DB AND SQL

Databases can be found in almost all software applications. SQL is the standard language to query a database. This course will teach you database design. Also, it teaches you basic to advanced SQL

What is Data?

In simple words data can be facts related to any object in consideration.

For example your name, age, height, weight, etc are some data related to you.

A picture , image , file , pdf etc can also be considered data.

What is a Database?

Database is a systematic collection of data. Databases support storage and manipulation of data. Databases make data management easy. Let's discuss few examples.

An online telephone directory would definitely use database to store data pertaining to people, phone numbers, other contact details, etc.

Your electricity service provider is obviously using a database to manage billing , client related issues, to handle fault data, etc.

Let's also consider the facebook. It needs to store, manipulate and present data related to members, their friends, member activities, messages, advertisements and lot more.

What is a Database Management System (DBMS)?

Database Management System (DBMS) is a collection of programs which enables its users to access database, manipulate data, reporting / representation of data .

It also helps to control access to the database.

Relational DBMS - this type of DBMS defines database relationships in form of tables, also known as relations. Unlike network DBMS, RDBMS does not support many to many relationships. Relational DBMS usually have pre-defined data types that they can support. This is the most popular DBMS type in the market. Examples of relational database management systems include MySQL, Oracle, and Microsoft SQL Server database.

What is SQL?

Structured Query language (SQL) **pronounced as "S-Q-L" or sometimes as "See-Quel**" is actually the standard language for dealing with Relational Databases.

SQL programming can be effectively used to insert, search, update, delete database records.

That doesn't mean SQL cannot do things beyond that.

In fact it can do lot of things including, but not limited to, optimizing and maintenance of databases.

Relational databases like MySQL Database, Oracle, Ms SQL server, Sybase, etc uses SQL ! **How to use sql syntaxes?**

SQL syntaxes used in these databases are almost similar, except the fact that some are using few different syntaxes and even proprietary SQL syntaxes.

SQL Example

```
SELECT * FROM Members WHERE Age > 30
```

What is NoSQL ?

NoSQL is an upcoming category of Database Management Systems. Its main characteristic is its non-adherence to Relational Database Concepts. NOSQL means "Not only SQL".

Concept of NoSQL databases grew with internet giants such as Google, Facebook, Amazon etc who deal with gigantic volumes of data.

When you use relational database for massive volumes of data , the system starts getting slow in terms of response time.

To overcome this , we could of course "scale up" our systems by upgrading our existing hardware.

The alternative to the above problem would be to distribute our database load on multiple hosts as the load increases.

This is known as "scaling out".

NOSQL database are **non-relational databases** that scale out better than relational databases and are designed with web applications in mind.

They do not use SQL to query the data and do not follow strict schemas like relational models.With NoSQL, ACID (Atomicity, Consistency, Isolation, Durability) features are not guaranteed always

Characteristics

The characteristics of these four properties as defined by Reuter and Härder are as follows:

Atomicity

Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code), but merely that any programming errors cannot result in the violation of any defined rules.

Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on the concurrency control method (i.e., if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.

Durability

The durability property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

Why it makes sense to learn SQL after NOSQL ?

With the advantages of NOSQL databases outlined above that scale out better than relational models, you might be thinking **why one would still want to learn about SQL database?**

Well, **NOSQL databases** are sort of highly specialized systems and have their special usage and limitations. NOSQL suit more for those who handles huge volumes of data. The vast majority, use relational databases and associated tools.

Relational databases have the following advantages over NOSQL databases;

- SQL(relational) databases have a mature data storage and management model . This is crucial for enterprise users.
- SQL databases support the notion of views which allow users to only see data that they are authorized to view. The data that they are not authorized to see is kept hidden from them.
- SQL databases support stored procedure sql which allow database developers to implement part of the business logic into the database.
- SQL databases have better security models compared to NoSQL databases.

The world has not deviated from use of relational databases. There is **growing** a demand for professionals who can handle relational databases. Thus learning databases and SQL still holds merit.

Summary

- DBMS stands for Database Management System.
- We have four major types of DBMSs namely Hierarchical, Network, Relational, Object Oriented
- The most widely used DBMS is the relational model that saves data in table formats. It uses SQL as the standard query language
- SQL language is used to Sql query a database
- The database approach has many advantages when it comes to

storing data compared to the traditional flat file based systems

What is MySQL?

MySQL is an open source relational database.

MySQL is cross platform which means it runs on a number of different platforms such as Windows, Linux, and Mac OS etc.

Why use MySQL?

There are a number of relational database management systems on the market.

Examples of relational databases include Microsoft SQL Server, Microsoft Access, Oracle, DB2 etc.

One may ask why we would choose MySQL over the other database management systems.

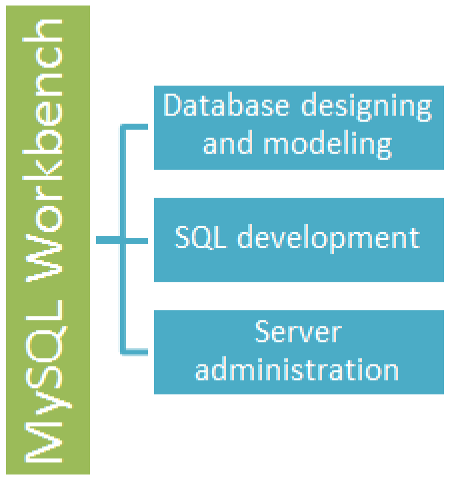
The answer to this question depends on a number of factors.

Let's look at the strengths of MySQL compared to over relational databases such as SQL Server-

- MySQL supports multiple storage engines each with its own specifications while other systems like SQL server only support a single storage engine. In order to appreciate this statement, let's look at two of the storage engines supported by MySQL.
    - InnoDB: - its default storage engine provided with MySQL as of version 5.5. InnoDB supports foreign keys for referential integrity and also supports ACID-standard transactions.
    - MyISAM: - it was the default storage engine for MySQL prior to version 5.5. MyISAM lacks support for transactions. Its advantages over InnoDB include simplicity and high performance.
- MySQL has high performance compared to other relation database systems. This is due to its simplicity in design and support for multiple-storage engines.
- Cost effective, it's relatively cheaper in terms of cost when compared to other relational databases. In fact, the community edition is free. The commercial edition has a licensing fee which is also cost effective compared to licensing fees for products such as Microsoft SQL Server.
- Cross platform - MySQL works on many platforms which means it can be deployed on most machines. Other systems such as MS SQL Server only run on the windows platform.

In order to interact with MySQL, you will need a **server access tool** that can communicate with MySQL server. MySQL supports multiple user connections.

Introducing MySQL Workbench



MySQLWorkbench is a **Visual database designing and modeling** access tool for MySQL server relational database. It facilitates creation of new physical data models and modification of existing MySQL databases with reverse/forward engineering and change management functions.

What is Database Design?

Database Design is a collection of processes that facilitate the designing, development, implementation and maintenance of enterprise data management systems

It helps produce database systems

1. That meet the requirements of the users
2. Have high performance.

The main objectives of database designing are to produce logical and physical designs models of the proposed database system.

The logical model concentrates on the data requirements and the data to be stored independent of physical considerations. It does not concern itself with how the data will be stored or where it will be stored physically.

The physical data design model involves translating the logical design of the database onto physical media using hardware resources and software systems such as database management systems (DBMS).

