.
    - **Durability:** ensures that the result or effect of a committed transaction persists in case of a system failure.

V.28. 2 Transaction Control:

There are following commands used to control transactions:

- **COMMIT:** to save the changes.
- **ROLLBACK:** to rollback the changes.
- **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK
- **SET TRANSACTION:** Places a name on a transaction.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

V.28. 2 .1 The COMMIT Command:

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for COMMIT command is as follows:

```
COMMIT;
```

Example:

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  2 | Khilan   |  25 | Delhi     | 1500.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  4 | Chaitali |  25 | Mumbai    | 6500.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is the example, which would delete records from the table having age = 25 and then COMMIT the changes in the database.

```
MYSQL> DELETE FROM
       CUSTOMERS WHERE
       AGE = 25;
MYSQL> COMMIT;
```

As a result, two rows from the table would be deleted and SELECT statement would produce the following result:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad | 2000.00  |
|  3 | kaushik  |  23 | Kota      | 2000.00  |
|  5 | Hardik   |  27 | Bhopal    | 8500.00  |
|  6 | Komal    |  22 | MP        | 4500.00  |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

V.28..2.2 The ROLLBACK Command:

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.

The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK
command was issued.

The syntax for ROLLBACK command is as follows:

```
```

```
ROLLBACK;
```

Example:

Consider the CUSTOMERS table having the following records:
Following is the example, which would delete records from the table having age = 25 and then ROLLBACK the changes in the database.

```
MYSQL>  DELETE  FROM
        CUSTOMERS
        WHERE AGE = 25;
MYSQL> ROLLBACK;
```

As a result, delete operation would not impact the table and SELECT statement would produce the following result:

```
                    -----+---------   +-------
+---    +----------  +              --         ---       +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
----+    ----------+-----  ----------+----------+    +

| 1  | Ramesh       | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan     | 25 | Delhi        | 1500.00 |
| 3 | kaushik   | 23 | Kota        | 2000.00 |
| 4 | Chaitali |  25 | Mumbai       | 6500.00 |
| 5 | Hardik    | 27 | Bhopal     |  8500.00  |
| 6 | Komal     |  22|MP          |  4500.00 |
| 7 | Muffy     | 24 | Indore     | 10000.00  |
+----  +----------  +-----+----------+----------    +
```

V.28. 2.3 The SAVEPOINT Command:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The syntax for SAVEPOINT command is as follows:

```
SAVEPOINT SAVEPOINT_NAME;
```

This command serves only in the creation of a SAVEPOINT among transactional statements.
The ROLLBACK
command is used to undo a group of transactions.

The syntax for rolling back to a SAVEPOINT is as follows:

```
ROLLBACK TO SAVEPOINT_NAME;
```

Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state:

Example:

Consider the CUSTOMERS table having the following records:
Now, here is the series of operations:

```
MYSQL>SAVEPOINT SP1;
Savepoint created.
```

```
MYSQL> DELETE FROM CUSTOMERS WHERE
ID=1; 1 row deleted.
MYSQL>
SAVEPOINT    SP2;
Savepoint created.
MYSQL> DELETE FROM CUSTOMERS WHERE
ID=2; 1 row deleted.
MYSQL>
SAVEPOINT    SP3;
Savepoint created.
MYSQL> DELETE FROM CUSTOMERS WHERE
ID=3; 1 row deleted.
```

Now that the three deletions have taken place, say you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
MYSQL>ROLLBACK  TO SP2;
Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2:

```
MYSQL>          SELECT          *          FROM          CUSTOM;
```

```
+----  +----------  + -----+----------- +----------    +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
+----  +----------  + -----+----------- +----------    +
|   2| Khilan   |  25 | Delhi        | 1500.00 |
|   3| kaushik  |  23 | Kota         | 2000.00 |
|   4| Chaitali |  25 | Mumbai       | 6500.00 |
|   5| Hardik   |  27 | Bhopal       | 8500.00    |
|   6| Komal    |  22|MP           |  4500.00 |
|   7| Muffy    |  24 | Indore       | 10000.00  |
+----  +----------  + -----+----------- +----------    +
6 rows selected.
```

V.28.2.4 The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that

you have created. The syntax for RELEASE SAVEPOINT is as follows:

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

V.28.2.5 The SET TRANSACTION Command:

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows.

