

**Database Management System**

**(DBMS)**

**LABORATORY WORKBOOK**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***CONTENTS***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S. No** | **Object of Experiments** | **Remarks** | **Date** | **Signature** |
| 1 | The Purpose of this Lab is to get familiar with the File System & File Handling |  |  |  |
| 2 | The Purpose of this Lab is to get familiar with the Relational DBMS Architecture & its Concepts |  |  |  |
| 3 | The Purpose of this Lab is to introduce Entity Relationship Diagram & its Notations |  |  |  |
| 4 | The Purpose of this Lab is to make Entity Relationship Diagram by using Case Studies |  |  |  |
| 5 | The Purpose of this Lab is to Introduce the DDL (Data Definition Language) which includes Create, Drop, Alter & Truncate Statements |  |  |  |
| 6 | The Purpose of this Lab is to Introduce the DML (Data Manipulation Language) Which includes Insert, Update & Delete Statements |  |  |  |
| 7 | The Purpose of this Lab is to Introduce the DCL (Data Control Language) Which includes Grant & Revoke Statements |  |  |  |
| 8 | The Purpose of this Lab is to get familiar with basic Select Queries which includes Selection & Projection Queries & Some Built-in Functions |  |  |  |
| 9 | The Purpose of this Lab is to get familiar the Tables Joins Which Includes Inner, Outer & Natural Joins |  |  |  |
| 10 | The Purpose of this Lab is to get familiar that how to use Groups & Having Clauses with Aggregates Functions |  |  |  |
| 11 | The Purpose of this Lab is to get familiar with the Analytical Queries Which includes Rollup, Cube & Top- N (Limits) Queries |  |  |  |
| 12 | The Purpose of this Lab is to get familiar with Stored procedures in MySQL. |  |  |  |
| 13 | The Purpose of this Lab is to make Normalized ERD – by using Case Studies |  |  |  |
| 14 | The Purpose of this Lab is to get familiar with the SQL Execution Plan and Performance Tuning |  |  |  |
| 15 | The Purpose of this Lab is to get familiar with the Database Administration that how we Install, Configure, Import & Export Databases |  |  |  |



**LAB # 01**

**The Purpose of this Lab is to get familiar with the File System & File Handling**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**File System & File Handling Experiment**

**01**

**Objective**

**The Purpose of this Lab is to get familiar with the File System & File Handling**

**Theory**

**Unit 1.1**

**File System**

A file system is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk. The word is also used to refer to a partition or disk that is used to store the files or the type of the file system. Thus, one might say ``I have two file systems'' meaning one has two partitions on which one stores files, or that one is using the ``extended file system'', meaning the type of the file system.

The difference between a disk or partition and the file system it contains is important. A few programs (including, reasonably enough, programs that create file systems) operate directly on the raw sectors of a disk or partition; if there is an existing file system there it will be destroyed or seriously corrupted. Most programs operate on a file system, and therefore won't work on a partition that doesn't contain one (or that contains one of the wrong type).

Before a partition or disk can be used as a file system, it needs to be initialized, and the bookkeeping data structures need to be written to the disk. This process is called making a file system.

**Block Size**

The block size specifies size that the file system will use to read and write data. Larger block sizes will help improve disk I/O performance when using large files, such as databases. This happens because the disk can read or write data for a longer period of time before having to search for the next block.

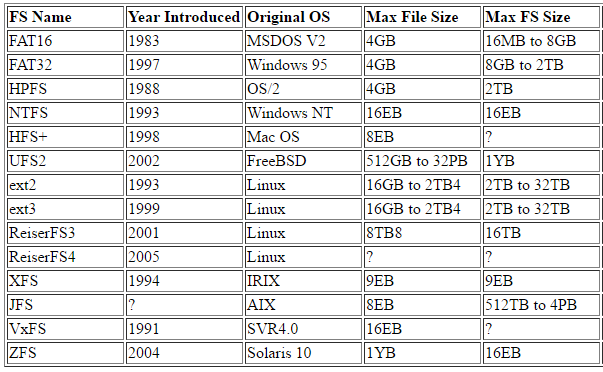
On the downside, if you are going to have a lot of smaller files on that file system, like the /etc, there the potential for a lot of wasted disk space.

For example, if you set your block size to 4096, or 4K, and you create a file that is 256 bytes in size, it will still consume 4K of space on your hard drive. For one file that may seem trivial, but when your file system contains hundreds or thousands of files, this can add up.

Block size can also effect the maximum supported file size on some file systems. This is because many modern file system are limited not by block size or file size, but by the number of blocks. Therefore you would be using a "block size \* max # of blocks = max block size" formula.

**Unit 1.2**

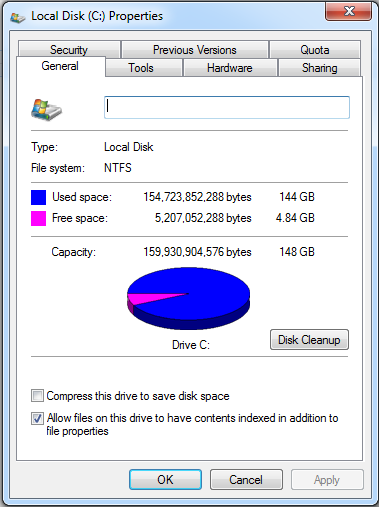
**File System comparison**

****

**Unit 1.3**

**How we can check which the file system in MS Windows?**

*Go to >*Computer*, right-click on the drive you wish to examine, and select >*Properties*from the menu. The >*General*tab shows the >*File system*.*



**Feedback**

|  |
| --- |
|  |



**LAB # 02**

**The Purpose of this Lab is to get familiar with the Relational DBMS Architecture & its Concepts**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Relational DBMS Architecture Experiment**

**02**

**Objective**

**The Purpose of this Lab is to get familiar with the Relational DBMS Architecture & its Concepts**

**Theory**

**Unit 2.1**

**RDBMS: Overview**

* A relational database management system (RDBMS):
  + Manages data according to the relational model
  + Organizes and stores data in the form of tables
  + Example: A bank database has tables for currency data, customer information, account information, and so on.
* A database is a collection of data placed in tables.
  + Synonymous with “schema”
* The relational model
  + Data logically organized as tables and relations
  + An improvement over the hierarchical database model, which has the following problems:
    - Requires additional steps to coordinate the data and programs
    - Difficult to develop applications and perform required operations
* In the relational model, the fundamental structure to organize data is the “relation,” which is where it gets its name. For most purposes, the term “table” is interchangeable with the more abstract term “relation.” The term “table” is used in the remainder of this course.
* “Database” is MySQL terminology. Other RDBMSs use the term “schema.”
* For example, a database for a bank might contain a table to store currency data, a table to store customer data, and a table for its accounts. These tables are obviously related: a customer can have one or more accounts, and an account can express its balance in a particular currency. However, the existence of these relationships is not a defining characteristic of a relational database. The only reason for a DBMS to be called **relational** is that it stores its data in tables, regardless of whether or not these tables are related to one another.
* Continuing with the bank example, a hierarchical representation would require a list of customers and, within each customer entry, a list of bank accounts, which contain a currency. To list all used currencies, you must first examine all the customers and then, for each customer, you must look at all accounts. This shows a case for using the relational model over the hierarchical model, which requires many additional steps to coordinate the data and programs that work with a database. The hierarchical model makes it incredibly time consuming to develop applications and to perform many required operations.

**Spreadsheet versus Database**

* + Spreadsheets are flexible and provide more user interface features:
    - Allow any data to be entered into a cell
    - Convert data into graphs
    - Format directly into printable reports with header and footer
  + Databases are much more powerful than spreadsheets due to data query and manipulation capabilities:
    - Retrieve all records that match certain criteria
    - Cross-reference records between tables
    - Update records in bulk
    - Perform complex aggregate calculations

**Unit 2.2**

**Entities and Relationships**

* **Entities:** Things in the real world that you store information about in a database.
  + Tables typically store data representing one type of entity.
  + Example:
    - A **bank** database has a **customer** table specifically for customer information.
    - The **customer** table can contain entities/columns such as the customer name, account number, and contact information.
* **Relationships:** Links between entities that have something to do with each other.
  + Example:
    - The bank customer name has a relationship to the customer account number and contact information.

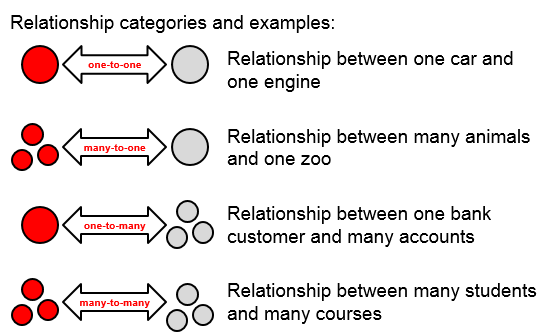
There can also be relationships between separate tables (for example, **customer** to **accounts**).

You cannot literally store things in a database, nor any relationships they might have. What you can store in a database is data. Storing information about things such as customers, bank accounts and currencies in our database, means that you store data in your database that represents these real-world things.

In relational databases, one table is typically used to store data that represents entities of one particular kind. As in the **bank** example, the **customers** table is used to store items of data, each of which represents exactly one entity of the "customer" kind.

**Relationship Categories**

Relationship categories and examples:



Relationships can be categorized according to how many entities can possibly be involved at either end of the relationship.

In the slide example of a **one-to-one** relationship, a car has one engine, and the engine can belong to only one car. There is a one-to-one relationship between the car and the engine. Often these types of relationships are put in the same table, but not always.

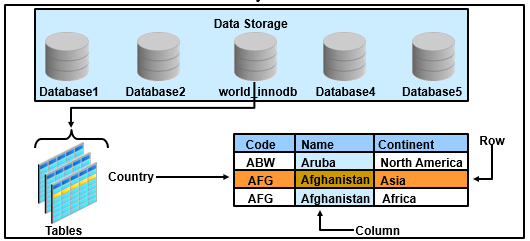
In the slide example of a **one-to-many** relationship, an individual customer of a bank can have more than one account. Note that this allows for customers to have zero, one, or more bank accounts. The word *many* refer to the maximum number of possible bank accounts for each customer. Alternatively, a bank account is tied to a single customer only. That is why this is a one-to-many relationship.

There is a **many-to-many** relationship between students and courses. A student can attend more than one course and a course can be attended by many students.

**Unit 2.3**

**RDBMS Database Structure**

* Two-dimensional tables, columns, and rows:
  + An ordered set of columns (1 or more)
  + An unordered set of rows (0 or more)
* Each table row has exactly one value for each column.



**Row-Column Relationship**

* **Rows** are a collection of values that describe an entity (such as a bank account).
* **Columns** are a collection of similar data among rows (such as customer surnames). Each column has a name and a data type.
* The intersection of row and column contains individual data items called **values**. Values are always atomic, that is, each position in a table may contain only one datum (piece of data). If you have ever used a spreadsheet, you have used a table of data.

It is common to present data in a simple two-dimensional table form. The relational model stipulates that a relation must have unique **tuples**—no two entities can be identical or you would not be able to distinguish one from the other. There must be some combination of columns (possibly, all columns) called a **key**, whose values uniquely identify each tuple. Most RDBMSs are less strict. Virtually all of them allow a table to contain duplicate rows, and allow a table to exist without a key.

**Note:** In SQL, tables are not required to have primary keys. In other words, the requirement of the relational model is not enforced in SQL. However, it is enforced as soon as a primary key is added to the table. A detailed discussion of primary keys, and how to create them, is provided later in the course.