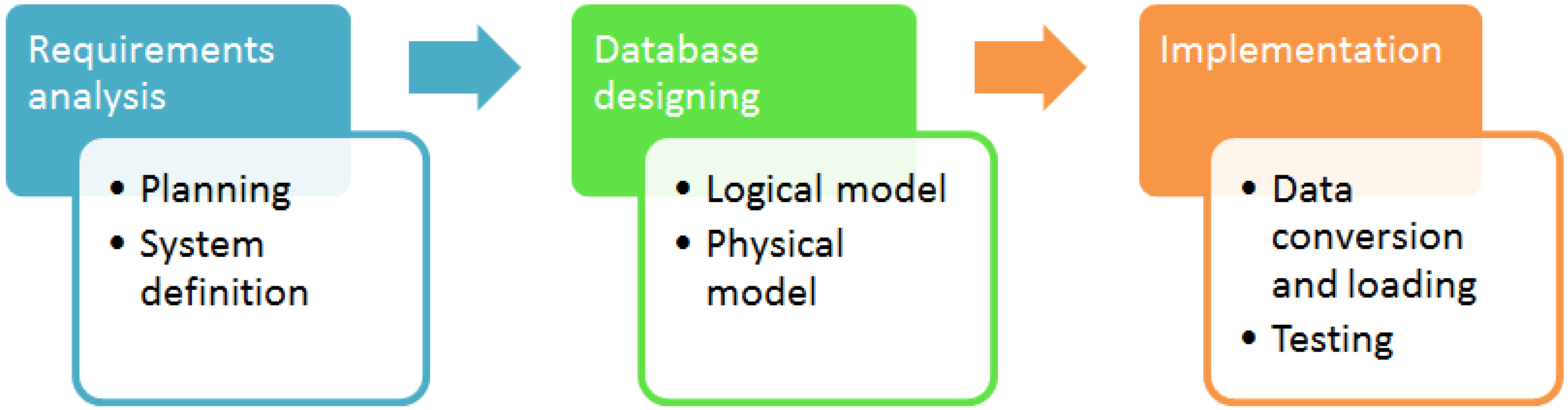
Why Database Design is Important ?

Database designing is crucial to **high performance** database system.

Apart from improving the performance, properly designed database are easy to maintain, improve data consistency and are cost effective in terms of disk storage space.

Note , the genius of a database is in its design . Data operations using SQL is relatively simple

Database development life cycle



The database development life cycle has a number of stages that are followed when developing database systems.

The steps in the development life cycle do not necessary have to be followed religiously in a sequential manner.

On small database systems, the database system development life cycle is usually very simple and does not involve a lot of steps.

In order to fully appreciate the above diagram, let's look at the individual components listed in each step.

Requirements analysis

- **Planning** - This stages concerns with planning of entire Database Development Life Cylce It takes into consideration the Information Systems strategy of the organization.
- **System definition** - This stage defines the scope and boundaries of the proposed database system.

Database designing

- **Logical model** - This stage is concerned with developing a database model based on requirements. The entire design is on paper without any physical implementations or specific DBMS considerations.
- **Physical model** - This stage implements the logical model of the database taking into account the DBMS and physical implementation factors.

Implementation

- **Data conversion and loading** - this stage is concerned with importing and converting data from the old system into the new database.

- **Testing** - this stage is concerned with the identification of errors in the newly implemented system .It checks the database against requirement specifications.

Two Types of Database Techniques

1. **Normalization**
2. **ER Modeling**

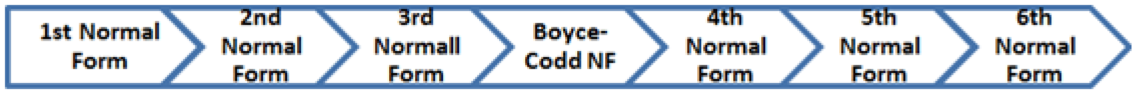
What is Normalization?

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

It divides larger tables to smaller tables and link them using relationships.

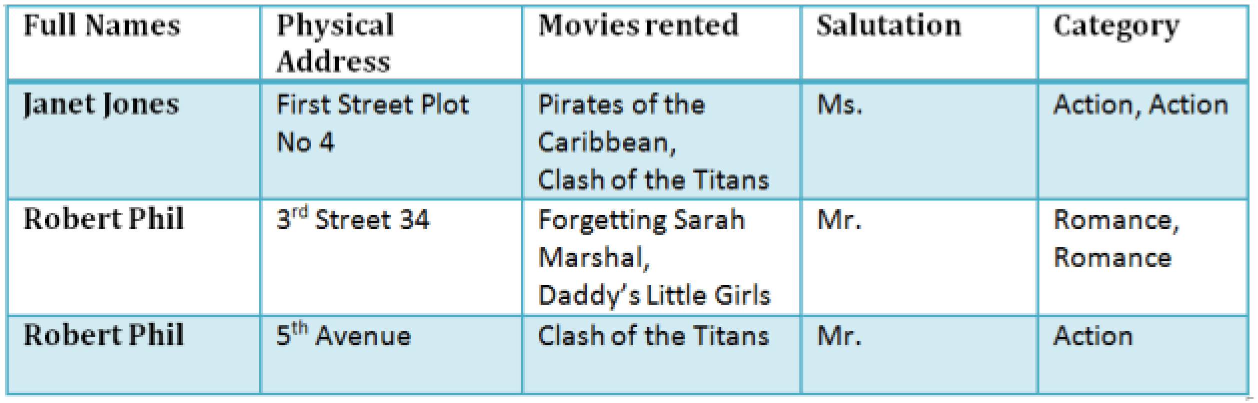
The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of First Normal Form and he continued to extend theory with Second and Third Normal Form. Later he joined with Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

Theory of Data Normalization in Sql is still being developed further. For example there are discussions even on 6th Normal Form. **But in most practical applications normalization achieves its best in 3rd Normal Form**. The evolution of Normalization theories is illustrated below-



Database Normalization Examples -

Assume a video library maintains a database of movies rented out. Without any normalization all information is stored in one table as shown below.



Full Names	Physical Address	Movies rented	Salutation	Category
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.	Action, Action
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.	Romance, Romance
Robert Phil	5th Avenue	Clash of the Titans	Mr.	Action

Table 1

Here you see **Movies Rented column has multiple values**.

Database Normal Forms

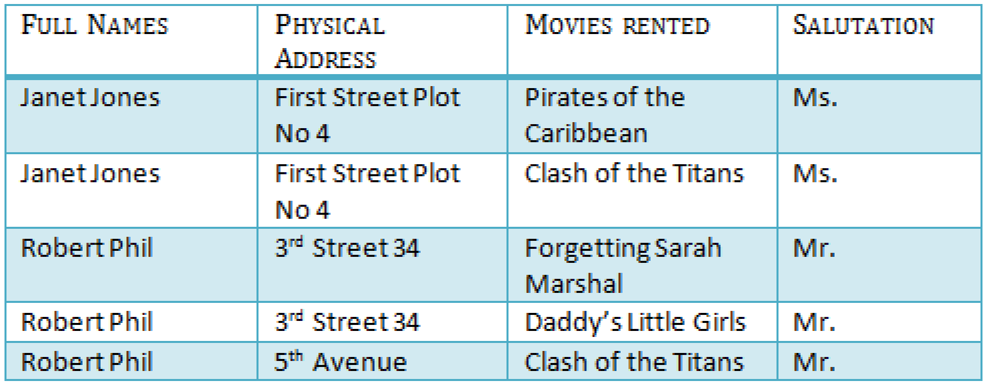
Now let's move in to 1st Normal Forms

1NF (First Normal Form) Rules

- Each table cell should contain single value.
- Each record needs to be unique.