For example, you can specify a transaction to be read only or read write. The syntax for SET

TRANSACTION is as follows:

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

V.29 Sql Usefuly Functions

SQL has many built-in functions for performing processing on string or numeric data. Following is the list of all useful SQL built-in functions:

- SQL COUNT Function - The SQL COUNT aggregate function is used to count the number of rows in a database table.
- SQL MAX Function - The SQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.
- SQL MIN Function - The SQL MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
- SQL AVG Function - The SQL AVG aggregate function selects the average value for certain

table column.

- SQL SUM Function - The SQL SUM aggregate function allows selecting the total for a numeric column.
- SQL SQRT Functions - This is used to generate a square root of a given number.
- SQL RAND Function - This is used to generate a random number using SQL command.
- SQL CONCAT Function - This is used to concatenate any string inside any SQL command.
- SQL Numeric Functions - Complete list of SQL functions required to manipulate numbers in SQL.
- SQL String Functions - Complete list of SQL functions required to manipulate strings in SQL.

V.29.1 SQL COUNT Function

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

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

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

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

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

```
MYSQL>SELECT COUNT(*) FROM employee_tbl
       -> WHERE name="Zara";
+----------          +
| COUNT(*) |
+----------          +
|          2 |
+----------          +
1 row in set (0.04 sec)
```

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

V.29.2 SQL MAX Function

SQL **MAX** function is used to find out the record with maximum value among a record set.

To understand **MAX** function, consider an **employee_tbl** table, which is having the following records:

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

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

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

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

```
MYSQL> SELECT id, name, MAX(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
------+     ------  ------------------------+
| id  | name | MAX(daily_typing_pages) |
+-----  +-----   +------------------------        +
|    3  | Jack |                        170 |
|    4  | Jill |                        220 |
```

```
|       1 |  John   |                250 |
|       2 |  Ram    |                220 |
|       5 |  Zara   |                350 |
+---------+---------+--------------------+
5 rows in set (0.00 sec)
```

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

```
MYSQL> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max
    -> FROM employee_tbl;
+-------+------+
| least | max  |
+-------+------+
|   100 |  350 |
+-------+------+
1 row in set (0.01 sec)
```

V.29.3 SQL MIN Function

SQL **MIN** function is used to find out the record with minimum value among a record set.

To understand **MIN** function, consider an **employee_tbl** table, which is having the following records:

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

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

```
MYSQL> SELECT MIN(daily_typing_pages)
    -> FROM employee_tbl;
+--------------------------+
| MIN(daily_typing_pages) |
+--------------------------+
|                      100 |
```

```
+-------------------------+
1 row in set (0.00 sec)
```

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

```
MYSQL> SELECT id, name, work_date, MIN(daily_typing_pages)
    -> FROM employee_tbl GROUP BY name;
+------+------+-------------------------+
| id   | name | MIN(daily_typing_pages) |
+------+------+-------------------------+
|    3 | Jack |                     100 |
|    4 | Jill |                     220 |
|    1 | John |                     250 |
|    2 |Ram   |                     220 |
|    5 | Zara |                     300 |
+------+------+-------------------------+
5 rows in set (0.00 sec)
```

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

```
MYSQL> SELECT MIN(daily_typing_pages) least,
    -> MAX(daily_typing_pages) max
    -> FROM employee_tbl;
+-------+------+
| least | max  |
+-------+------+
|   100 |  350 |
+-------+------+
1 row in set (0.01 sec
```