**Exercise**

**Question 1**

Modern School of higher education has around 2000 students and three departments. The students belong to various departments in the school. Every department has a Head of Department along with teachers. The head of department manages his departmental teachers and the students belonging to his department. The head of department also teaches the students. All the head of departments report to the Principal of the school. The Principal manages the departments and also teaches the students. Handling the administrative staff of the school is also the responsibility of the Principal. Every individual, except the students, are the employees of the school

**Problem**

* Identify the entity
* Identify the entity’s attribute
* Identify the primary and foreign Keys
* Identify relationship between Entities
* Identify the Cardinality constraints

**Question 2**

1. a professor teaches zero, one or many classes and a class is taught by one professor
2. a course may generate zero, one or many classes and a class comes from one course
3. a class is held in one room but a room has many classes

**Feedback**

|  |
| --- |
|  |



**LAB # 03**

**The Purpose of this Lab is to introduce Entity Relationship Diagram & its Notations**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Entity Relationship Diagram – Introduction Experiment**

**03**

**Objective**

**The Purpose of this Lab is to introduce Entity Relationship Diagram & its Notations**

**Theory**

**Unit 3.1**

**Notations**

**Entities Notation**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Shape Name** | **Description** |
| ERD Symbols and Meaning - Entity | Entity | An entity is represented by a rectangle which contains the entity’s name. |
| ERD Symbols and Meaning - Weak Entity | Weak Entity | An entity that cannot be uniquely identified by its attributes alone. The existence of a weak entity is dependent upon another entity called the owner entity. The weak entity’s identifier is a combination of the identifier of the owner entity and the partial key of the weak entity. |
| ERD Symbols and Meaning - Associative Entity | Associative Entity | An entity used in a many-to-many relationship (represents an extra table). All relationships for the associative entity should be many |

**Attributes Notation**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Shape Name** | **Description** |
| ERD Symbols and Meaning - Attribute | Attribute | Each attribute is represented by an oval containing attribute’s name |
| ERD Symbols and Meaning - Key attribute | Key attribute | An attribute that uniquely identifies a particular entity. The name of a key attribute is underscored. |
| ERD Symbols and Meaning - Multivalue attribute | Multivalued attribute | An attribute that can have many values (there are many distinct values entered for it in the same column of the table). Multivalued attribute is depicted by a dual oval. |
| ERD Symbols and Meaning - Derived attribute | Derived attribute | An attribute whose value is calculated (derived) from other attributes. The derived attribute may or may not be physically stored in the database. In the Chen notation, this attribute is represented by dashed oval. |

**Relationship Notation**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Shape Name** | **Description** |
| ERD Symbols and Meaning - Relationship | Strong relationship | A relationship where entity is existence-independent of other entities, and PK of Child doesn’t contain PK component of Parent Entity. A strong relationship is represented by a single rhombus |
| ERD Symbols and Meaning - Identifying Relationship | Weak (identifying) relationship | A relationship where Child entity is existence-dependent on parent, and PK of Child Entity contains PK component of Parent Entity. This relationship is represented by a double rhombus. |

ER Model is represented by means of an ER diagram. Any object, for example, entities, attributes of an entity, relationship sets, and attributes of relationship sets, can be represented with the help of an ER diagram.

**Unit 3.2**

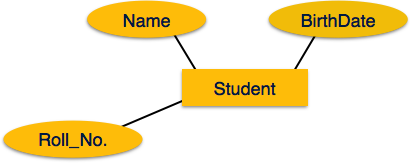
**Entity**

Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.

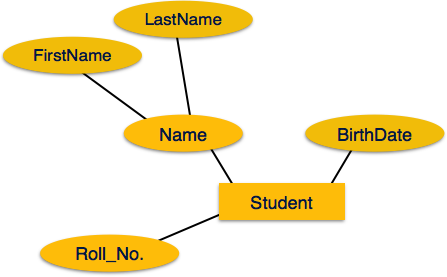
Entities in a school database

**Attributes**

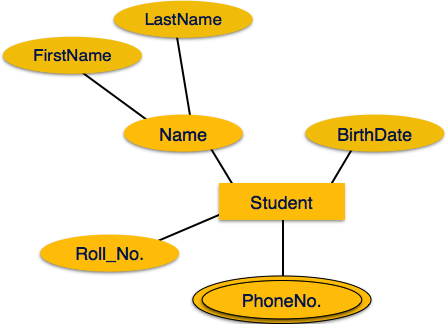
Attributes are the properties of entities. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).



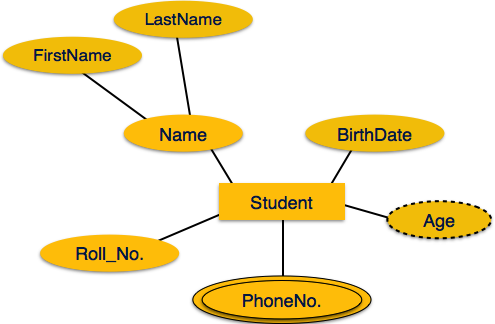
If the attributes are **composite**, they are further divided in a tree like structure. Every node is then connected to its attribute. That is, composite attributes are represented by ellipses that are connected with an ellipse.



**Multivalued** attributes are depicted by double ellipse.



**Derived** attributes are depicted by dashed ellipse.



**Unit 3.3**

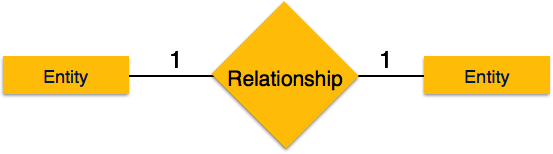
**Relationship**

Relationships are represented by diamond-shaped box. Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line.

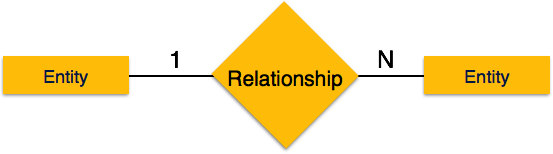
**Binary Relationship and Cardinality**

A relationship where two entities are participating is called a **binary relationship**. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

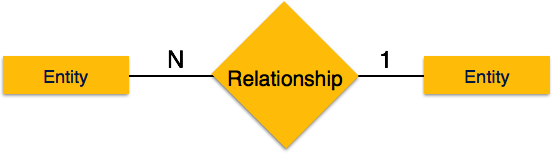
* **One-to-one** − when only one instance of an entity is associated with the relationship, it is marked as '1:1'. The following image reflects that only one instance of each entity should be associated with the relationship. It depicts one-to-one relationship.



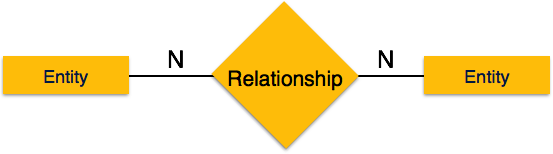
* **One-to-many** − When more than one instance of an entity is associated with a relationship, it is marked as '1:N'. The following image reflects that only one instance of entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts one-to-many relationship.



* **Many-to-one** − when more than one instance of entity is associated with the relationship, it is marked as 'N:1'. The following image reflects that more than one instance of an entity on the left and only one instance of an entity on the right can be associated with the relationship. It depicts many-to-one relationship.

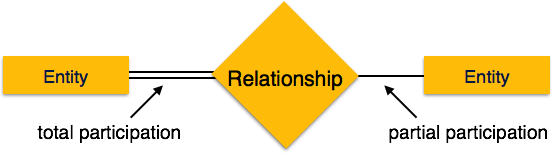


* **Many-to-many** − The following image reflects that more than one instance of an entity on the left and more than one instance of an entity on the right can be associated with the relationship. It depicts many-to-many relationship.



**Participation Constraints**

* **Total Participation** − Each entity is involved in the relationship. Total participation is represented by double lines.
* **Partial participation** − Not all entities are involved in the relationship. Partial participation is represented by single lines.



**Exercise**

**Question 1**

Modern School of higher education has around 2000 students and three departments. The students belong to various departments in the school. Every department has a Head of Department along with teachers. The head of department manages his departmental teachers and the students belonging to his department. The head of department also teaches the students. All the head of departments report to the Principal of the school. The Principal manages the departments and also teaches the students. Handling the administrative staff of the school is also the responsibility of the Principal. Every individual, except the students, are the employees of the school

* Identify the entity
* Identify the entity’s attribute
* Identify the primary and foreign Keys
* Identify relationship between Entities
* Identify the Cardinality constraints
* Draw ERD

**Question 2**

1. an invoice is written by one sales rep but a sales rep writes many invoices
2. a vendor sells many products but a product is bought from one vendor
3. an invoice has one or many products and a product is found on zero, one or many invoices

* Identify the entity
* Identify the entity’s attribute
* Identify the primary and foreign Keys
* Identify relationship between Entities
* Identify the Cardinality constraints
* Draw ERD

**Feedback**

|  |
| --- |
|  |



**LAB # 04**

**The Purpose of this Lab is to make Entity Relationship Diagram by using Case Studies**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Entity Relationship Diagram – Case Study Experiment**

**04**

**Objective**

**The Purpose of this Lab is to make Entity Relationship Diagram by using Case Studies**

**Theory**

**CASE - STUDY**

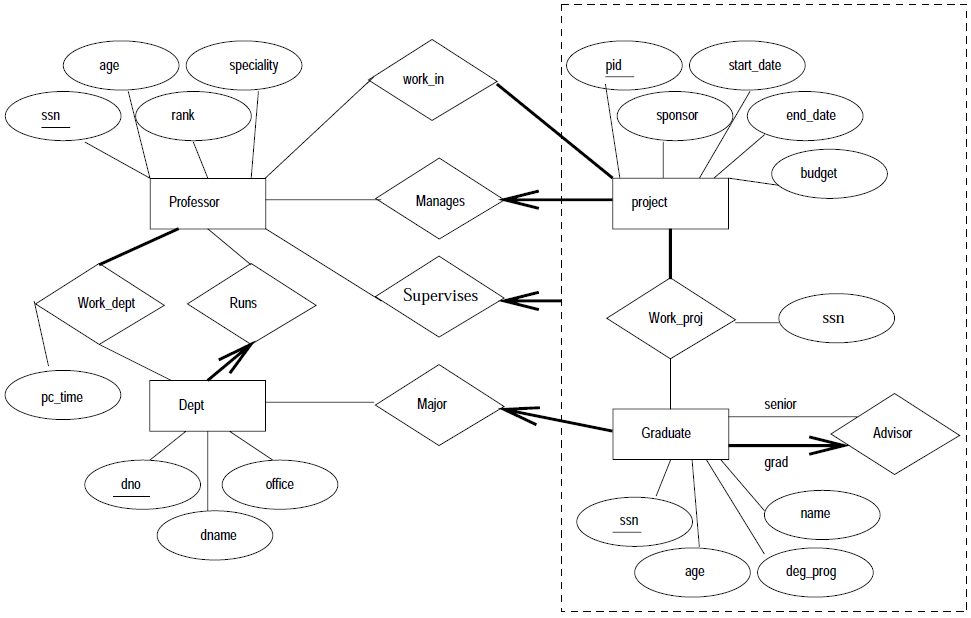
Consider the following information about a university database:

* Professors have an SSN, a name, an age, a rank, and a research specialty.
* Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
* Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
* Each project is managed by one professor (known as the project’s principal investigator).
* Each project is worked on by one or more professors (known as the project’s co-investigators).
* Professors can manage and/or work on multiple projects.
* Each project is worked on by one or more graduate students (known as the project’s research assistants).
* When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
* Departments have a department number, a department name, and a main office.
* Departments have a professor (known as the chairman) who runs the department.
* Professor’s work in one or more departments and for each department that they work in, a time percentage is associated with their job.
* Graduate students have one major department in which they are working on their degree.
* Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university.

Use only the basic ER model here; that is, entities, relationships, and attributes. Be

Sure to indicate any key and participation constraints.



**Exercise**

**CASE - STUDY -II**

The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicine if you design its database. Given the rising cost of health care, you agree. Here’s the information that you gather:

* Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
* Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
* Each pharmaceutical company is identified by name and has a phone number.
* For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.
* Each pharmacy has a name, address, and phone number.
* Every patient has a primary physician. Every doctor has at least one patient.
* Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.
* Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors.
* Each prescription has a date and a quantity associated with it. You can assume that, if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.
* Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.
* Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

Draw an ER diagram that captures the preceding information. Identify any constraints not captured by the ER diagram.

How would your design change if each drug must be sold at a fixed price by all pharmacies?

How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored?

**Feedback**

|  |
| --- |
|  |



**LAB # 05**

**The Purpose of this Lab is to introduce the DDL (Data Definition Language) which includes Create, Drop, and Alter & Truncate Statements**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Introduction to DDL Experiment**

**05**

**Objective**

**The Purpose of this Lab is to introduce the DDL (Data Definition Language) which includes Create, Drop, and Alter & Truncate Statements**

**Theory**

**Unit 5.1**

**Data Definition Language: SQL Statements**

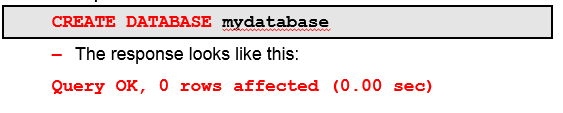
The DDL SQL statements are used to define database and table data. The statements shown in the slide are followed by options to specify data operations.

**Creating a Database**

* General syntax for creating a database:



* **CREATE SCHEMA** is a synonym for **CREATE DATABASE**.
* Example:



When your database design is complete, create the actual database by using the MySQL statement **CREATE DATABASE**. It is important for you to name the database according to its purpose. After the database is created, you can add the tables.

You can add the **IF NOT EXISTS** option statement. An error occurs if the database exists and you did not specify **IF NOT EXISTS**.

For more information about creating a database, see the *MySQL Reference Manual* at <http://dev.mysql.com/doc/refman/5.5/en/create-database.html>.

**MySQL Naming Conventions**

* Database and table names are generally not   
  case-sensitive, but this depends on the file system.
* You cannot use certain characters, including ASCII(**0**), ASCII(**255**), **/**, **\**, and **.**
* Names cannot have more than 64 characters.
* You can use reserved words and special characters in quotation marks.