The above table in 1NF-

1NF Exmple



Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Table 1 : In 1NF Form

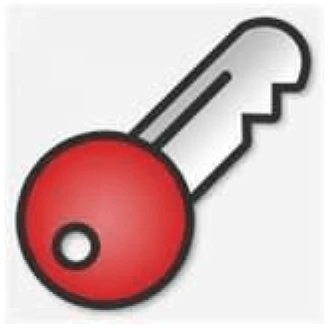
Before we proceed lets understand a few things --

What is a KEY ?

A KEY is a value used to uniquely identify a record in a table. A KEY could be a single column or combination of multiple columns

Note: Columns in a table that are NOT used to uniquely identify a record are called non-key columns.

What is a primary Key?



A primary is a single column values used to uniquely identify a database record.

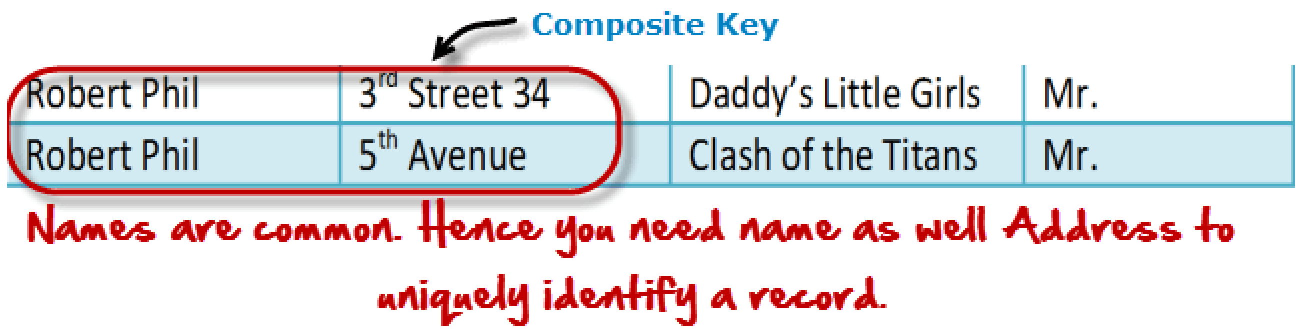
It has following attributes

- A primary key cannot be NULL
- A primary key value must be unique
- The primary key values can not be changed
- The primary key must be given a value when a new record is inserted.

What is Composite Key?

A composite key is a primary key composed of multiple columns used to identify a record uniquely

In our database , we have two people with the same name Robert Phil but they live at different places.


Composite Key

Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Names are common. Hence you need name as well Address to uniquely identify a record.

Hence we require both Full Name and Address to uniquely identify a record. This is a composite key.

Let's move into second normal form 2NF

2NF (Second Normal Form) Rules

- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Table 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

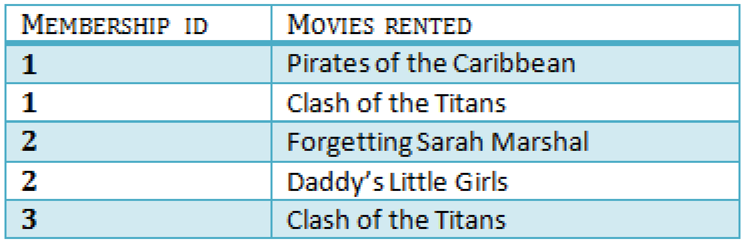
Table 2

We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.

We have introduced a new column called Membership_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id

Database - Foreign Key

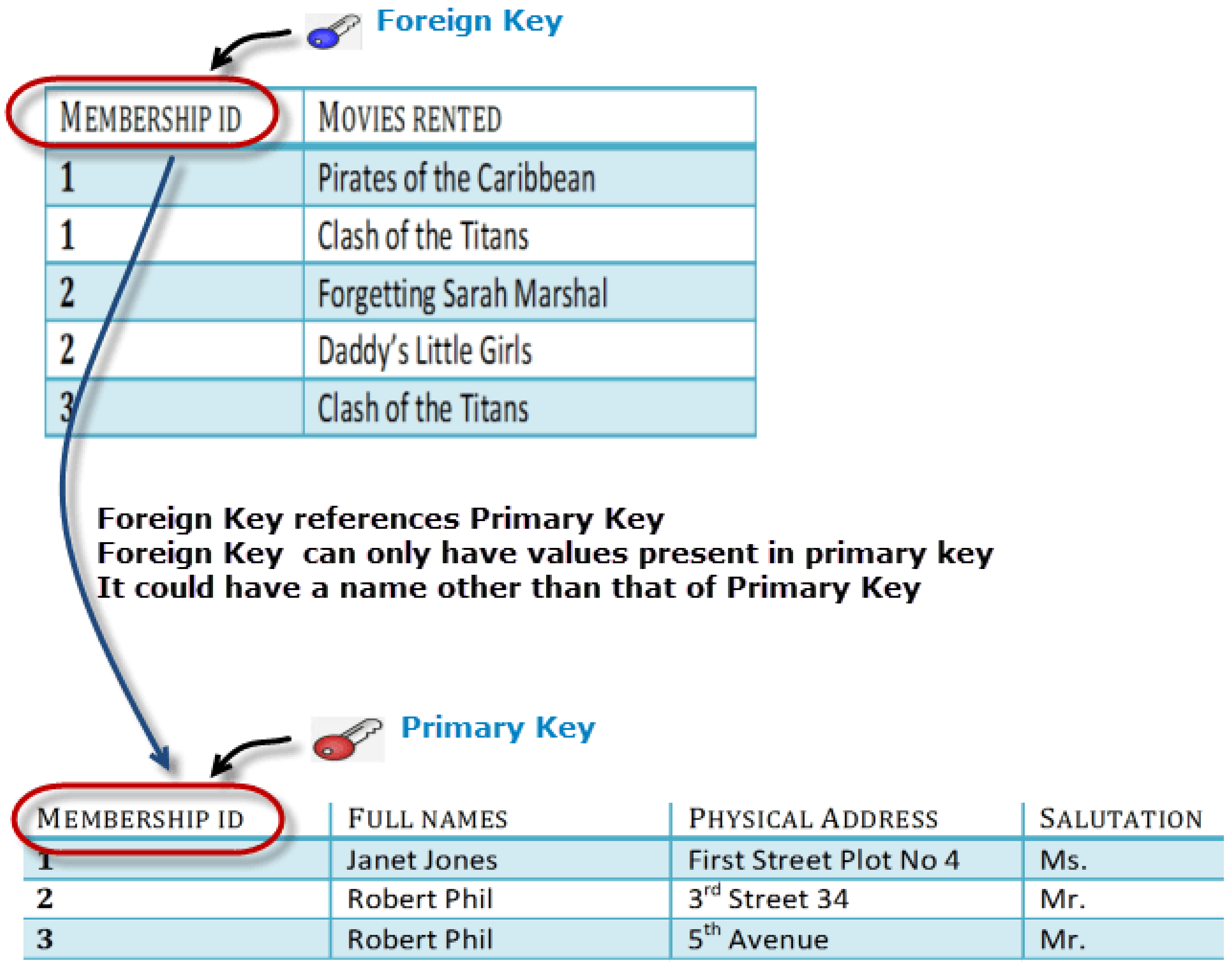
In Table 2, Membership_ID is the foreign Key