V.29.4 SQL AVG Function

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

```
MYSQL> SELECT * FROM employee_tbl;
+------+------+------------+--------------------+
| id   | name | work_date  | daily_typing_pages |
+------+------+------------+--------------------+
|    1 | John | 2007-01-24 |                250 |
|    2 | Ram  | 2007-05-27 |                220 |
|    3 | Jack | 2007-05-06 |                170 |
|    3 | Jack | 2007-04-06 |                100 |
|    4 | Jill | 2007-04-06 |                220 |
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
```

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

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

```
MYSQL> SELECT AVG(daily_typing_pages)
       -> FROM employee_tbl;
+--------------------------+
| AVG(daily_typing_pages) |
+--------------------------+
|                230.0000 |
+--------------------------+
1 row in set (0.03 sec)
```

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

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

V.29.5 SQL SUM Function

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

```
MYSQL> SELECT * FROM employee_tbl;
+-----  +------ +------------ +--------------------+
| id   | name | work_date  | daily_typing_pages |
+-----  +------ +------------ +--------------------+
|    1| John | 2007-01-24 |                250 |
|    2| Ram  | 2007-05-27 |                220 |
|    3| Jack | 2007-05-06 |                170 |
|    3| Jack | 2007-04-06 |                100 |
|    4| Jill | 2007-04-06 |                220 |
```

```
|    5 | Zara | 2007-06-06 |                300 |
|    5 | Zara | 2007-02-06 |                350 |
+------+------+------------+--------------------+
7 rows in set (0.00 sec)
```

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

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

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

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

V.30 SQL SUBQUERIES

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

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

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

Things to Keep in Mind

1. A subquery is just a SELECT statement inside of another.
2. Subqueries are always enclosed in parenthesis ().
3. A subquery that returns a single value can be used anywhere you would use an expression, such as in a column list or filter expression.
4. A subquery that returns more than one value is typically used where a list of values, such as those used in and **IN** operator.
5. Warning! Subqueries can be very inefficient. If there are more direct means to achieve the same result, such as using an inner join, you are better for it.

V.30.1 Sub queries with the SELECT Statement:

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

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

Example:

Consider the CUSTOMERS table having the following records:

```
                  -----+---------   +--------
+---   +----------  +              --      --          +
| ID | NAME     | AGE | ADDRESS      | SALARY  |
+---   +----------  + -----+----------- +----------    +
| 1  | Ramesh   | 35 | Ahmedabad | 2000.00 |
|   2| Khilan   | 25 | Delhi       | 1500.00 |
|   3| kaushik  | 23 | Kota        | 2000.00 |
|   4| Chaitali |  25 | Mumbai       | 6500.00 |
|   5| Hardik   | 27 | Bhopal     | 8500.00   |
|   6| Komal    |  22|MP          |  4500.00 |
|   7| Muffy    | 24 | Indore     | 10000.00  |
+---   +----------  + -----+----------- +----------    +
```

Now, let us check the following subquery with SELECT statement:

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

This would produce the following result:

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  5 | Hardik   |  27 | Bhopal  |  8500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

V.30.2 Subqueries with the INSERT Statement:

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

The basic syntax is as follows:

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

Example:

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

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

V.30.3 Subqueries with the UPDATE Statement:

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

The basic syntax is as follows:

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

Example:
Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. Following example updates SALARY by 0.25 times in CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

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

This would impact two rows and finally CUSTOMERS table would have the following records:

```
+    +          +       +          +           +
| ID | NAME     | AGE | ADDRESS       | SALARY  |
                -----+----------
+    +----------  +              --   +----------        +
| 1  | Ramesh   | 35 | Ahmedabad |      125.00 |
| 2  | Khilan   | 25 | Delhi       |  1500.00 |
| 3  | kaushik  | 23 | Kota        |  2000.00 |
| 4  | Chaitali |  25 | Mumbai      | 6500.00 |
| 5  | Hardik   | 27 | Bhopal    |   2125.00 |
| 6  | Komal    |  22|MP           |  4500.00 |
| 7  | Muffy    | 24 | Indore      | 10000.00 |
```

V.30.4 Subqueries with the DELETE Statement:

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

The basic syntax is as follows:

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

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

table.