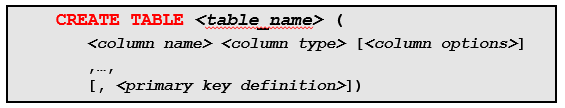
MySQL in general does not consider names (databases and tables) to be case-sensitive. However, it does treat them as case-sensitive if the file system on which the database resides is itself case-sensitive.

Table names can be (or contain) reserved words as long as the name is quoted (such as a table named '**my\_table**' or '**$**'), but this practice is discouraged.

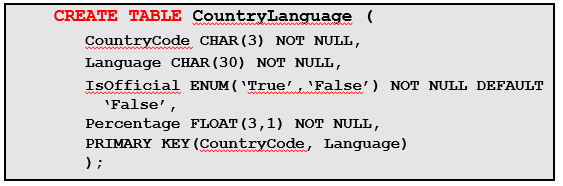
**Unit 5.2**

**Creating a Table**

* General syntax for creating a table:



* Example:



After the database structure is designed and the database has been created, you can add the individual tables. You must use accurate assignment of data types and options, then you can add the tables from the design plan to the database. Table and column options are covered later in this lesson.

Creating tables is a complex topic due to the process of choosing how to create the table and the syntax, which consists of many elements. The statement syntax shown in the slide includes the column and table options.

A line-by-line description of the **CREATE TABLE** example:

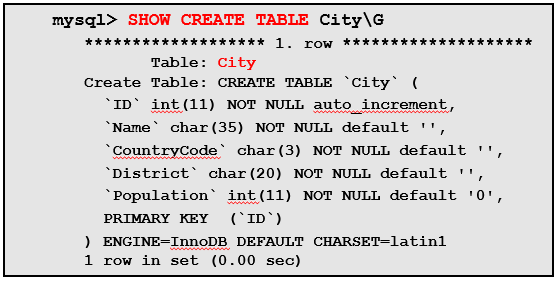
1. Creates a table called **CountryLanguage**"**(**" is the beginning of the table structure definition that ends with "**)**".
2. The column named **CountryCode** is assigned the data type of **CHAR** and maximum length of **3** characters that cannot accept **NULL**. The comma at the end of the line indicates that you are continuing to define columns, or set a primary key.
3. The column named **Language** is assigned the **CHAR** data type and a length of **30**, also with no **NULL**s.
4. The column named **IsOffical** is assigned the **ENUM** data type (with the value **'True'** or **'False'**). The addition of **NOT NULL** means that each row must contain one of the specified values (as indicated in single quotation marks). If the field is not provided, it is set to '**False**' by the phrase **DEFAULT 'False'**.
5. The column named **Percentage** is assigned the **FLOAT** data type. It contains three digits, including one to the right of the decimal point. This column cannot accept **NULL**.
6. Defines the **PRIMARY KEY**, the type of key, and to what columns it applies. In this case, a combination of both **CountryCode** and **Language** columns define a unique identifier for each row.
7. "**)**" is the closing of the table structure definition. “**;**” means execute this statement (using the MySQL command-line client) and create the **CountryLanguage** table.

**Note:** You can also use the **CREATE TABLE** statement to create temporary tables. Temporary tables are covered in detail later in the course.

For more information about creating a table, see the *MySQL Reference Manual* at <http://dev.mysql.com/doc/refman/5.5/en/create-table.html>.

**Showing How a Table Was Created**

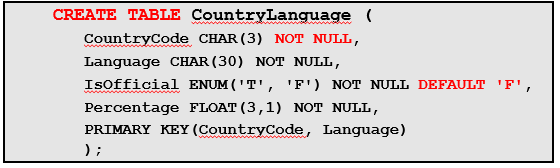
* View the exact statement used to create a table.
* Example:



The **SHOW CREATE TABLE** statement produces a statement that can create a table with the same properties as the table specified. This can help you to understand the structure of a table. You can also use it as a basis for the syntax to create a new table. For example, to show the **CREATE TABLE** statement used to create the **City** table, execute the statement shown in the example in the slide.

**Column Options**

* Each table must have at least one column.
* You can add options to the **CREATE TABLE** statement, such as:
  + **NULL**– **DEFAULT**
  + **NOT NULL**– **AUTO\_INCREMENT**
* Example:



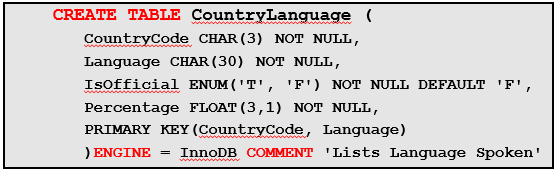
The **CREATE TABLE** column options modify how MySQL handles the associated column. Some common column options:

* **NULL**:Is the default option that allows storing of **NULL** values
* **NOT NULL**
  + Disallows **NULL**
  + Saves response time and, in some cases, disk space
  + Is a necessity for a **PRIMARY KEY**
  + **NULL** and **NOT NULL** are mutually exclusive.
* **DEFAULT *<value>***:If no value is supplied by the user, this specified value is stored.
* **AUTO\_INCREMENT**
  + A "running number" option for indexed columns
  + Used for **INTEGER** data type columns only
  + Using **NULL**, or not specifying a value for the column, causes an automatic insert of the next available number in sequence
  + Must be indexed
  + Only one permitted per table

**Note:** Indexes are covered later in this lesson.

**Table Options**

* You can add options to the **CREATE TABLE** statement, such as:
  + **ENGINE**
  + **COMMENT**
  + **DEFAULT CHARACTER SET**
* Example:



The **CREATE TABLE**table options modify how MySQL handles the entire table. Some common table options:

* **ENGINE = *<storage\_engine\_name>*:** Indicates the storage engine to be used for the table
* **COMMENT:** Up to 60 characters of free form text
* **DEFAULT CHARACTER SET *<character set>*:** Specifies the default character set for the table

**Unit 5.3**

**Table Indexing**

* Original table data is not put in any specific order or location.
  + Server must scan all data to find relevant rows.
* Indexes assist in finding rows quickly and easily.
  + Only contain column information for locating rows
* Indexes are assigned when creating a table, or you can add them later.
* Composite indexes are created on multiple columns.
* “Key” is normally a synonym for “index.”

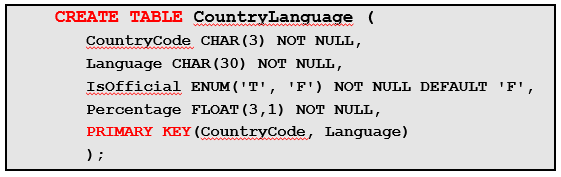
When you insert a row into a table, the database server does not attempt to put the data in any particular location within the table. Instead, the server simply places the data in any available location within the table. Tables can be very large, and as a table gets bigger, retrievals from it become slower. This is due to the fact that the location of the data is unknown, so the server must look at all the data to find the relevant rows.

An index (or key) in MySQL serves to assist in finding rows quickly, like an index at the back of a technical manual. Database indexes are used to locate rows in a table. Instead of containing all of the row data, an index contains only the columns used to locate the rows. It also contains information describing where the rows are physically located. Usually, you create all the indexes you need when you are creating tables.

You can create indexes on single columns or multiple columns (composite indexes). For example, the phone number for an individual in a phone book can be used as a single column index, whereas the last name and first name can be used to create a composite index.

**MySQL Indexing**

* Commonly used indexes:
  + **PRIMARY KEY:** Only one allowed
  + **UNIQUE:** Multiple allowed
* Indexes are optional and must be created by the user.
* Example:



The following are the commonly used indexes:

* **PRIMARY KEY**
  + Only one **PRIMARY KEY** is allowed per table.
  + It uniquely identifies a single row in the table.
  + No **NULL** values are permitted.
  + No duplicate values are permitted.
  + If a duplicate value is **INSERT** or **UPDATE** MySQL returns an error and the attempted operation is not performed.
* **UNIQUE**
  + The same rules as the **PRIMARY KEY**, with two exceptions:

Can have values stored as **NULL**

Can have multiple **UNIQUE** indexes in a table

* Each non-**NULL** value uniquely identifies a single row in a table.

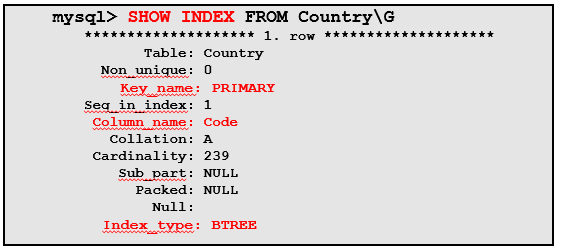
For more information about how MySQL uses indexes, see the *MySQL Reference Manual* at <http://dev.mysql.com/doc/refman/5.5/en/mysql-indexes.html>.

**Showing the Table Indexes**

* General syntax for displaying table indexes:



* Example:



The result of the **SHOW INDEX** statement example in the slide shows that there is a **PRIMARY** index on the **Code** column of the **Country** table, and an index type of **BTREE** table. The **Cardinality** column shows the number of distinct values that exist in the table: **239** in this case.

**Note:** Most indexes are stored in **B-trees**. Exceptions are indexes on spatial data types use **R-trees**, and MEMORY tables also support hash indexes.

For more information about the **SHOW INDEX** statement, see the *MySQL Reference Manual* at <http://dev.mysql.com/doc/refman/5.5/en/show-index.html>.

**Unit 5.4**

**Table Constraints**

* A constraint is a restriction placed on one or more column values of a table to actively enforce integrity rules.
* Constraints are implemented using indexes.
* Types of constraints:
  + **PRIMARY KEY**
  + **FOREIGN KEY**
  + **UNIQUE**
* MySQL generates indexes to enforce the above constraints.

A constraint is a restriction placed on one or more column values of a table to actively enforce integrity rules.

Constraints are implemented using the following indexes:

* **PRIMARY KEY:** Defines the columns that guarantee uniqueness within a table
* **FOREIGN KEY:** Restricts the columns to contain only values that match those in another table’s **PRIMARY KEY** or **UNIQUE** columns
* **UNIQUE:** The same as **PRIMARY KEY** except that the columns are allowed to be null-able, and there may be multiple **UNIQUE** constraints in one table

For example, if the MySQL server allowed you to modify the **world\_innodb** database by changing the **CountryCode** value in the **CountryLanguage** table without changing the corresponding **CountryCode** value in the **City** table, then you end up with rows that no longer point to valid country records (known as **orphaned rows**). With constraints in place (**FOREIGN KEY**, in this case), the server raises an error if an attempt is made to modify or delete data that is referenced by other tables, or it propagates the changes to other tables for you.

**Note:** Only the InnoDB storage engine supports **FOREIGN KEY** constraints. Other storage engines ignore these constraints in table definitions.

**DROP DATABASE Statement**

* Removes a database when it is no longer needed
  + The database can be empty or contain tables and data.
* Returns a row count, which is the number of tables deleted
* Must be used with extreme caution, because it does not have the “undo” feature
* Examples:



* Returns an **error** if the database does not exist



* + Returns a **warning** if the database does not exist

Use **DROP DATABASE** when you are sure that the database is no longer needed.

You must have the **DROP** privilege for each database.

**DROP DATABASE** does not require the database to be empty. Before dropping the database, MySQL removes any objects that it contains, such as tables, stored routines, and triggers.

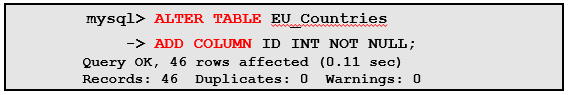
A successful **DROP DATABASE** returns a row count that indicates the number of tables dropped. (This actually is the number of **.frm**files removed, which amounts to the same thing.) You can check your current databases to make sure that the database is removed by using the **SHOW DATABASES** statement.

A database is represented by a directory under the data directory. The server deletes only files and directories that it can identify as having been created by itself (such as **.frm** files). It does not delete other files and directories. If you have put nontable files in that directory, those files are not deleted by the **DROP DATABASE** statement. This results in a failure to remove the database directory. In that case, the database continues to be listed by **SHOW DATABASES**. To correct this problem, you can manually remove the database directory and any files that it contains.

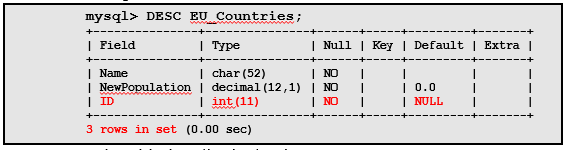
Any warning generated by **DROP DATABASE** can be displayed with the **SHOW WARNINGS** statement.

**Adding a Table Column**

* Use the **ALTER TABLE** statement with **ADD COLUMN**.
* Example:



* Changes the structure as follows:



* + **ID** is added as the last column.