Membership id	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Foreign Key references primary key of another Table! It helps connect your Tables

- A foreign key can have a different name from its primary key
- It ensures rows in one table have corresponding rows in another
- Unlike Primary key they do not have to be unique. Most often they aren't
- Foreign keys can be null even though primary keys can not



MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

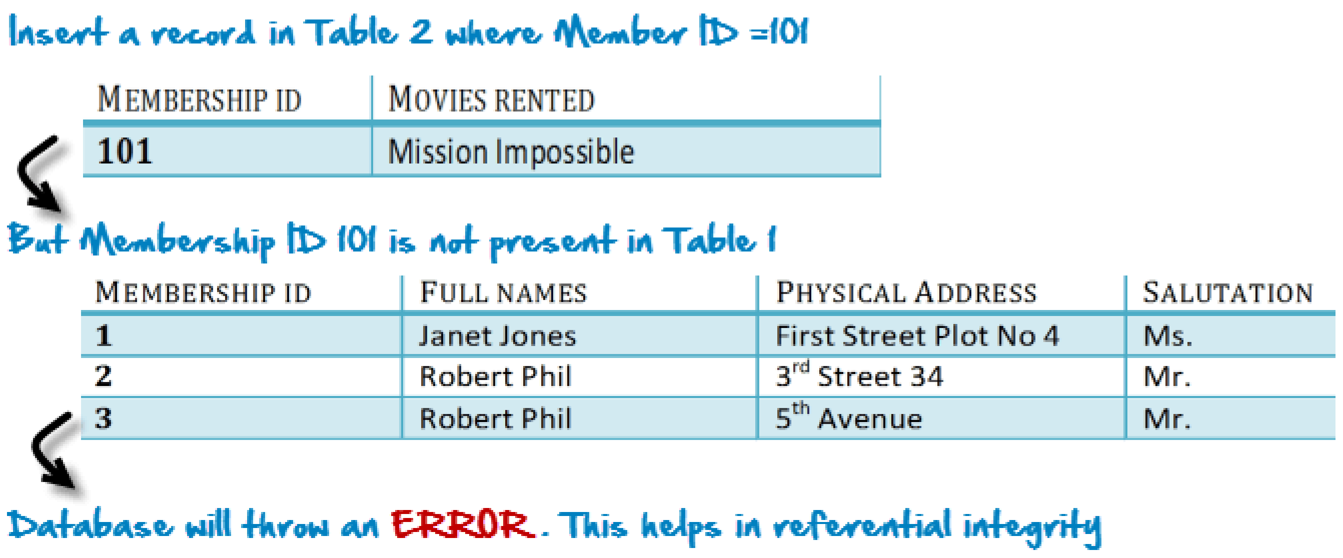
Foreign Key references Primary Key
Foreign Key can only have values present in primary key
It could have a name other than that of Primary Key

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Why do you need a foreign key ?

Suppose an idiot inserts a record in Table B such as

You will only be able to insert values into your foreign key that exist in the unique key in the parent table. This helps in referential integrity.



Insert a record in Table 2 where Member ID =101

MEMBERSHIP ID	MOVIES RENTED
101	Mission Impossible

But Membership ID 101 is not present in Table 1

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Database will throw an ERROR. This helps in referential integrity

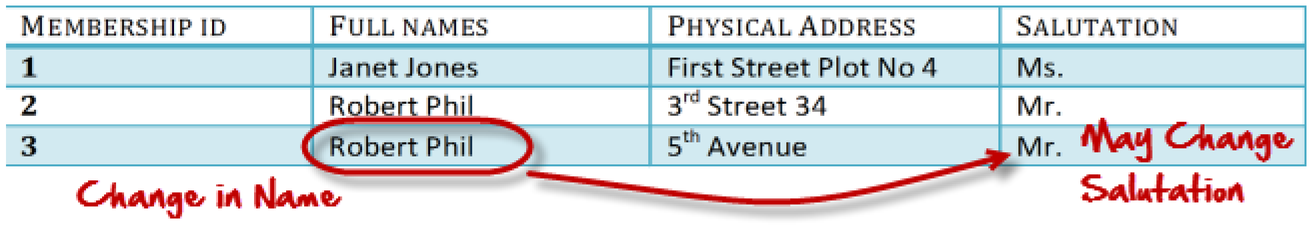
The above problem can be overcome by declaring membership id from Table2 as foreign key of membership id from Table1

Now , if somebody tries to insert a value in the membership id field that does not exist in the parent table , an error will be shown!

What is a transitive functional dependencies?

A transitive functional dependency is when changing a non-key column , might cause any of the other non-key columns to change

Consider the table 1. Changing the non-key column Full Name , may change Salutation.



MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Change in Name
May Change Salutation

Let's move ito 3NF

3NF (ThirdNormal Form) Rules

- Rule 1- Be in 2NF
- Rule 2- Has no transitive functional dependencies

To move our 2NF table into 3NF we again need to need divide our table.

3NF Example

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

TABLE 1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Table 2

SALUTATION ID	SALUTATION
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

Table 3

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies and hence our table is in 3NF

In Table 3 Salutation ID is primary key and in Table 1 Salutation ID is foreign to primary key in Table 3

Now our little example is in a level that cannot further be decomposed to attain higher forms of normalization. In fact it is already in higher normalization forms. Separate efforts for moving in to next levels of normalizing data are normally needed in complex databases. However we will be discussing about next levels of normalizations in brief in the following.

Boyce Codd Normal Form (BCNF)

Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has more than one **Candidate** Key.

Sometimes is BCNF is also referred as **3.5 Normal Form.**

4NF (Fourth Normal Form) Rules

If no database table instance contains two or more, independent and multivalued data describing the relevant entity , then it is in 4th Normal Form.

5NF (Fifth Normal Form) Rules

A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed in to any number of smaller tables without loss of data.

6NF (Sixth Normal Form) Proposed

6th Normal Form is not standardized yet however it is being discussed by database experts for some time. Hopefully we would have clear standardized definition for 6th Normal Form in near future.

That's all to Normalization!!!

Summary

- Database designing is critical to the successful implementation of a database management system that meets the data requirements of an enterprise system.
- Normalization helps produce database systems that are cost effective, cost effective and have better security models.
- Functional dependencies are a very important component of the normalize data process

- Most database systems are normalized database up to the third normal forms.
- A primary uniquely identifies are record in a Table and cannot be null
- A foreign key helps connect table and references a primary key

What is ER Modeling?

Entity Relationship Modeling (ER Modeling) is a graphical approach to database design. It uses Entity/Relationship to represent real world objects.

An **Entity** is a thing or object in real world that is distinguishable from surrounding environment. For example each employee of an organization is a separate entity. Following are some of major characteristics of entities.

- An entity has a set of properties.
- Entity properties can have values.