Following example deletes records from CUSTOMERS table for all the customers whose AGE is greater than or equal to 27:

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

This would impact two rows and finally CUSTOMERS table would have the following records:

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

Review Questions

1) Write a query to find the names (first_name, last_name) and salaries of the employees who have higher salary than the employee whose last_name='Bull'.

```
SELECT FIRST_NAME, LAST_NAME, SALARY
FROM employees
WHERE SALARY >
(SELECT salary FROM employees WHERE last_name = 'Bull');
```

2) Find the names (first_name, last_name) of all employees who works in the IT department.

```
SELECT first_name, last_name
FROM employees
WHERE department_id
IN (SELECT department_id FROM departments WHERE department_name='IT');
```

3) Find the names (first_name, last_name) of the employees who have a manager who works for a department based in United States.

4) Find the names (first_name, last_name) of the employees who are managers.

5) Find the names (first_name, last_name), salary of the employees whose salary is greater than the average salary.

```
SELECT first_name, last_name, salary FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

6) Find the names (first_name, last_name), salary of the employees whose salary is equal to the minimum salary for their job grade .

```
SELECT first_name, last_name, salary
FROM employees
WHERE employees.salary =
(SELECT min_salary FROM jobs WHERE employees.job_id = jobs.job_id);
```

7) Find the names (first_name, last_name), salary of the employees who earn more than the average salary and who works in any of the IT departments.

```
SELECT first_name, last_name, salary
FROM employees
WHERE department_id IN
(SELECT department_id FROM departments WHERE department_name LIKE 'IT%')
AND salary > (SELECT avg(salary) FROM employees);
```

V.31: Database Security

Database security means the way of protecting data against accidental or intentional loss in multi-users database environment.

V.31.1Creating a MySQL User and Granting All Privileges

Just as you start using MySQL, you will be given a username and a password. These initial credentials grant you 'root access'. The root user has full access to all the databases and tables within those databases.

But often times, you need to give the access of the database to someone else without granting them full control. For instance, companies who hire developers to maintain databases, but do not want to give them the ability to delete or modify any sensitive information, will often give them credentials of a non-root user. This way, the company can keep control of what their developers can and cannot do with the data.

Creating a new user is fairly simple in MySQL. We will show you how to create a new MySQL user, and how to grant them all privileges of your database. It is not wise, in a practical sense, to give full reign to a non-root user, but it is a good entry-point to learn about user privileges. To create a new user, follow these steps:

1. Access command line and enter MySQL server:

   mysql

2. Execute the following command:

   CREATE USER 'non-root'@'localhost' IDENTIFIED BY '123';

   Note: In this command, ‘non-root’ is the name we’ve given to our new user. And ‘123’ is the password for this user. You can replace these values with your own, inside the quotation marks.

3. Simply creating this new user isn’t enough. You have to grant them privileges. To grant the newly created user all privileges of the database, execute the following command:

   GRANT ALL PRIVILEGES ON * . * TO 'non-root'@'localhost';

4. For changes to take effect immediately flush the privileges by typing in the following command:

   FLUSH PRIVILEGES;

And that’s it! Your new user has the same access to the database as the root user.

Step 2 – Granting Specific Privileges for a MySQL User

As stated before, it is not smart to grant root level access to a non-root user. Most of the time, you will want to give different levels of access to different kinds of users. MySQL makes it a breeze granting these privileges, by the following simple command:

GRANT [permission type] ON [database name].[table name] TO ‘non-root’@'localhost’;

You simply have to replace the ‘permission type’ value with the kind of permission you want to grant the new user. You also have to specify database and table names, which gives the root user fine-grain control over granting privileges. Similar to the example above, ‘non-root’ is the username, so you’re free to replace it with your own choice. MySQL has quite a few permission types, some of which are described below:

- **CREATE** – Enables users to create databases/tables
- **SELECT** – Enables users to retrieve data
- **INSERT** – Enables users to add new entries in tables
- **UPDATE** – Enables users to modify existing entries in tables
- **DELETE** – Enables users to delete table entries
- **DROP** – Enables users to delete entire databases/tables

To use any of these options, simply replace **[permission type]** with the appropriate keyword. To apply multiple privileges, separate them with a command like this. For example, we can assign **CREATE** and **SELECT** privileges for our non-root MySQL user with this command:

GRANT CREATE, SELECT ON * . * TO 'non-root'@'localhost';

Remark: Of course, you might face a situation where you need to revoke given privileges from a user. You can do this by using the following command:

REVOKE [permission type] ON [database name].[table name] FROM 'non-root'@'localhost';

For example, to revoke all privileges for our non-root user we should use:

REVOKE ALL PRIVILEGES ON *.* FROM 'non-root'@'localhost';

Finally, you can entirely delete an existing user by using the following command:

DROP USER 'non-root'@'localhost';
Remember, to execute any of these commands, you need to have root access. Also, be sure to execute **FLUSH PRIVILEGES** after any changes made to the privileges.

V.31.2 Backup and Recovery

Backup a single database to a plain text file, containing the sql commands to restore the tables and their data