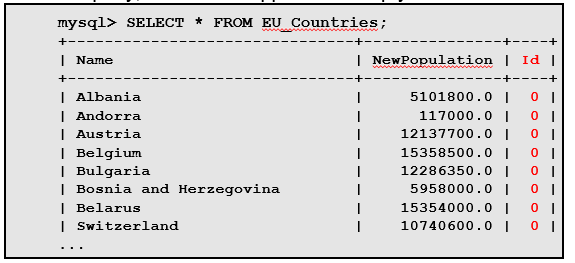
To add a new column (or remove a column), use the **ALTER TABLE** statement with the appropriate clause that specifies the column’s definition. A column definition uses the same syntax for **ALTER TABLE** as for **CREATE TABLE**. You can use **ALTER TABLE** to:

* Add or remove a column
* Add or remove an index
* Change the specification of a current column

Column names within a table must be unique, so you cannot add a column with the same name as one that already exists in the table. Also, column names are *not* case-sensitive, so if the table already contains a column named **ID**, you cannot add a new column using any of these names: **ID**, **id**, **Id**, or **iD**. They are all considered to be the same.

**Adding a Table Column**

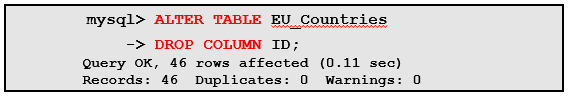
* Adding a column to a table populates the rows with **NULL**, the default value as specified, or the data type implicit default.
* In a query, the column appears as empty or zero:



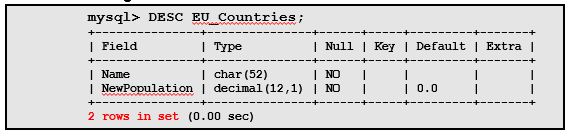
**Note:** The statement for populating a table with data is covered in detail later in the course.

**Removing a Table Column**

* Use the **ALTER TABLE** statement with **DROP COLUMN**.
* Example:



* Changes the structure as follows:



* + The **ID** column has been removed.

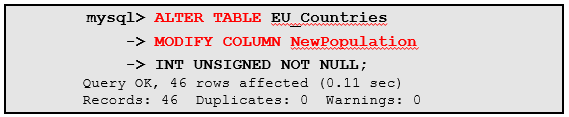
If a table contains only one column, you cannot drop the column. If you intend to remove the table, use **DROP TABLE** instead.

You should not remove a column from a table if it is a primary key. You cannot remove a column that is a foreign key from another table.

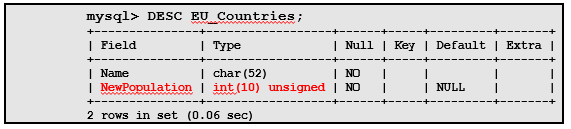
For more information about using **ALTER TABLE**, see the *MySQL Reference Manual* at: <http://dev.mysql.com/doc/refman/5.5/en/alter-table.html>.

**Modifying a Table Column**

* Use the **ALTER TABLE** statement with **MODIFY COLUMN**.
* Example:



* + Changes the structure as follows:



* + **NewPopulation** has been changed to the **INT** data type.

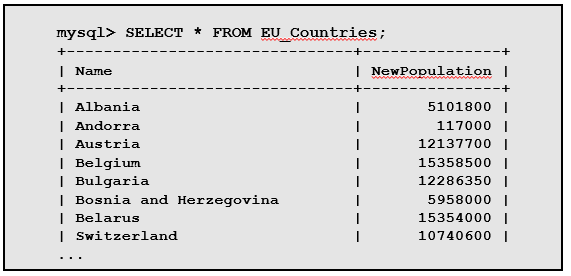
The example in the slide shows how you can change the **NewPopulation** column’s data type from **DECIMAL** to **INT**, to restrict the column value to whole numbers. You can also make the **UNSIGNED** column to disallow negative values.

If you also want to disallow **NULL** in the column, you must include the **NOT NULL** attribute in the column definition for **MODIFYCOLUMN**, even if you originally defined the column with **NOT NULL**. This is true for other column attributes as well. If you do not specify them explicitly, the new definition does not carry them over from the old definition.

You cannot modify a column from a table if it is a primary key and if a foreign key from another table references the column.

**Modifying a Table Column: Row Changes**

The previous **ALTER TABLE...MODIFY COLUMN** statement example changes the row contents of the table as follows:



**Note:** The statement for populating a table with data is covered in detail later in the course.

Anything that can be added using **CREATE TABLE** can be changed using **ALTER TABLE**.

For more information about using **ALTER TABLE**, see the *MySQL Reference Manual* at: <http://dev.mysql.com/doc/refman/5.5/en/alter-table.html>.

**Truncate a 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.

**TRUNCATE TABLE** EU\_Countries;

**truncate** command is different from **delete** command. delete command will delete all the rows from a table whereas truncate command re-initializes a table(like a newly created table).

**For e.g.** If you have a table with 10 rows and an auto\_increment primary key, if you use *delete* command to delete all the rows, it will delete all the rows, but will not initialize the primary key, hence if you will insert any row after using delete command, the auto\_increment primary key will start from 11. But in case of *truncate* command, primary key is re-initialized.

**Exercise**

Consider the following schema:

1. **Customer** (cust\_id, cust\_name).
2. **Product** (prod\_code ,prod\_name, unit\_price)
3. **Customer\_Order** (order\_code, order\_date, cust\_id)
4. **Order\_Item** (order\_code, prod\_code, num\_of\_units)
5. Develop DDL of in SQL
6. Add Column contact in Customer Table
7. Add Column company\_name in Product Table

**Feedback**

|  |
| --- |
|  |



**LAB # 06**

**The Purpose of this Lab is to introduce the DML (Data Manipulation Language) which includes Insert, Update & Delete Statements**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Introduction to DML Experiment**

**06**

**Objective**

**The Purpose of this Lab is to introduce the DML (Data Manipulation Language) which includes Insert, Update & Delete Statements**

**Theory**

**Unit 6.1**

**Data Manipulation Language: SQL Statements**

The DML SQL statements are used to manipulate database and table data. The statements shown in the slide are followed by options to specify data operations (such as table name).

**Manipulation of Table Row Data**

* It is common for databases to require modification:
  + This is due to changes in data usage situations.
  + Row data can be inserted, updated, replaced, or deleted.

Data manipulation can be unsafe. Take the following precautions with your data:

* + Do not grant users (including yourself) more permissions than required. For example, if you are running some ad hoc queries on a database for the finance department, do not use the MySQL **root** account. Instead, create a user who has permission to run only the needed **SELECT** queries and then log in as the user.
  + Keep daily backups of the data stored in MySQL.
  + Always make a backup before you make any major changes, or when using unfamiliar features.
  + For beginners, a useful startup option is the **safe updates** option. It can be turned on with the command-line client option **--safe-updates**, or by issuing **SET SQL\_SAFE\_UPDATES=1**. It is helpful for cases when you might have issued a **DELETE FROM tbl\_name** statement but forgotten the **WHERE** clause. Normally, such a statement deletes all rows from the table. With this option, you can delete rows *only* by specifying the key values that identify them. This helps prevent accidents.
  + Test queries and updates on a test table before running them on the real table.
  + Execute a **SELECT** statement with the same **WHERE** clause to verify that the right results are returned, before running a **DELETE** or an **UPDATE** statement.

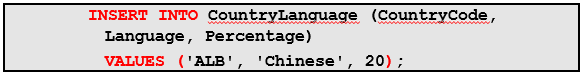
**Unit 6.2**

**INSERT Statement**

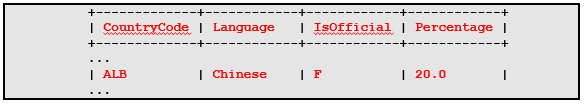
* After creating a table, you can populate it with row data.
* General syntax includes the **INTO** clause:



* Example of inserting a **singlerow**:



* + The content of the new row:



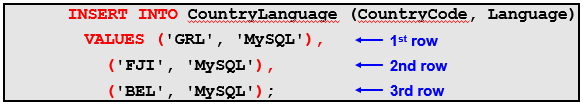
The syntax for **INSERT** uses separate column and value lists following the name of the table into which you add the single record. The number of columns and values must be the same.

In the **INSERT** example in the slide, you add **Chinese** as a language for **Albania**, with a percentage of **20**, and the default **F** (false) for the **IsOfficial** column is acceptable because it is not the official language of the country.

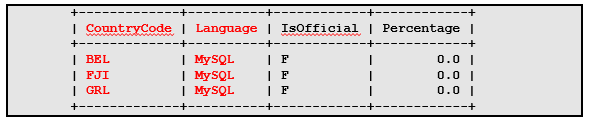
In the result for the example in the slide, the false (**F**) value for the **IsOfficial** column is the default value, which is used in the absence of a specific assignment.

**Using INSERT for Multiple Rows**

* Create multiple new rows in a table.
* Example:



* + The content of the new row:



In the **INSERT** example in the slide, you add **MySQL** as a language for **Greenland**, **Fiji**, and **Belgium**. However, you do not want to give it a percentage, and the default **F** (false) for the **IsOfficial** column is acceptable. The row data for each of the three rows is enclosed in parentheses.

The result for the example in the slide is achieved with the following query:

**SELECT \* FROM CountryLanguage**

**WHERE CountryCode IN ('GRL', 'FJI', 'BEL')**

**AND LANGUAGE = 'MYSQL';**

In the example result in the slide, the **F** (false) value for the **IsOfficial** column is the default value, which is used in the absence of a specific assignment.

**Note:** You must enclose all column values in single quotation marks (for string and temporal data types) in the **INSERT** statement.

**NULL condition with INSERT**

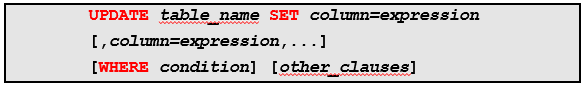
If values are *not* specified and columns have *no default* value:

* Sets to **NULL** if column accepts **NULL** values
* Sets to implicit default for the column data type if it does not accept **NULL**
* **AUTO\_INCREMENT** columns automatically generate the next sequence number

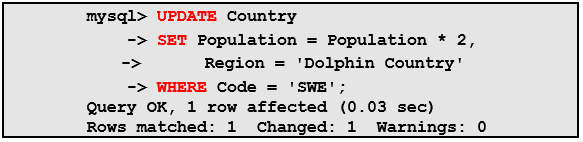
**Unit 6.3**

**UPDATE Statement**

* Modifies contents of existing rows
* Is used with the **SET** clause for column assignments
* General syntax:



* Example:



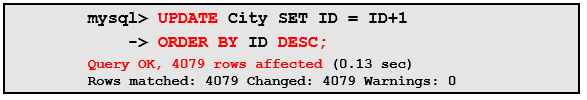
The example in the slide demonstrates how you can update the **Country** database by changing the **Population** column calculation and the **Region** to **Dolphin Country** for the country **Sweden** (code **'SWE'**). The results show one row **affected**, one row **matched** (selected by the **WHERE** clause), and one row **changed**.

**UPDATE Statement Ordering**

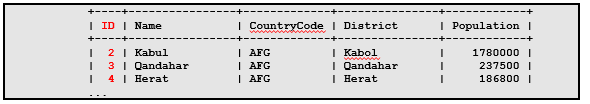
* There is no guarantee about the order in which rows are updated, which can result in errors:



* Use **ORDER BY** to control the order:



* + The content of the new row:



To solve the problem of the first **UPDATE** in the slide that causes an error, add an **ORDERBY** clause to cause the row updates to occur in a particular order, as shown in the second **UPDATE**.

The results shown in the slide are a result of the following query:

**mysql> SELECT \* FROM City ORDER BY ID;**

**+------+----------------+-------------+---------------+------------+**

**| ID | Name | CountryCode | District | Population |**

**+------+----------------+-------------+---------------+------------+**

**| 2 | Kabul | AFG | Kabol | 1780000 |**

**| 3 | Qandahar | AFG | Qandahar | 237500 |**

**| 4 | Herat | AFG | Herat | 186800 |**

**| 5 | Mazar-e-Sharif | AFG | Balkh | 127800 |**

**| 6 | Amsterdam | NLD | Noord-Holland | 731200 |**

**...**

**| 4079 | Nablus | PSE | Nablus | 100231 |**

**| 4080 | Rafah | PSE | Rafah | 92020 |**

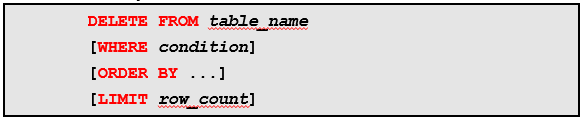
**+------+----------------+-------------+---------------+------------+**

**4079 rows in set (0.02 sec)**

**Note:** The **ID**s start at the number **2**, due to the update.

**DELETE Statement**

* Use DELETE to remove whole, specified table rows instead of individual columns
* General syntax:



* + - Use **WHERE** to indicate which rows to remove.
* Indicate table name only to remove *all* table rows:



* Use DELETE with extreme caution, because it does not have the “undo” feature.

The **DELETE** result indicates the number of rows affected, which can be zero (0) if the statement did not cause a change to be made.