Let's consider our first example again. An employee of an organization is an entity. If "Peter" is a programmer (an **employee**) at Microsoft, he can have **attributes (**properties) like name, age, weight, height, etc. It is obvious that those do hold values relevant to him.

Each attribute can have **Values**. In most cases single attribute have one value. But it is possible for attributes have **multiple values** also. For example Peter's age has a single value. But his "phone numbers" property can have multiple values.

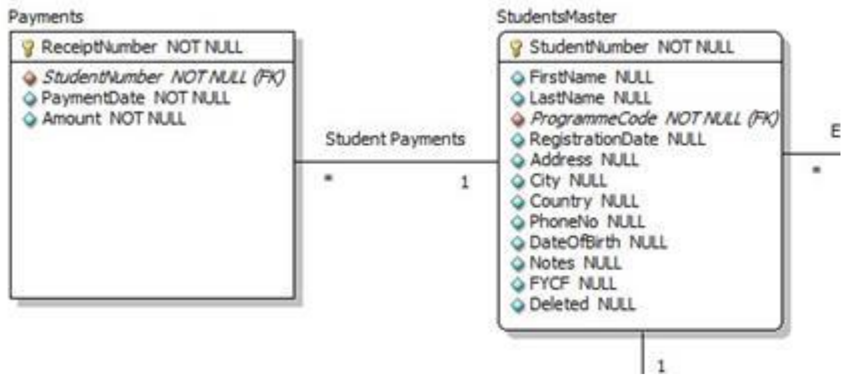
Entities can have **relationships** with each other. Let's consider a simplest example. Assume that each Microsoft Programmer is given a Computer. It is clear that that **Peter's Computer** is also an entity. Peter is using that computer and the same computer is used by Peter. In other words there is a mutual relationship among Peter and his computer.

In **Entity Relationship Modeling,** we model entities, their attributes and relationships among entities.

Enhanced Entity Relationship (EER) Model

Enhanced Entity Relationship (EER) Model is a high level data model which provides extensions to original **Entity Relationship** (ER) model. EER Models supports more details design. EER Modeling emerged as a solution for modeling highly complex databases.

EER uses UML notation. UML is the acronym for Unified Modeling Language; it is a general purpose modeling language used when designing object oriented systems. Entities are represented as class diagrams. Relationships are represented as associations between entities. The diagram shown below illustrates an ER diagram using the UML notation.



Why use ER Model?

Now you may think why use ER modeling when we can simply create the database and all of its objects without ER modeling? One of the challenges faced when designing database is the fact that designers, developers and end-users tend to view data and its usage differently. If this situation is left unchecked, we can end up producing a database system that does not meet the requirements of the users.

Communication tools understood by all stakeholders(technical as well non-technical users) are critical in producing database systems that meet the requirements of the users. ER models are examples of such tools.

ER diagrams also increase user productivity as they can be easily translated into relational tables.

Steps for Create Database Mysql

Create Database in two ways

1) By executing a simple SQL query

2) By using forward engineering in MySQL Workbench

As SQL beginner , let's look into the query method first.

Create Database

CREATE DATABASE is the SQL command for creating a database.

Imagine you need to create a database with name "movies". You can do it by executing following SQL command.

```
CREATE DATABASE movies;
```

Note: you can also use the command CREATE SCHEMA instead of CREATE DATABASE

Now let's improve our SQL query adding more parameters and specifications.

IF NOT EXISTS

A single MySQL server could have multiple databases. If you are not the only one accessing the same MySQL server or if you have to deal with multiple databases there is a probability of attempting to create a new database with name of an existing database . **IF NOT EXISTS** let you to instruct MySQL server to check the existence of a database with a similar name prior to creating database.

When **IF NOT EXISTS** is used database is created only if given name does not conflict with an existing database's name. Without the use of **IF NOT EXISTS** MySQL throws an error.

```
CREATE DATABASE IF NOT EXISTS movies;
```

Collation and Character Set

Collation is set of **rules used in comparison.** Many people use MySQL to store data other than English. Data is stored in MySQL using a specific character set. The character set can be defined at different levels viz, server , database , table and columns.

You need to select the rules of collation which in turn depend on the character set chosen.

For instance, the Latine1 character set uses the `latin1_swedish_ci` collation which is the Swedish case insensitive order.

```
CREATE DATABASE IF NOT EXISTS movies CHARACTER SET latin1 COLLATE lati
n1_swedish_ci
```

The best practice while using local languages like Arabic , Chinese etc is to select Unicode (utf-8) character set which has several collations or just stick to default collation utf8-general-ci.

You can find the list of all collations and character sets here here

You can see list of existing databases by running following SQL command.

```
SHOW DATABASES
```

Creating Tables MySQL

Tables can be created using **CREATE TABLE** statement and it actually has the following syntax.

```
CREATE  TABLE [IF NOT EXISTS] `TableName` (`fieldname` dataType [optiona
l parameters]) ENGINE = storage Engine;
```

HERE

- "CREATE TABLE" is the one responsible for the creation of the table in the database.
- "[IF NOT EXISTS]" is optional and only create the table if no matching table name is found.

- "`fieldName`" is the name of the field and "data Type" defines the nature of the data to be stored in the field.
- "[optional parameters]" additional information about a field such as " AUTO_INCREMENT" , NOT NULL etc

Create Table Example:-

```
CREATE  TABLE IF NOT EXISTS `MyFlixDB`.`Members` (
  `membership_number` INT  AUTOINCREMENT ,
  `full_names` VARCHAR(150) NOT NULL ,
  `gender` VARCHAR(6) ,
  `date_of_birth` DATE ,
  `physical_address` VARCHAR(255) ,
  `postal_address` VARCHAR(255) ,
  `contact_number` VARCHAR(75) ,
  `email` VARCHAR(255) ,
  PRIMARY KEY (`membership_number`) )
ENGINE = InnoDB;
```

Now let's see what the MySQL's data types are. You can use any of them depending on your need. You should always try to not to underestimate or overestimate potential range of data when creating a database.

DATA TYPES

Data types define the nature of the data that can be stored in a particular column of a table

MySQL has **3** main categories of data types namely

1. Numeric,
2. Text
3. Date/time.

Numeric Data types

Numeric data types are used to store numeric values. It is very important to make sure range of your data is between lower and upper boundaries of numeric data types.

TINYINT( )	-128 to 127 normal 0 to 255 UNSIGNED.

SMALLINT( )	-32768 to 32767 normal 0 to 65535 UNSIGNED.
MEDIUMINT( )	-8388608 to 8388607 normal 0 to 16777215 UNSIGNED.
INT( )	-2147483648 to 2147483647 normal 0 to 4294967295 UNSIGNED.
BIGINT( )	-9223372036854775808 to 9223372036854775807 normal 0 to 18446744073709551615 UNSIGNED.
FLOAT	A small approximate number with a floating decimal point.
DOUBLE(,)	A large number with a floating decimal point.
DECIMAL(,)	A DOUBLE stored as a string , allowing for a fixed decimal point. Choice for storing currency values.

Text Data Types

As data type category name implies these are used to store text values. Always make sure you length of your textual data do not exceed maximum lengths.