```
mysqldump -u [user] -p [database_name] > [filename].sql
```

Backup a single database to a gzipped version of the sql file

```
mysqldump -u [user] -p [database_name] | gzip > [file_name].sql.gz
```

Backup a single database to a bz2 compressed sql file.

```
mysqldump -u [user] -p [database_name] | bzip2 > [file_name].sql.bz2
```

You can usually upload gz, and bz2 files directly to a database using PHPMyAdmin, so is a good idea to have the outpud compressed.

Backup more than one database

You can backup more than one database at the same time.

```
mysqldump -u [user] -p –databases [database_name_1] [database_name_2] [database_name_n] > [filename].sql
```

You can use the same options as with the single database to compress the mysqldump command output.

Backup of all databases

```
mysqldump -u [user] -p –all-databases > [file_name].sql
```

This will dump all databases to a single file, and you can use that file to restore all databases at once. This is very useful to move your databases from one server to another.

Restore

Using the dump file, it is possible to restore the database with all its tables to a new MySQL server.

Create the database

```
mysql -u [user] -p
```

The [user]field in this case will usually be root. At the mysql> root.

```
create database [database_name];
```

Create a user for that database, it is actually not needed, but it is a good security measure.

```
grant all privileges on [database_name].* to [new_user]@[hostname] identified by [new_user_password];

exit;
```

Once again on the Linux command line prompt.

Restore database dump file