**Exercise**

**Question 1**

Insert the following data into the following table

**Movie** (id, title, year, director)

|  |  |  |  |
| --- | --- | --- | --- |
| **Id** | **Title** | **Year** | **Director** |
| 1 | Ben hur | 2016 | Thomas |
| 2 | Get Smart | 2012 | Richard Bell |
| 3 | Spider Man | 2009 | Tam Morry |
| 4 | Batman V/S Super Man | 2015 | Gerald Hond |

**Feedback**

|  |
| --- |
|  |



**LAB # 07**

**The Purpose of this Lab is to introduce the DCL (Data Control Language) which includes Grant & Revoke Statements**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Introduction to DCL Experiment**

**07**

**Objective**

**The Purpose of this Lab is to introduce the DCL (Data Control Language) which includes Grant & Revoke Statements**

**Theory**

Unit 7.1

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

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

DCL defines two commands,

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

#### To Allow a User to create Session

**grant** create session to *username*;

#### To Allow a User to create Table

**grant** create table to *username*;

#### To provide User with some Space on Table space to store Table

**alter** user *username* quota unlimited on system;

#### To Grant all privilege to a User

**grant**sysdba to *username*

#### To Grant permission to Create any Table

**grant***create* any table to *username*

#### To Grant permission to Drop any Table

**grant***drop* any table to *username*

#### To take back Permissions

**revoke** create table from *username*

**Exercise**

**Question 1**

Grant all privilege to user which name Ali.

**Question 2**

Grant only creates permission to user which name Fahad.

**Feedback**

|  |
| --- |
|  |



**LAB # 08**

**The Purpose of this Lab is to get familiar with basic Select Queries which includes Selection & Projection Queries & Some Built-in Functions**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Select Queries (Selection, Projection, Functions) Experiment**

**08**

**Objective**

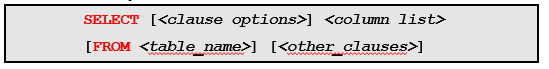
**The Purpose of this Lab is to get familiar with basic Select Queries which includes Selection & Projection Queries & Some Built-in Functions**

**Theory**

**unit 8.1**

**SELECT Statement**

* SELECT is the most commonly used **DML** command for queries:
  + Retrieves rows from tables in a database
  + Returns rows as a “result set” in the form of a table
* General syntax:

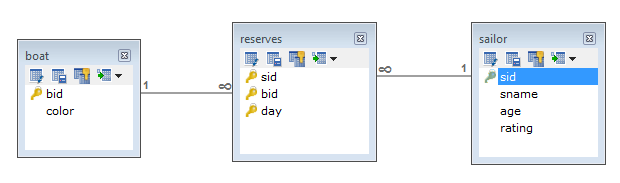


* ***column\_list*** is a list of column names that make up the result set.
  + Separate the items in the list with a comma separator (**,**).

**SELECT** is the most commonly used data manipulation language (DML) statement. The **SELECT** statement forms a description of the data that is to be retrieved from the database. It does not specify exactly how the data is to be retrieved. The result returned by the database server in response to a **SELECT** statement takes the form of a table, which is a collection of rows.

An expression that does not contain any column name is allowed in ***<column\_list>***.

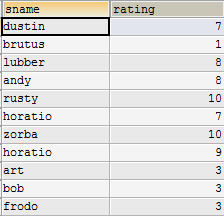
We have a database name **boat\_reservation:**

****

* **sailor** have id, name , age & rating.
* **boat** have id & color.
* Sailor can reserves multiple boats.
* Boat can be reserve by multiple sailors.
* So, There is a many-to-many relationship between **sailor**&**boat.**
* We Resolve this by Introducing new Relation name **reserves**.

**SELECT** example:

**mysql> SELECT sname, rating FROM sailor**



**Projection : Deletes unwanted columns from relation.**

Projection, which retrieves all the rows in the **Name** and **Rating** columns *from* the **Sailor** table. Queries can be performed on one, multiple, or all columns, on one or more tables.

For example, you can retrieve all the column data from the **Sailor**table, using (**\***):

**mysql> SELECT \* FROM sailor**

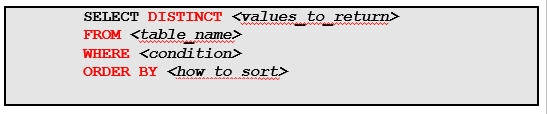
**Note:** The word “query” is most commonly used to describe a **SELECT** statement.

Unit 8.2

**Using SELECT Clauses**

Optional clauses can be used alone (or in combination) to generate specific query results.

* Types of clauses:
  + **DISTINCT:** Eliminates duplicate rows from the result set
  + **FROM:** Specifies what tables to retrieve data from
  + **WHERE:** Filters for specific rows
  + **ORDER BY:** Sorts rows by specified order
* Syntax example (with the order of the clauses fixed):



The syntax example in the slide shows the correct order and usage of each of the listed optional clauses. This statement performs the following:

* Specifically selecting **DISTINCT** rows
* **FROM** a named table
* **WHERE** some condition defines which rows to return
* Given a specific sorting order (**ORDER BY**)

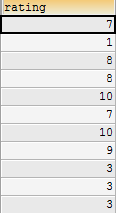
The order of the clauses is fixed. Therefore, if you do not keep them in their proper order, you get a syntax error. No result set is returned.

**Using SELECT with DISTINCT**

* Removes duplicate rows, so every result set row is unique
* Example of difference between **SELECT** *without* and *with* **DISTINCT**:

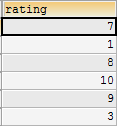
Without DISTINCT

**mysql>SELECT rating FROM sailor**

****

With DISTINCT

**mysql>SELECT DISTINCT rating FROM sailor**

****

**SELECT with WHERE**

Selection: Selects a subset of rows from relation.

Subset of Relation can be obtained using Some Condition (or Using Where Clause)

Expressions in a **WHERE** clause can use the following types of operators:

* **Arithmetic:+**, **-**, **\***, **/**, **DIV**, **%**
* **Comparison:<**, **<=**, **=**, **<=>**, **<>** or **!=**, **>=**, **>**, **BETWEEN**
* **Logical:AND**, **OR**, **XOR**, **NOT**
* **Additional options:IN**, **IS NULL**, **LIKE**, **()**

**Arithmetic Operators**

Perform mathematical operations on column value expressions:

* **+**  Addition
* **-**  Subtraction
* **\***  Multiplication
* **/**  Division
* **DIV** Integer division
* **%** Modulo (remainder after division)

**Comparison Operators**

Compare column value expressions

* **+**  Addition
* **<** Less than
* **<=** Less than or equal to
* **=**  Equal to
* **<=>** Equal to (works even for **NULL** values)
* **<>** or **!=** Not equal to
* **>=** Greater than or equal to
* **>** Greater than
* **BETWEEN *<value1>* AND *<value2>***Indicate a range of values (inclusive)

**Unit 8.3**

**Logical Operators**

Combine Boolean expressions

* **AND**: Logical AND
* **OR**: Logical OR
* **XOR**: Logical exclusive-OR
* **NOT**: Logical negation

Additional options in a **WHERE** condition:

* **IN** is equivalent to a list of comparisons with **OR**, but more readable and efficient.
* You can use **IS NULL** to check whether a value is null (such as **... *<column\_name>* IS NULL**).
* The **LIKE** operator can be used for pattern matching (such as **WHERE *<value>*LIKE *'<pattern>'***).
* You can also use parentheses **(*<expression>*)** to group parts of an expression.
* Operators can be applied to almost all types of value expressions such as literals, columns, function calls, and so on.
* You can also combine several criteria with logical operators.

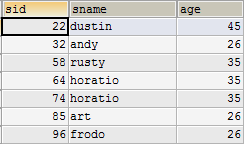
**SELECT with WHERE**

Example of **WHERE** with **IN**:

**mysql> SELECT sid, sname, age**

**FROM sailor**

**WHERE age IN (26,45,35)**



**SELECT with WHERE**

Example of **WHERE** with **AND** and**OR**:

**mysql> SELECT sid, sname, rating**

**FROM sailor**

**WHERE age>30 AND (rating = 3 OR rating = 8)**



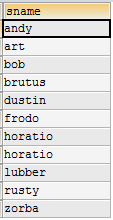
**SELECT with ORDER BY**

* Returns output rows in a specific order
* Example of **ORDER BY**:

**mysql> SELECT sname**

**FROM sailor**

**ORDER BY sname**

****

By default, the server returns the rows in the **SELECT** statement to the client in no particular order. When you issue a query, the server is free to return the rows in any convenient order. This order can be affected by factors such as the order in which the rows are actually stored in the table, or which indexes are used to process the query. If you require output rows to be returned in a specific order, include an **ORDER BY** clause that indicates how to sort the results.

The example in the slide sorts the names of the countries in alphabetical order by **sname**, from the **sailor**table.

**Previous Example With ORDER By Clause:**

**mysql> SELECT sid, sname, rating**

**FROM sailor**

**WHERE age>30 AND (rating = 3 OR rating = 8)**

**ORDER BY sname**

****

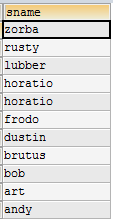
**SELECT with ORDER BY with ASC and DESC**

* Specify order with **ASC** and **DESC**
  + - **ASC:** Ascending order (default)
    - **DESC:** Descending order
* Example of **ORDER BY** with **DESC**:

**mysql> SELECT sname**

**FROM sailor**

**ORDER BY sname DESC**

****

Ascending order is the default sort order in an **ORDER BY** clause. You can specify explicitly whether to sort a column in ascending or descending order by using the **ASC** or **DESC** keywords after the column names. Ascending means that the lowest value comes first.

**Functions**

**MySQL Expressions**

* Functions can be invoked within expressions.
* An expression returns a value that is used in place of the function call when evaluated.
* General syntax:



* + The parentheses following the function name are required.
* Examples:
  + **SELECT NOW():** The **NOW** function returns the current date and time.
  + **SELECT VERSION():** The **VERSION** function returns the MySQL server version currently being used on the host.

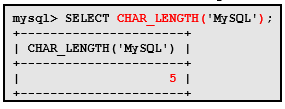
In the general syntax, a function call includes the function name followed by parentheses (which may contain arguments used to perform the function).

**Unit 8.4**

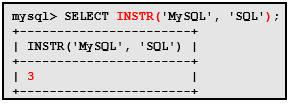
**String Functions**

**String Function: Numeric Category Examples**

* Returns the number of characters in the string:

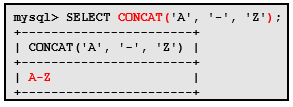


* Returns the position in the string where substring occurs:

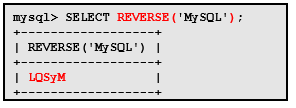


**String Function: String Category Examples (CONCAT, REVERSE, LEFT, RIGHT)**

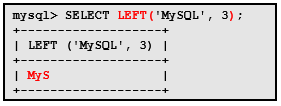
* Concatenates the given arguments into one string:



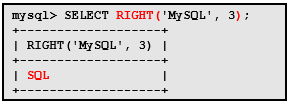
* Returns string with the characters in reverse order:



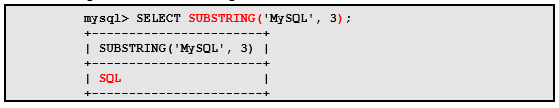
* Returns the left-most length characters of string:



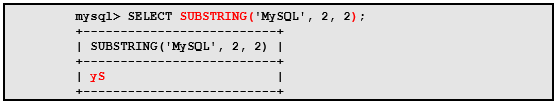
* Returns the right-most length characters of string:



* Returns the part of the string starting at the specified position through the end of string:

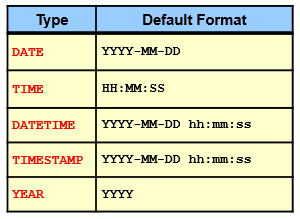


* Returns the part of the string starting at the specified position, and the number of characters indicated:



**Unit 8.5**

**Temporal Functions: Date/Time Formats**

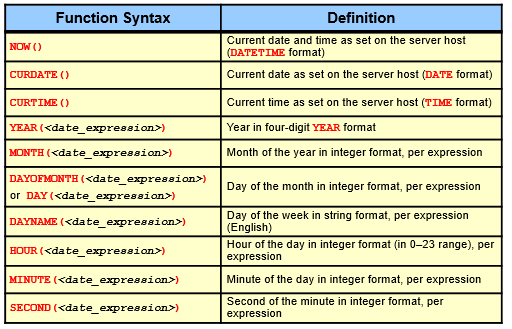


The Date/Time formats chart in the slide gives a list of all date and time formats permitted to be used within temporal functions.

See the **Data Types** lesson for specific information about the values and ranges for these format types.

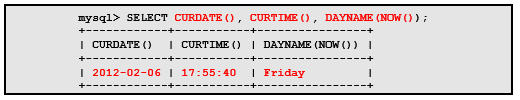
Functions that expect *date* values usually accept **DATETIME** values and ignore the *time* part. Functions that expect *time* values usually accept **DATETIME** values and ignore the *date* part.