CHAR( )	A fixed section from 0 to 255 characters long.
VARCHAR( )	A variable section from 0 to 255 characters long.
TINYTEXT	A string with a maximum length of 255 characters.
TEXT	A string with a maximum length of 65535 characters.
BLOB	A string with a maximum length of 65535 characters.
MEDIUMTEXT	A string with a maximum length of 16777215 characters.
MEDIUMBLOB	A string with a maximum length of 16777215 characters.

LONGTEXT	A string with a maximum length of 4294967295 characters.
LONGBLOB	A string with a maximum length of 4294967295 characters.

Date / Time

DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS

Apart from above there are some other data types in MySQL.

ENUM	To store text value chosen from a list of predefined text values
SET	This is also used for storing text values chosen from a list of predefined text values. It can have multiple values.
BOOL	Synonym for TINYINT(1), used to store Boolean values
BINARY	Similar to CHAR, difference is texts are stored in binary format.
VARBINARY	Similar to VARCHAR, difference is texts are stored in binary format.

Now let's see a sample SQL query for creating a table which has data of all data types. Study it and identify how each data type is defined.

```
CREATE TABLE`all_data_types` (
  `varchar` VARCHAR( 20 )  ,
  `tinyint` TINYINT  ,
  `text` TEXT  ,
  `date` DATE  ,
  `smallint` SMALLINT  ,
  `mediumint` MEDIUMINT  ,
  `int` INT  ,
  `bigint` BIGINT  ,
  `float` FLOAT( 10, 2 )  ,
  `double` DOUBLE  ,
  `decimal` DECIMAL( 10, 2 )  ,
  `datetime` DATETIME  ,
  `timestamp` TIMESTAMP  ,
  `time` TIME  ,
  `year` YEAR  ,
  `char` CHAR( 10 )  ,
  `tinyblob` TINYBLOB  ,
  `tinytext` TINYTEXT  ,
  `blob` BLOB  ,
  `mediumblob` MEDIUMBLOB  ,
  `mediumtext` MEDIUMTEXT  ,
  `longblob` LONGBLOB  ,
  `longtext` LONGTEXT  ,
  `enum` ENUM( '1', '2', '3' )  ,
   `set` SET( '1', '2', '3' )  ,
   `bool` BOOL  ,
   `binary` BINARY( 20 )  ,
   `varbinary` VARBINARY( 20 )
) ENGINE= MYISAM ;
```

Best practices

- Use upper case letters for SQL keywords i.e. "DROP SCHEMA IF EXISTS `MyFlixDB`;"
- End all your SQL commands using semi colons.
- Avoid using spaces in schema, table and field names. Use underscores instead to separate schema, table or field names.

So

- Creating a database involves translating the logical database design model into the physical database.
- MySQL supports a number of data types for numeric, dates and strings values.
- CREATE DATABASE command is used to create a database
- CREATE TABLE command is used to create tables in a database

MySQL SELECT Statement with Examples

Databases store data for later retrieval. Ever wondered how that is achieved? It's the **SELECT** SQL command that does the job.

That's what it's all about, retrieving data from the database tables. It's part of the **data manipulation language** that is responsible for **query the data from the database**.

SQL SELECT statement syntax

It is the most frequently used SQL command and has the following general syntax

```
SELECT [DISTINCT|ALL ] { * | [fieldExpression [AS newName]} FROM tableNa
me [alias] [WHERE condition][GROUP BY fieldName(s)]  [HAVING condition] OR
DER BY fieldName(s)
```
HERE

- **SELECT** is the SQL keyword that lets the database know that you want to retrieve data.
- **[DISTINCT | ALL]** are optional keywords that can be used to fine tune the results returned from the SQL SELECT statement. If nothing is specified then ALL is assumed as the default.
- **{*| [fieldExpression [AS newName]}** at least one part must be specified, "*" selected all the fields from the specified table name, fieldExpression performs some computations on the specified fields such as adding numbers or putting together two string fields into one.
- **FROM** tableName is mandatory and must contain at least one table, multiple tables must be separated using commas or joined using the JOIN keyword.
- **WHERE** condition is optional, it can be used to specify criteria in the result set returned from the query.
- **GROUP BY** is used to put together records that have the same field values.
- **HAVING** condition is used to specify criteria when working using the GROUP BY keyword.
- **ORDER BY** is used to specify the sort order of the result set.

*

The Star symbol is used to select all the columns in table. An example of a simple SELECT statement looks like the one shown below.

```
SELECT * FROM `members`;
```

The above statement selects all the fields from the members table. The semi-colon is a statement terminate. It's not mandatory but is considered a good practice to end your statements like that.

Practical examples

Click to download the myflix DB used for practical examples.

You can learn to import the .sql file into MySQL WorkBench

The Examples are performed on the following two tables Table 1: **members** table

membership_number	**full_names**	**gender**	**date_of_**	**physical_addres**	**postal_addre**	**contct_numb**	**email**

			birth	**s**	**ss**	**er**	
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-198	2nd Street	NULL	NULL	NULL

			4	23			

Table 2: **movies** table

movie_id	**title**	**director**	**year_released**	**category_id**
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
3	X-Men	NULL	2008	NULL
4	Code Name Black	Edgar Jimz	2010	NULL
5	Daddy's Little Girls	NULL	2007	8

6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8
16	67% Guilty	NULL	2012	NULL

Getting members listing

Let's suppose that we want to get a list of all the registered library members from our database, we would use the script shown below to do that.

```
SELECT * FROM `members`;
```

Executing the above script in MySQL workbench produces the following results.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email

1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL

Our above query has returned all the rows and columns from the members table.

Let's say we are only interested in getting only the full_names, gender, physical_address and email fields only. The following script would help us to achieve this.

```
SELECT `full_names`,`gender`,`physical_address`, `email` FROM `members`;
```

Executing the above script in MySQL workbench produces the following results.

full_names	gender	physical_address	email
Janet Jones	Female	First Street Plot No 4	janetjones@yagoo.cm
Janet Smith Jones	Female	Melrose 123	jj@fstreet.com
Robert Phil	Male	3rd Street 34	rm@tstreet.com
Gloria Williams	Female	2nd Street 23	NULL

Getting movies listing

Remember in our above discussion that we mention expressions been used in SELECT statements. Let's say we want to get a list of movie from our database. We want to have the movie title and the name of the movie director in one field. The name of the movie director should be in brackets. We also want to get the year that the movie was released. The following script helps us do that.

```
SELECT Concat(title, ' (', `director`, ')') , `year_released` FROM `movies`;
```

HERE

- The Concat () MySQL function is used join the columns values together.
- The line "Concat (`title`, ' (', `director`, ')') gets the title, adds an opening bracket followed by the name of the director then adds the closing bracket.

String portions are separated using commas in the Concat () function.

Executing the above script in MySQL workbench produces the following result set.

Concat(`title`, ' (', `director`, ')')	**year_released**
Pirates of the Caribean 4 (Rob Marshall)	2011
Forgetting Sarah Marshal (Nicholas Stoller)	2008
NULL	2008
Code Name Black (Edgar Jimz)	2010
NULL	2007
NULL	2007
NULL	2007
Honey mooners (John Schultz)	2005
NULL	2012

Alias field names

The above example returned the Concatenation code as the field name for our results. Suppose we want to use a more descriptive field name in our result set. We would use the column alias name to achieve that. The following is the basic syntax for the column alias name

```
SELECT `column_name|value|expression` [AS] `alias_name`;
```

HERE

- ***"SELECT ` column_name|value|expression `"*** is the regular SELECT statement which can be a column name, value or expression.
- ***"[AS]"*** is the optional keyword before the alias name that denotes the expression, value or field name will be returned as.
- ***"`alias_name`"*** is the alias name that we want to return in our result set as the field name.