```
mysql -u [new_user] -p [database_name] < [file_name].sql
```

Review questions

Part NO1

Create Table

You have just started a new company. It is time to hire some employees. You will need to create a table that will contain the following information about your new employees: firstname, lastname, title, age, and salary.

Insert statement

It is time to insert data into your new employee table.

Your first three employees are the following:

Jonie	Weber,	Secretary,	28,	19500.00
Potsy	Weber,	Programmer,	32,	45300.00

Dirk Smith, Programmer II, 45, 75020.00

Enter these employees into your table first, and then insert at least 5 more of your own list of employees in the table.

Select Records

1. Select all columns for everyone in your employee table.
2. Select all columns for everyone with a salary over 30000.
3. Select first and last names for everyone that's under 30 years old.
4. Select first name, last name, and salary for anyone with "Programmer" in their title.
5. Select all columns for everyone whose last name contains "ebe".
6. Select the first name for everyone whose first name equals "Potsy".
7. Select all columns for everyone over 80 years old.
8. Select all columns for everyone whose last name ends in "ith".

Updating Records

1. Jonie Weber just got married to Bob Williams. She has requested that her last name be updated to Weber-Williams.
2. Dirk Smith's birthday is today, add 1 to his age.
3. All secretaries are now called "Administrative Assistant". Update all titles accordingly.
4. Everyone that's making under 30000 are to receive a 3500 a year raise.

5. Everyone that's making over 33500 are to receive a 4500 a year raise.
6. All "Programmer II" titles are now promoted to "Programmer III".
7. All "Programmer" titles are now promoted to "Programmer II".

Deleting Records

1. Jonie Weber-Williams just quit, remove her record from the table:
2. It's time for budget cuts. Remove all employees who are making over 70000 dollars.

Part N02

Remark: All sql commands you type in mysql must be copied to the notepad file that you have created on desktop and name it your registration number (Link mysql and notepad file).

1. Create a user whose name is yourfirstname@localhost with “abc123” as password.

2. After creating, connect yourfirstname@localhost to the localhost and create database whose name is company.

3. You have just started a new company. It is time to hire some employees. You will need to create a table whose name is “empinfo” that will contain the following information about your new employees: First Name, LastName, ID, Age, Salary, city and state and enter the following sample data by using SQL Interpreter.

No	FirstName	LastName	ID	Age	Salary	city	state
1	John	Jones	9998	45	20000 $	Payson	Arizona
2	Mary	Joe	9998	25	35000 $	Payson	Arizona
3	Eric	Edwards	8823	32	40000 $	San Diego	California
4	Mary Ann	Edwards	8823	32	30000 $	San Diego	California
5	Ginger	Howell	9800	42	40000 $	cottonwood	Arizona
6	Sebastian	smith	9200	23	25000 $	cottonwood	Arizona
7	Gus	Gray	2232	35	40000 $	cottonwood	Arizona

1. Write SQL command to create another table whose name is managers with the same attributes as empinfo table.
2. Write SQL command to delete last record of empinfo.
3. Write SQL command to save the database changes.
4. Write SQL command to insert two first records into managers table by using sql clone table concept.
5. The user “yourfirstname@localhost” can create, alter, select, update, insert, delete , and drop are permitted on empinfo table but these are totally prohibited on managers table.
6. Write a sub query to find the names (First Name, Last Name) and salary of the employees who have higher salary than the employee whose Last Name='joe'.

7. Write a sub query to find the names (First Name, Last Name) of all employees who live in the Payson city.
8. Write a sub query to Find the names (FirstName, LastName), and salary of the employees whose salary is than the average salary.
9. Write a query to list the number of states available in the empinfo table.
10. Write a query to get the difference between the highest and lowest salaries of empinfo table.
11. Write a query to get the No and the total salary payable in each city of empinfo table.
12. Write a query to get the highest, lowest, sum, and average salary of all employees of empinfo table.
13. Write SQL command to undo all database changes.
14. Create a folder and name it backup on local disk C, and then write SQL command to backup company database on the above predetermined path (filename of backup, you should name it "exercise.sql").
15. Write a SQL command to drop company database while root user is directly connected to the local host.
16. Write a SQL command to restore exercises.sql file while root user is still connected to the local host (database name must be named **company**).

REFERENCES

DOCUMENT REFERENCES

- SQL Server Security Distilled 2nd Edition By **Morris Lewis** (March 21, 2004)
- SQL Server Query Performance Tuning Distilled 2nd Edition By **Sajal Dam** (October 19, 2004)
- Pro SQL Server 2012 Integration Services by Francis Rodriguez, Michael Coles, David Dye (june 26,2012)
- Beginning MySQL Database Design and Optimization By **Chad Russell** , **Jon Stephens** (October 27, 2004)

ELECTRONIC REFERENCES

http://www.apress.com/databases/mysql

http://www.tutorialspoint.com/sql/sql_tutorial.pdf

http://portal.aauj.edu/portal_resources/downloads/database/microsoft_sqlL_server_blackbook.pdf