**Temporal Functions: Function Types**

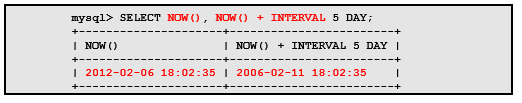


**Temporal Functions: Extracting Temporal Data Examples**

* Extracting current temporal data (date, time, and day of week):

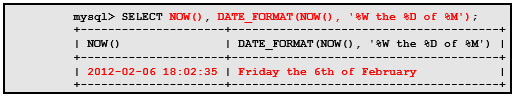


* Addition of a specified interval of days from the current date and time:



**Temporal Functions: Extracting Temporal Data Examples**

* Customizing the output format of temporal data:



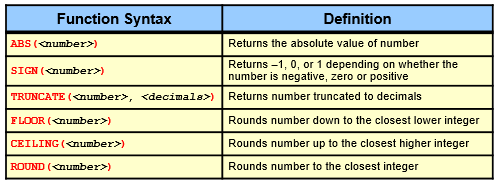
Syntax and additional information for the temporal functions:

* **DATE\_FORMAT (*<date>*, *<format>*)**
* The **%**symbol is required before any predefined date/time format specifiers.

**Numeric Functions**

Perform mathematical operations such as:

* Rounding
* Truncation
* Trigonometric calculations
* Generating random numbers

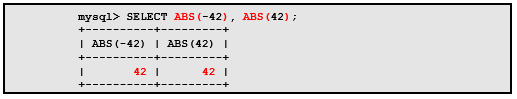


**ROUND (*<number>*):**

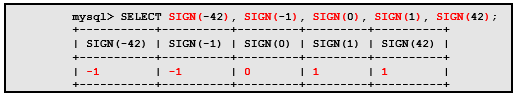
* Rounds up on **DECIMAL** data types
* Defaults to zero, if not specified
* For exact-value numbers, a value with a fractional part less than .5 is rounded down to the next integer if positive, or up to the next integer if negative.
* For approximate-value numbers, the result depends on the C library. On many systems, this means that a value with any fractional part is rounded to the nearest even integer.

**Numeric Functions: Examples**

* Returns the absolute value of the negative and positive values:



* Returns results of sign determination (–1=negative, 0=zero, 1=positive)**:**



**Numeric Functions: Additional Functions**

* Geometric functions:
  + **DEGREES()**, **PI()**, **RADIANS()**
* Trigonometric functions:
  + **COS()**, **SIN()**, **COT()**
  + **ACOS()**, **ASIN()**, **ATAN()**, **ATAN2()**
* Other functions:
  + **EXP()**, **LN()**, **LOG()**, **LOG2()**, **LOG10()**
  + **POWER()**, **SQRT()**
  + **MOD()**

**Exercise**

|  |  |  |  |
| --- | --- | --- | --- |
| Emp\_num | Emp\_Name | Emp\_Job | Emp\_Sal |
| E101 | Salman | Analyst | 6000 |
| E102 | Bushra | Programmer | 5000 |
| E103 | Madiha | Web Designer | 6000 |
| E104 | Batool | ERD designer | 4000 |
| E105 | Hameed | Web Designer | 3000 |
| E106 | Nini | Analyst | 2500 |
| E107 | Imtiaz | Web Designer | 6500 |
| E108 | Rashid | Programmer | 4000 |
| E109 | Muzzamil | ERD designer | 2000 |

1. List all employees’ number, employee’s name and jobs from emp.
2. List all employees’ number, employee’s name and jobs from emp whose salaries greater than 5,000.
3. List all employees’ number, employee’s name and jobs from emp whose salaries less than 5,000.
4. List all employees’ number, employee’s name and jobs from emp whose salaries between 1,000 to 5,000.
5. List all employees’ number, employee’s name, jobs and salaries from emp.

**Feedback**

|  |
| --- |
|  |



**LAB # 09**

**The Purpose of this Lab is to get familiar the Tables Joins Which Includes Inner, Outer & Natural Joins**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Select Queries (Join Types: Inner, Outer, Natural) Experiment**

**09**

**Objective**

**The Purpose of this Lab is to get familiar the Tables Joins Which Includes Inner, Outer & Natural Joins**

**Theory**

**Unit 9.1**

**Combining Multiple Tables**

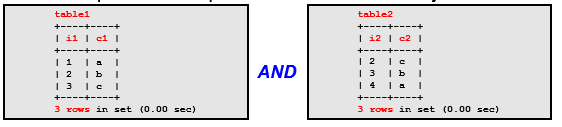
* Some questions cannot be answered using just one table.
* Some queries require **SELECT** statements that include a combination of multiple tables.
* A **join** operation is used to combine the data from one table with the data of another table.

The **SELECT** queries shown so far in this course retrieve data from a single table at a time. This lesson addresses the performance of multiple table queries.

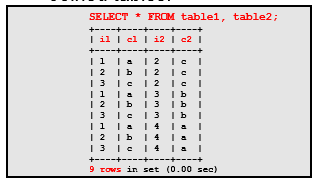
If you want to find the details of records referenced in a foreign key, you combine data from two or more tables by performing a table join.

**Joining Tables with SELECT**

* Example of two separate tables that can be joined:



* Joined tables:

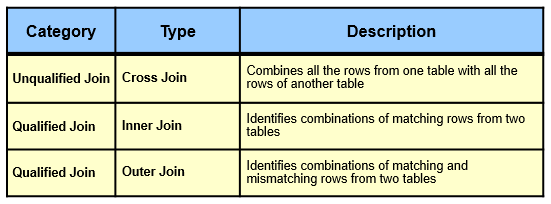


* + This join results in a **Cartesian product**.
  + The result includes all combinations of rows from **table1** and **table2**.

When you cross-join two tables (as in the example in the slide), each row from one table is combined with each row from every other table, yielding all possible combinations of rows. This result is also known as the **Cartesian product**. Joining tables this way has the potential to produce a very large number of rows because the row count is the product of the number of rows in each table.

A cross-join between two tables each containing 1000 rows will return 1000 × 1000 = 1 million rows. That is a lot of rows, even though the individual tables are small. Such a join is also called an **unqualified join**.

**Categories of Joins**



* **Unqualified join:** Retains all row pairs
* **Qualified join:** Retains only specific row pairs according to a particular *join condition*

The large majority of joins are qualified. Most qualified joins serve to combine related tables.

**Unit 9.2**

**INNER JOIN Keywords**

* **INNER JOIN** replaces the comma separator.
  + Can be used with the **ON** or **USING** clause
* Examples:

|  |  |
| --- | --- |
| Find the names of those sailor who reserved the boat. | |
| **mysql> SELECT s.sname**  **FROM sailor s INNER JOIN reserves r**  **ON s.sid = r.sid** |  |
| **mysql> SELECT s.sname**  **FROM sailor s INNER JOIN reserves r**  **USING(sid)** |

The syntax discussed earlier uses the comma separator in the **FROM** clause to name (and separate) the joined tables. Another inner join syntax uses the **INNER JOIN** keywords. With this syntax, those keywords replace the comma separator between table names. Also with **INNER JOIN**, the conditions that indicate how to perform row matching for the tables move from the **WHERE** clause to become part of the **FROM** clause.

The **ON** or **USING** clause conditions indicate how to perform the join.

**Note: ON** and **USING** are not exactly identical. **USING** treats the columns from the two tables as the same, whereas **ON** treats them as two different columns.

**JOIN Keywords**

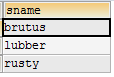
* **JOIN** is equivalent to **INNER JOIN**.
  + Can be used with the **ON** and **WHERE** clauses
* Example:Find the names of those sailor who reserved boat no. 103.

**mysql> SELECT s.sname**

**FROM sailor s JOIN reserves r**

**ON s.sid = r.sid**

**WHERE r.bid=103**

****

**Note:** The **INNER JOIN** keywords are equivalent to the **JOIN** keyword, and they can be used interchangeably.

**unit 9.3**

**Outer Joins**

* Identify combinations of matching and mismatching rows from two tables
* Can use two different types of **SELECT** query syntax:
  + **LEFT JOIN**: Uses the table to the left of the keywords to base the join comparison on
  + **RIGHT JOIN**: Uses the table to the right of the keywords to base the join comparison on

You do not get mismatches as you would with a cross-join.

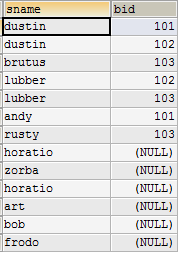
The **LEFT JOIN** and **RIGHT JOIN** answer the same kinds of questions, but differ slightly in their syntax. A **LEFT JOIN** can always be rewritten into an equivalent **RIGHT JOIN**.

**LEFT JOIN Keywords**

**mysql> SELECT s.sname, r.bid**

**FROM sailor s LEFT JOIN reserves r**

**ON s.sid = r.sid**

****

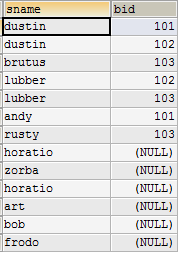
In the above example, the **LEFT JOIN** results in a list that shows where there were **NULL**s for those sailor who were not reserved any boat:

**RIGHT JOIN Keywords**

**mysql> SELECT s.sname, r.bid**

**FROM reserves r RIGHT JOIN sailor s**

**ON r.sid = s.sid**

****

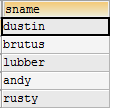
In the above example, the **RIGHT JOIN** results in a list that shows where there were **NULL**s for those sailor who were not reserved any boat:

**Natural Join**

* The associated tables have one or more pairs of identically named columns.
* The columns must be the same data type.
* Don’t use ON clause in a NATURAL JOIN.
* Example

**mysql> SELECT s.sname**

**FROM sailor s NATURAL JOIN reserves r**

****

**Exercise**

**Question No. 1**

**We have the following tables which creates statement are**

1. create table supplier(S\_NO int identity(1,1), SUP\_ID varchar(50) primary key,SUP\_NAME varchar (50), SUP\_ADD varchar (50), SUP\_NIC int, rank int)
2. create table Product(Prod\_ID varchar(50) primary key,Prod\_Name varchar(50),[Price/Unit] float)
3. create table Orders(Ord\_ID varchar(50) primary key,Prod\_ID varchar(50),Qty int,Totalrate int,SUP\_ID varchar (50))

**Write the Queries using Joins on above tables**

1. List of all supplier that palced order
2. List of all Product that are supplied by supplier whose id is 101
3. Find all order(s) of product named Rice

**Feedback**

|  |
| --- |
|  |



**LAB # 10**

**The Purpose of this Lab is to get familiar that how to use Groups & Having Clauses with Aggregates Functions**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Select Queries (Aggregates, Groups, Having) Experiment**

**10**

**Objective**

**The Purpose of this Lab is to get familiar that how to use Groups & Having Clauses with Aggregates Functions**

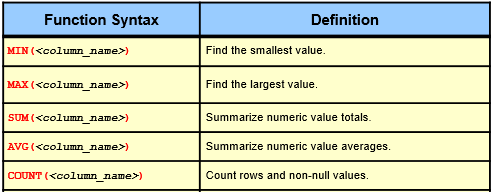
**Theory**

**Aggregate Functions**

* Perform summary operations on a set of values, such as:
  + Counting
  + Averaging
  + Finding minimum or maximum values
* Calculate a single value based on a group of values from different rows.
* Result value is based only on non-**NULL** values from the selected rows.

Aggregate functions are also known as *group* functions. A **SELECT** statement can produce a list of rows that match a given set of conditions. This list provides the details about the selected rows. However, if you want to know about the overall characteristics of the rows, you are more interested in getting a summary instead. The resulting value is based only on non-**NULL** values from the selected rows (with the exception that **COUNT(\*)** counts all rows).

Some of the aggregate function types:



All the functions can use the **DISTINCT** keyword, although it is not useful for the **MAX()** and **MIN()** functions.

**DISTINCT** examples:

**SUM(DISTINCT <column\_name>)**

**AVG(DISTINCT <column\_name>)**

**COUNT(DISTINCT <column\_name>)**

**Aggregate Functions: COUNT Function Examples**

**mysql>SELECT COUNT(\*) FROM reserves**

****

**mysql>SELECT COUNT(sid) FROM reserves**

****

**mysql> SELECT COUNT(DISTINCT sid) FROM reserves**

****

**Aggregate Functions: SUM Function Example**

**mysql> SELECT SUM(sid) FROM reserves**

****

**Aggregate Functions: AVG Function Example**

**mysql>SELECT AVG(sid) FROM reserves**

****

**mysql> SELECT AVG(rating) FROM sailor**

****

**Aggregate Functions: MIN Function Example**

**mysql> SELECT MIN(rating) FROM sailor**

****

**Aggregate Functions: MAX Function Example**

**mysql> SELECT MAX(rating) FROM sailor**

****

**GROUP BY**

* The **GROUP BY** clause places rows into groups.
  + Each group consists of rows having the same value in one or more columns.
  + Calculates a summary value for each group
* General Syntax:

SELECT [DISTINCT] *target-list*

FROM *relation-list*

WHERE *qualification*

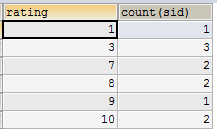
GROUP BY *grouping-list*

* Example:

**mysql> SELECT rating, COUNT(sid)**

**FROM sailor**

**GROUP BY rating**

****

In above Example, group is make on rating column and count the no. of sailor for each rating.