The above query with a more meaningful column name

```
SELECT Concat(`title`, ' (', `director`, ')') AS 'Concat', `year_released` FROM `movies`;
```

We get the following result

Concat	**year_released**
Pirates of the Caribean 4 (Rob Marshall)	2011
Forgetting Sarah Marshal (Nicholas Stoller)	2008

NULL	2008
Code Name Black (Edgar Jimz)	2010
NULL	2007
NULL	2007
NULL	2007
Honey mooners (John Schultz)	2005
NULL	2012

Getting members listing showing the year of birth

Suppose we want to get a list of all the members showing the membership number, full names and year of birth, we can use the LEFT string function to extract the year of birth from the date of birth field. The script shown below helps us to do that.

```
SELECT `membership_number`,`full_names`,LEFT(`date_of_birth`,4) AS `ye
ar_of_birth` FROM members;
```

HERE

- ***"LEFT(`date_of_birth`,4)"*** the **LEFT string function** accepts the date of birth as the parameter and only returns 4 characters from the left.

- ***"AS `year_of_birth`"*** is the **column alias name** that will be returned in our results. Note the ***AS keyword is optional***, you can leave it out and the query will still work.

Executing the above query in MySQL workbench against the myflixdb gives us the results shown below.

membership_number	full_names	year_of_birth
1	Janet Jones	1980
2	Janet Smith Jones	1980
3	Robert Phil	1989
4	Gloria Williams	1984

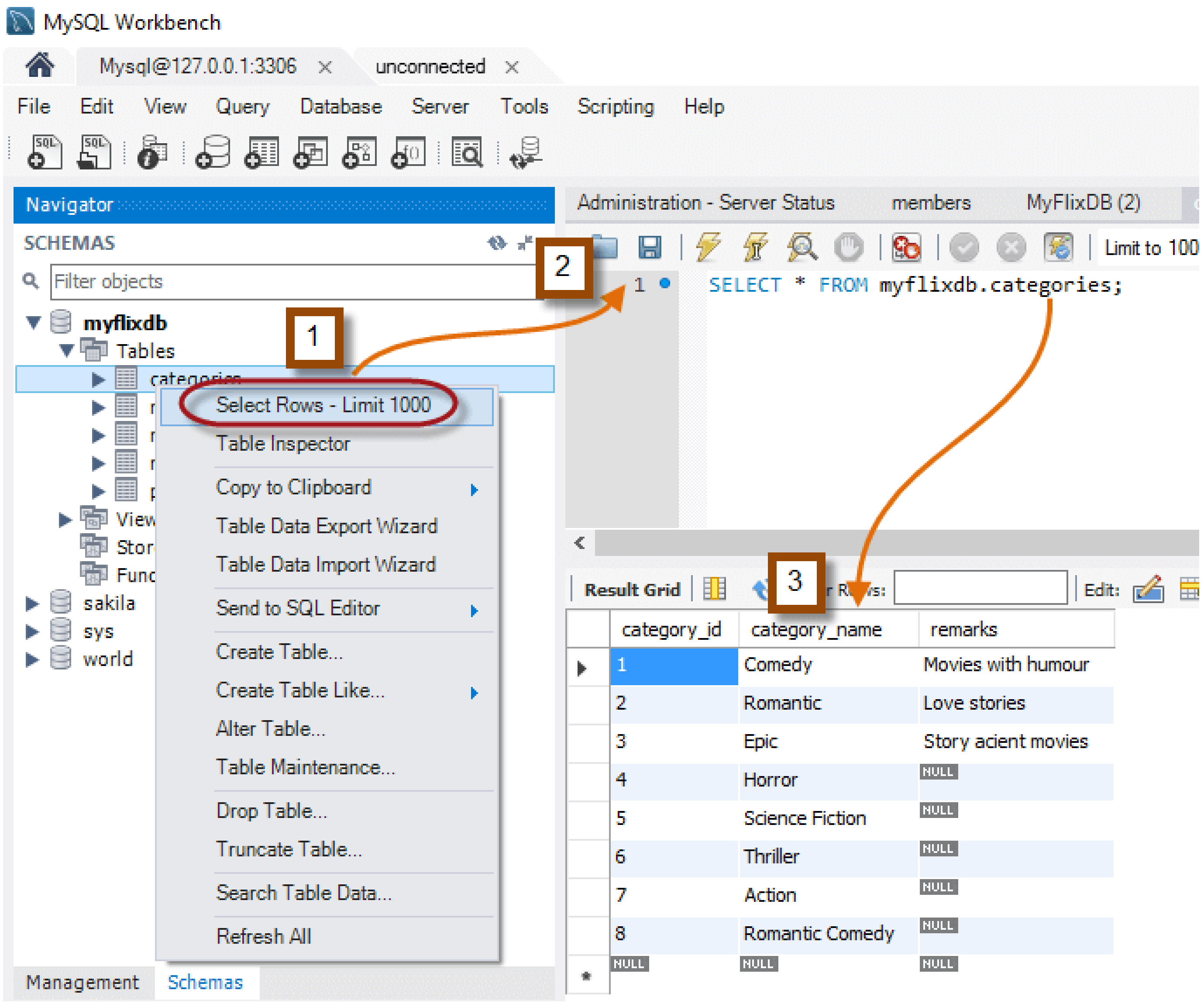
SQL using MySQL Workbench

We are now going to use MySQL workbench to generate the script that will display all the field names from our categories table.

1. Right Click on the Categories Table. Click on "Select Rows - Limit 1000"

2. MySQL workbench will automatically create a SQL query and paste in the editor.

3. Query Results will be show



```
SELECT * FROM myflixdb.categories;
```

category_id	category_name	remarks
1	Comedy	Movies with humour
2	Romantic	Love stories
3	Epic	Story acient movies
4	Horror	NULL
5	Science Fiction	NULL
6	Thriller	NULL
7	Action	NULL
8	Romantic Comedy	NULL
NULL	NULL	NULL

Notice that we didn't write the SELECT statement ourselves. MySQL workbench generated it for us.

Why use the SELECT SQL command when we have MySQL Workbench?

Now, you might be thinking why learn the SQL SELECT command to query data from the database when you can simply use a tool such as MySQL workbench's to get the same results without knowledge of the SQL language. Of course that is possible, but **learning how to use the SELECT command** gives you more **flexibility** and **control** over your **SQL SELECT statements**.

MySQL workbench falls in the category of "**Query by Example**" QBE tools. It's intended to help generate SQL statements faster to increase the user productivity.

Learning the SQL SELECT command can enable you to create **complex queries** that cannot be easily generated using Query by Example utilities such as MySQL workbench.

To improve productivity you can **generate the code using MySQL workbench** then **customize** it to **meet your requirements**. This can only happen if you understand how the SQL statements work!

Summary

- The SQL SELECT keyword is used to query data from the database and it's the most commonly used command.
- The simplest form has the syntax "SELECT * FROM tableName;"
- Expressions can also be used in the select statement . Example "SELECT quantity + price FROM Sales"
- The SQL SELECT command can also have other optional parameters such as WHERE, GROUP BY, HAVING, ORDER BY. They will be discussed later.
- MySQL workbench can help develop SQL statements, execute them and produce the output result in the same window.