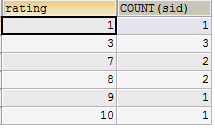
Find no. of sailor for each rating in which sailor age must be greater than 25

**mysql> SELECT rating, COUNT(sid)**

**FROM sailor**

**WHERE age>25**

**GROUP BY rating**

****

**HAVING**

* Use the **HAVING** clause to eliminate rows based on aggregate values.
  + Evaluated after the grouping implied by **GROUP BY**
* General Syntax:

SELECT [DISTINCT] *target-list*

FROM *relation-list*

WHERE *qualification*

GROUP BY *grouping-list*

HAVING *group-qualification*

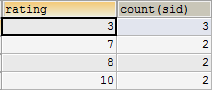
* Example

**mysql> SELECT rating, COUNT(sid)**

**FROM sailor**

**GROUP BY rating**

**HAVING COUNT(sid)>1**

****

In above Example, group is make on rating column and count the no. of sailor for each rating which rating have minimum sailor count is 2.

You can use the **HAVING** modifier to require that the groups produced by a **GROUP BY** clause satisfy particular criteria.

As such, it resembles the **WHERE** clause. The difference is that the **HAVING** clause is evaluated after the grouping implied by the **GROUP BY** clause. This means that the **HAVING** condition can refer to aggregate functions. However, any part of a condition that can also appear in the **WHERE** clause should not be in the **HAVING** clause. A good **HAVING** clause is always based on aggregate functions (because these are not allowed in the **WHERE** clause).

**Exercise**

**Question No. 1**

1. List average salary of each job.
2. Find average and sum of all the salaries of each job excluding clerks.
3. Find average and sum of the salaries of each job excluding salesmen', clerk' and 'manager'.
4. Find count, sum and average salaries of each job excluding salesmen', clerk' and 'manager'.
5. List average salary of each department.

**Feedback**

|  |
| --- |
|  |



**LAB # 11**

**The Purpose of this Lab is to get familiar with the Analytical Queries Which includes Rollup, Cube & Top- N (Limits) Queries**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Analytical Queries (Rollup, Cube) Experiment**

**11**

**Objective**

**The Purpose of this Lab is to get familiar with the Analytical Queries Which includes Rollup, Cube & Top- N (Limits) Queries**

**Theory**

The ROLLUP operator is useful in generating reports that contain subtotals and totals. The ROLLUP operator generates a result set that is similar to the result sets generated by the CUBE operator.

**ROLLUP**

* ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.
* General Syntax:

SELECT [DISTINCT] *target-list*

FROM *relation-list*

WHERE *qualification*

GROUP BY *grouping-list*

WITH ROLLUP

* Example:

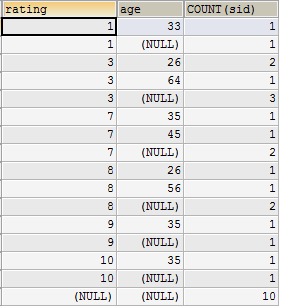
**mysql> SELECT rating, age, COUNT(sid)**

**FROM sailor**

**WHERE age>25**

**GROUP BY rating, age**

**WITH ROLLUP**

****

**CUBE**

* CUBE generates a result set that shows aggregates for all combinations of values in the selected columns.

**LIMIT**

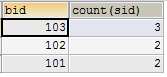
* Specifies the number of rows to output in result set
* Example without**LIMIT**:

**mysql> SELECT bid,COUNT(sid)**

**FROM reserves**

**GROUP BY bid**

**ORDER BY bid DESC**

****

* Example with **LIMIT**

**mysql> SELECT bid, COUNT(sid)**

**FROM reserves**

**GROUP BY bid**

**ORDER BY bid DESC**

**LIMIT 1**

****

**Exercise**

**Question No. 1**

1. Find the names of Top 10 employees which salaries are highest.

**Feedback**

|  |
| --- |
|  |



**LAB # 12**

**The Purpose of this Lab is to get familiar with the concept of Normalization**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**MYSQL Stored procedures Experiment**

**12**

**Objective**

**The Purpose of this Lab is to get familiar with the stored procedures using MYSQL**

**Theory**

## Stored Procedure

A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. A procedure has a name, a parameter list, and SQL statement(s). All most all relational database system supports stored procedure, MySQL 5 introduce stored procedure.

stored procedures which you call, or functions whose return values you use in other SQL statements the same way that you use pre-installed MySQL functions like pi(). The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement.

**Why Stored Procedures?**

### Reduce network traffic

Stored procedures help reduce the network traffic between applications and MySQL Server. Because instead of sending multiple lengthy SQL statements, applications have to send only the name and parameters of stored procedures.

### Centralize business logic in the database

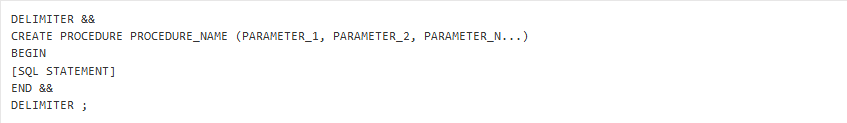
You can use the stored procedures to implement business logic that is reusable by multiple applications. The stored procedures help reduce the efforts of duplicating the same logic in many applications and make your database more consistent.

### Make database more secure

The database administrator can grant appropriate privileges to applications that only access specific stored procedures without giving any privileges on the underlying tables.

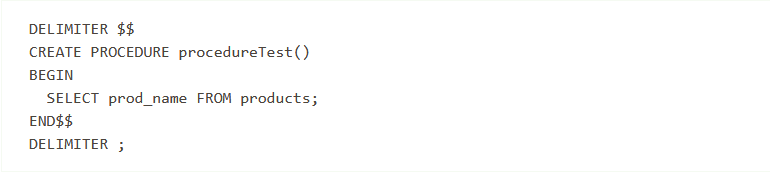
### Stored Procedure Syntax

The basic syntax of creating a stored procedure in MySQL database is shown below:

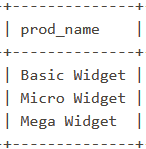


* The DELIMITER && line at the beginning tells MySQL server to treat the following SQL statements as a single statement, rather than executing them individually. Another && is included on a later line to mark the end of this statement.
* PROCEDURE\_NAME is where the name of your stored procedure is declared.
* The procedure name is followed by a set of parentheses, and these enclose the parameters to your procedure. Stored procedures support comma-separated parameters, and this feature makes them more flexible. The BEGIN...END commands enclose the SQL statement that you want to be executed by the stored procedure.
* In the end, the statement DELIMITER ; is issued again to change the delimiter back to the default value of ;

The following MySQL statements demonstrate how to create a very basic stored procedure named **procedureTest**. This procedure performs a simple lookup on the **products** table that we used in the stored function example



To invoke the stored procedure, use the following MySQL statement:



### Stored Procedure Parameters

Each parameter for a procedure has a type, name, and a data type, separated by spaces:

PARAMETER\_TYPE PARAMETER\_NAME DATA\_TYPE

For example, to create a parameter of type IN, named category, with the VARCHAR data type that has a length of **50** characters, use this syntax:

IN category VARCHAR (50)

MySQL supports three types of parameters:

* IN: The value of the parameter must be specified by the calling client. This value can not be changed by the stored procedure.

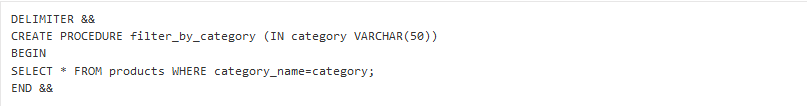
For example, if you pass a MySQL session variable as an IN parameter to a procedure, and the procedure modifies this value in its statements, your session variable will remain unmodified after the procedure exits.

* OUT: This type of parameter is also specified by the calling program, but its value can be changed by the stored procedure and retrieved by the calling program.

Note that the stored procedure cannot access the initial value of a variable that is passed as an OUT parameter.

* INOUT: A parameter of this type combines the behaviors of IN and OUT parameters:
  + The stored procedure can read the initial value of the parameter.
  + The parameter can be changed during stored procedure execution.
  + The changed value can be returned back to the calling program, if the calling program passed a variable as the parameter.

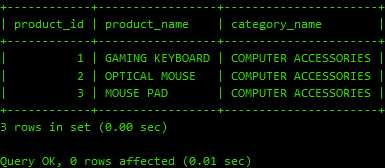
Parameter example



To execute the filter\_by\_category stored procedure that we created above, enter the command below:



The stored procedure should now output all products in the COMPUTER ACCESSORIES category because we have specified COMPUTER ACCESSORIES as a parameter:



## Deleting Stored Procedures

You can delete a MySQL stored procedure if you no longer want to use it or if you want to recreate it from scratch. The basic syntax of dropping the stored procedure is shown below:

DROP PROCEDURE IF EXISTS PROCEDURE\_NAME;

DROP PROCEDURE IF EXISTS filter\_by\_category;

**Exercise**

1-Creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table.

2-creates a stored procedure that selects Customers from a particular City from the "Customers" table

3- Creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

**Feedback**

|  |
| --- |
|  |



**LAB # 13**

**The Purpose of this Lab is to make Normalized ERD – by using Case Studies**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Normalization – Case Study Experiment**

**13**

**Objective**

**The Purpose of this Lab is to make Normalized ERD – by using Case Studies**

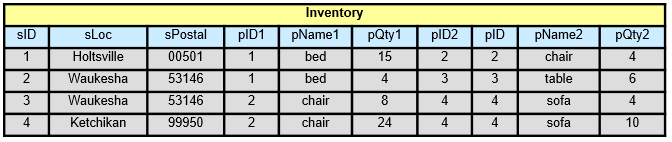
**Theory**

**CASE - STUDY**

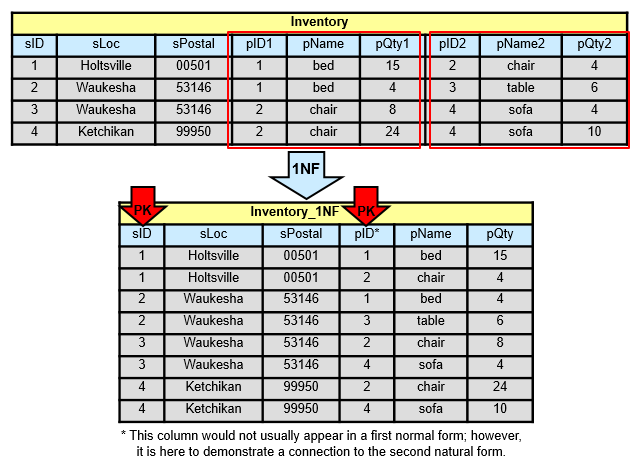
The normalization process is demonstrated by using a very simple example:

* Furniture store
  + You want to turn your inventory spreadsheet into a database, so you can:
    - Execute intelligent searches
    - Make modifications
    - Prepare for future growth of your business

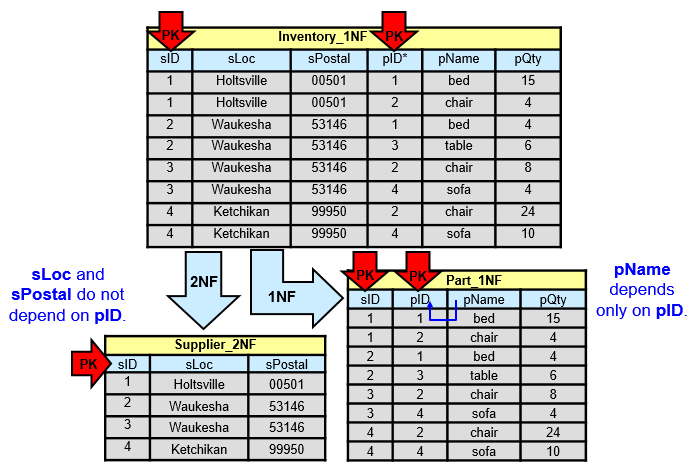
Original inventory spreadsheet



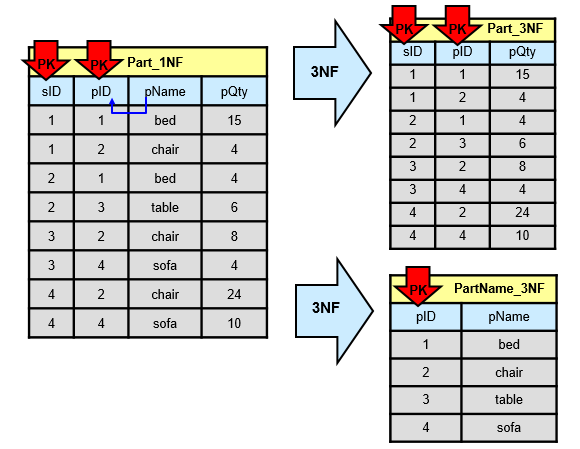
**First Normal Form: 1NF**



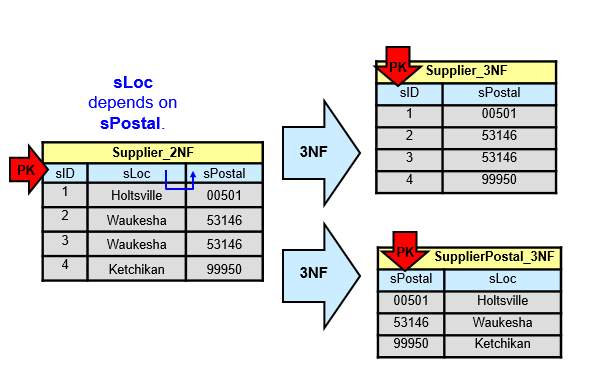
**Second Normal Form: 2NF**



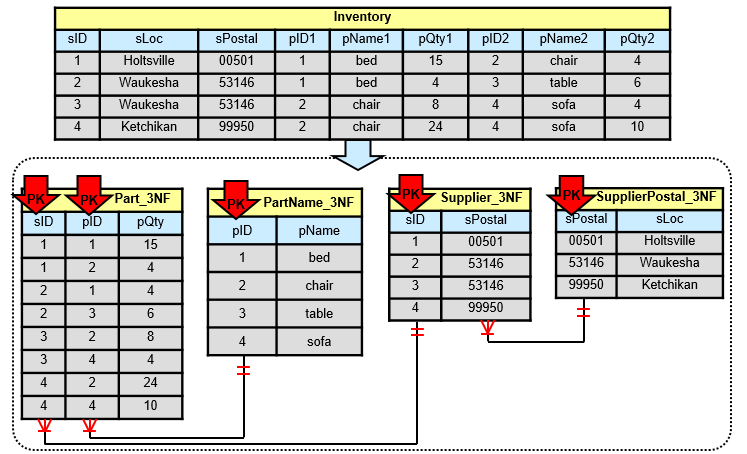
**Third Normal Form: 3NF**



**Third Normal Form: 3NF**



**Normalized “Furniture Store” Database**

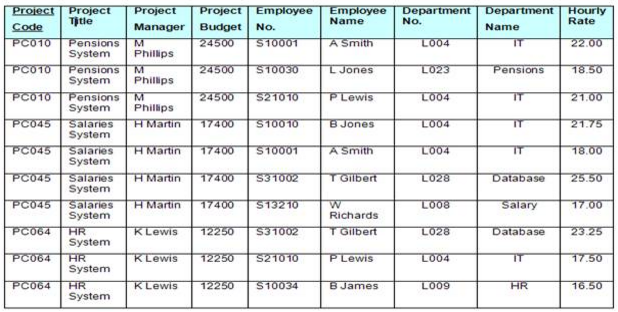


**Exercise**

**CASE - STUDY**

We have an Un-Normalized data of Project Management System. Transform this into the Normalized Forms.

1. **1NF** - No Repeating Groups Within the Row
2. **2NF -** Every non-key column value is dependent on the Primary Key
3. **3NF -** Dependent Solely on the primary key and no other non-key (supporting) column value



**Feedback**

|  |
| --- |
|  |



**LAB # 15**

**The Purpose of this Lab is to get familiar with the Database Administration that how we Install, Configure, Import & Export Databases**

|  |  |
| --- | --- |
| **Name** |  |
| **Date** |  |
| **Registration No** |  |
| **Department** |  |
| **Quiz** |  |
| **Assignment** |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab Instructor Signature**

**Database Administration Experiment**

**15**

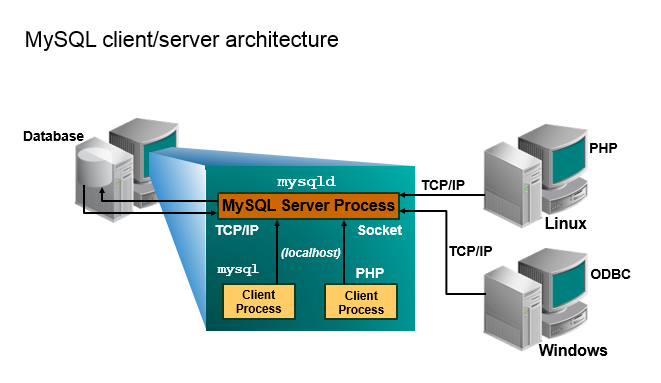
**Objective**

**The Purpose of this Lab is to get familiar with the Database Administration that how we Install, Configure, Import & Export Databases**

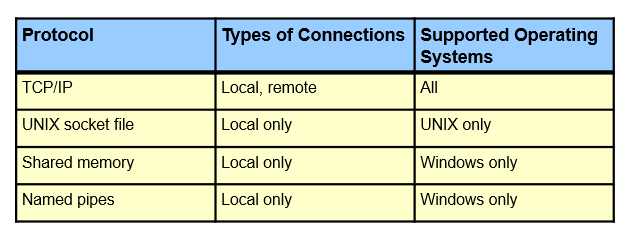
**Theory**

**Installation**

**MySQL Client/Server Model**



**Communication Protocols**



* Protocols are implemented in the client libraries and drivers.
* The speed of a connection protocol varies with the local settings.

**LAMP Stack**

* MySQL is the database component in the **LAMP** stack, which has become the preferred stack for web applications:
  + **Linux:** Operating system
  + **Apache:** Web server
  + **MySQL:** Relational Database Management System
  + **PHP**/**Perl**/**Python:** Programming languages
* The LAMP stack offers many degrees of freedom.
* You can go from **LAMP** to **WAMP** (Windows).

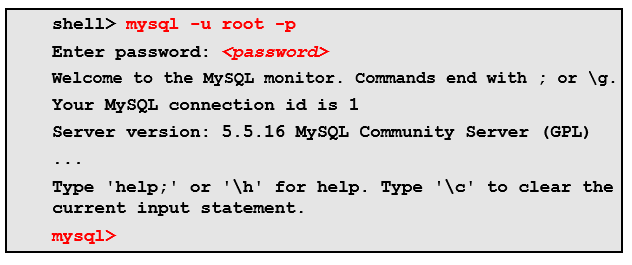
**Installing MySQL**

* You can download MySQL products from:
  + Community: <http://dev.mysql.com/downloads>
  + Enterprise: <https://edelivery.oracle.com/>
* The Windows operating system is used for this course.
* The MySQL installer for Windows is recommended:
  + <http://dev.mysql.com/downloads/installer/>
  + Installs several MySQL products
* Example databases for testing MySQL features:
  + The **world\_innodb** database is used for examples and practices in this course.
  + This and other sample databases are available (free) online.

**Configuration**

**Starting the MySQL Command-Line Client**

Start the **mysql** client with the following command:



**Startup Command-Line Options**

Get a full list of options by using **–-help**:



* Some primary options:
  + **-h:** Host name or IP address
  + **-u:** Username
  + **-p:** Password
  + ***<database>*:** Database name
  + **<:** Input script
  + **>:**  Output file name
  + **-b:** Beeps
* Syntax example:



**Keyboard Editing**

**mysql** client command-line editing methods:

* Keyboard directional arrows
* Command history preserved during session
* Interrupt a query
* Full readline capabilities under Linux
* Close quotes
* Copy and paste into window

**tee File**

* + The tee file is a text file that logs all **mysql** client statements and their output.
  + Everything you see displayed on the screen is appended into a given file.
  + You can enable this feature interactively within the **mysql** client with the **tee** command:

**mysql> tee session\_tee\_log.txt**

* + The **tee** file can be disabled with the **notee** command.
  + Executing tee again re-enables logging.
  + View the text file from the file location by using the command-line text editor or Windows Explorer.

**Exporting Data**

* Backups are important!
* Backup principles:
  + Perform regularly.
  + Provide consistent and meaningful naming scheme.
  + Expire backup files.
  + Include with regular system backups.
* Reasons for exporting data:
  + Copying databases from one server to another:
    - From within the same host
    - To another host
  + Testing a new MySQL release with real data
  + Transferring data from one RDBMS to another

**Exporting with a Query**

* Write query results directly into a file by using **SELECT** with the **INTO OUTFILE** clause.
* Example:



* + The name of the file indicates the location of the write.
* Default file format:
  + Values are delimited by tab characters.
  + Lines are terminated with newlines.
* The file format can be changed by using specific **INTO OUTFILE** options.

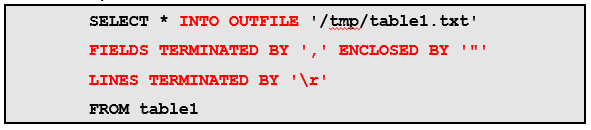
**Exporting with a Query: INTO OUTFILE**

**INTO OUTFILE** changes the standard **SELECT** operation:

* A file is written to the server host rather than to the executing client.
* Data is written to a new file only; there are no overwrites.
* The file contains one line per row selected by the statement.
* The user that executes the statement must have the **FILE** privilege.
* The file is created with file system access permissions.

**Exporting with a Query: CSV Format**

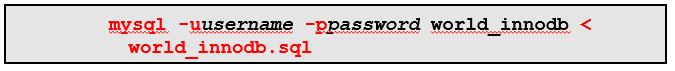
* Create a text file that contains information in   
  comma-separated values (**CSV**) format:
  + Values are enclosed in double quotation marks.
  + Lines are terminated by carriage returns.
* Example:



**Importing Data**

New data can be imported into a database by using a MySQL (**.sql**) statement file, which contains all the information needed to create tables.

* Example: Import the **world\_innodb** database from file by using the **mysql** command.



* + Use the input operator (<) to indicate the SQL file name.
* Example: Import the data file within the **mysql** client.



**Caution:** Do *not* import over an existing database file with the same name.

Shell-level client commands do not require a semicolon (;) at the end of the statement.

A text file (.txt) can also be used to import data.

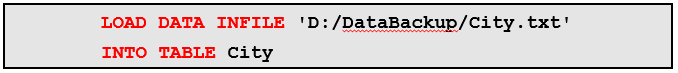
**Importing from a Data File**

You should know the following characteristics of the data file:

* The column value separator
* The order of the columns
* The row separator
* The file system where the file resides
* What are the values enclosed within? (example: double quotation marks)
* Are the column names specified in the file?
* Is there a header indicating rows of the table to skip before importing?
* Are privileges required to access the file?

**Importing with the LOAD DATA INFILE Statement**

* **LOAD DATA INFILE** is the reverse operation of **SELECT** with **INTO OUTFILE**.
  + However, it uses similar clauses and format specifiers.
* It reads row values from a file into a table.
* Files can be in tab-delimited or comma-separated format.
* Example:



* MySQL assumes that the file is located on the server host in the database data directory.

The simplest form of the **LOAD DATA INFILE** statement specifies only the name of the data file and the table into which to load the file, as shown in the example in the slide.

The syntax for **LOAD DATA INFILE** is as follows, where optional parts of the statement are indicated by square brackets:

**LOAD DATA [LOCAL] INFILE 'file\_name'**

**[IGNORE | REPLACE]**

**INTO TABLE table\_name**

**format\_specifiers**

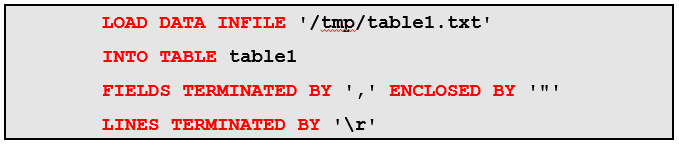
**[IGNORE n LINES]**

**[(column\_list)]**

**[SET (assignment\_list)]**

**Importing with LOAD DATA INFILE: CSV Format**

* Import a text file that contains information in   
  comma-separated values format.
* Example:



The file name is given as a quoted string. On Windows, the pathname separator character is “**\**”, but MySQL treats the backslash as the escape character in strings. To deal with this issue, write separators in Windows pathnames either as “**/**” or as “**\\**”. To load a file named **D:\mydata\data.txt**, specify the file name as shown in either of the following statements:

**LOAD DATA INFILE 'D:\\mydata\\data.txt' INTO TABLE t**

The example in the slide shows how to use a **LOAD DATA INFILE** statement to import a file named **/tmp/data.txt** that contains information in comma-separated values (CSV) format, with values enclosed within double quotation marks and lines terminated by carriage returns into a table named **table1**.

**Exercise**

**Question No. 1**

Using SELECT statement find the version of the server you are running and print the name of the current database?

**Question No. 2**

Write the statement to export tables from MySQL DBMS?

**Question No. 3**

Write the statement to log all SQL statements and their results in a text file.

**Feedback**

|  |
| --- |
|  |