MySQL WHERE Clause with Examples - AND, OR, IN, NOT IN

What is the WHERE Clause?

We looked at how to query data from a database using the SELECT statement in the previous tutorial. The SELECT statement returned all the results from the queried database table.

They are however, times when we want to restrict the query results to a specified condition. The SQL WHERE clause comes in handy in such situations.

SELECT * FROM PRESENTS WHERE CONTENTS ARE Toys or Chocolates

WHERE clause Syntax

The basic syntax for the WHERE clause when used in a SELECT statement is as follows.

```
SELECT * FROM tableName WHERE condition;
```

HERE

- **"SELECT * FROM tableName"** is the standard SELECT statement
- **"WHERE"** is the keyword that restricts our select query result set and **"condition"** is the filter to be applied on the results. The filter could be a range, single value or sub query.

Let's now look at a **practical example**.

Suppose we want to get a member's personal details from members table given the membership number 1, we would use the following script to achieve that.

```
SELECT * FROM `members` WHERE `membership_number` = 1;
```

Executing the above script in MySQL workbench on the "myflixdb" would produce the following results.

membership	full_n	gen	date_of	physical_	postal_a	contct_n	email

_number	ames	der	_birth	address	ddress	umber	
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm

WHERE clause combined with - *AND* LOGICAL Operator

The WHERE clause when used together with the AND logical operator, is only executed if ALL filter criteria specified are met.

Let's now look at a practical example - Suppose we want to get a list of all the movies in category 2 that were released in 2008, we would use the script shown below is achieve that.

```
SELECT * FROM `movies` WHERE `category_id` = 2 AND `year_released` = 2
008;
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

movie_id	title	director	year_released	category_id
2	Forgetting Sarah	Nicholas	2008	2

	Marshal	Stoller		

WHERE clause combined with - *OR* LOGICAL Operator

The WHERE clause when used together with the OR operator, is only executed if any or the entire specified filter criteria is met.
The following script gets all the movies in either category 1 or category 2

```
SELECT * FROM `movies` WHERE `category_id` = 1 OR `category_id` = 2;
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

movie_id	**title**	**director**	**year_released**	**category_id**
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2

WHERE clause combined with - *IN* Keyword

The WHERE clause when used together with the IN keyword only affects the rows whose values matches the list of values provided in the IN keyword. IN helps reduces number of OR clauses you may have to use
The following query gives rows where membership_number is either 1 , 2 or 3

```
SELECT * FROM `members` WHERE `membership_number` IN (1,2,3);
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

membership_number	**full_names**	**gender**	**date_of_birth**	**physical_address**	**postal_address**	**contct_number**	**email**
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com

WHERE clause combined with - *NOT IN* Keyword

The WHERE clause when used together with the NOT IN keyword DOES NOT affects the rows whose values matches the list of values provided in the NOT IN keyword.
The following query gives rows where membership_number is NOT 1 , 2 or 3

```
SELECT * FROM `members` WHERE `membership_number` NOT IN (1,2,3);
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL

WHERE clause combined with - COMPARISON OPERATORS

The less than (), equal to (=), not equal to () comparison operators can be used with the Where clause

= Equal To

The following script gets all the female members from the members table using the equal to comparison operator.

```
SELECT * FROM `members` WHERE `gender` = 'Female';
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
4	Gloria	Fem	14-02-	2nd Street	NULL	NULL	NULL

	Williams	ale	1984	23

> Greater than

The following script gets all the payments that are greater than 2,000 from the payments table.
SELECT * FROM `payments` WHERE `amount_paid` > 2000;
Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

payment_id	**membership_number**	**payment_date**	**description**	**amount_paid**	**external_reference_number**
1	1	23-07-2012	Movie rental payment	2500	11
3	3	30-07-2012	Movie rental payment	6000	NULL

< > Not Equal To

The following script gets all the movies whose category id is not 1.

```
SELECT * FROM `movies` WHERE `category_id`<> 1;
```

Executing the above script in MySQL workbench against the "myflixdb" produces the following results.

movie_id	title	director	year_released	category_id
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8

Summary

- The SQL WHERE clause is used to restrict the number of rows affected by a SELECT, UPDATE or DELETE query.
- The WHERE clause can be used in conjunction with logical operators such as AND and OR, comparison operators such as ,= etc.
- When used with the AND logical operator, all the criteria must be met.
- When used with the OR logical operator, any of the criteria must be met.
- The key word IN is used to select rows matching a list of values.

Brain Teaser

Let's suppose that we want to get a list of rented movies that have not been returned on time 24/06/2012.

We can use the WHERE clause together with the less than comparison operator and AND logical operator to achieve that.

```
SELECT * FROM `movierentals` WHERE `return_date` < '2012-06-25' AND mo
vie_returned = 0;
```

Executing the above script in MySQL workbench gives the following results.

reference_number	**transaction_date**	**return_date**	**membership_number**	**movie_id**	**movie_returned**

14	21-06-2012	24-06-2012	2	2	0

MySQL query INSERT INTO Table with Examples

What is INSERT INTO?

The main goal of database systems is to store data in the tables. The data is usually supplied by application programs that run on top of the database. Towards that end, SQL has the INSERT command that is used to store data into a table. The **INSERT command creates a new row** in the table to store data.

Basic syntax

Let's look at the basic syntax of the SQL INSERT command shown below.

```
INSERT INTO `table_name`(column_1,column_2,...) VALUES (value_1,value_2,...);
```

HERE

- **INSERT INTO `table_name`** is the command that tells MySQL server to add new row into a table named `table_name`.

- **(column_1,column_2,...)** specifies the columns to be updated in the new row
- **VALUES (value_1,value_2,...)** specifies the values to be added into the new row

When supplying the data values to be inserted into the new table, the following should be considered while dealing with different data types.

- **String data types** - all the string values should be enclosed in single quotes.
- **Numeric data types** - all numeric values should be supplied directly without enclosing them in single or double quotes.
- **Date data types** - enclose date values in single quotes in the format 'YYYY-MM-DD'.

Example:

Suppose that we have the following list of new library members that need to be added into the database.

Full names	Date of Birth	gender	Physical address	Postal address	Contact number	Email Address
Leonard Hofstadter		Male	Woodcrest		0845738767	
Sheldon Cooper		Male	Woodcrest		0976736763	
Rajesh Koothrappali		Male	Fairview		0938867763	
Leslie Winkle	14/02/1984	Male			0987636553	
Howard Wolowitz	24/08/1981	Male	South Park	P.O. Box 4563	0987786553	lwolowitz@email.me

Lets' INSERT data one by one. We will start with Leonard Hofstadter. We will treat the contact number as a numeric data type and not enclose the number in single quotes.

```
INSERT INTO `members` (`full_names`,`gender`,`physical_address`,`contac
t_number`) VALUES ('Leonard Hofstadter','Male','Woodcrest',0845738767);
```

Executing the above script drops the 0 from Leonard's contact number. This is because the value will be treated as a numeric value and the zero (0) at the beginning is dropped since it's not significant.

In order to avoid such problems, the value must be enclosed in single quotes as shown below -

```
INSERT INTO `members` (`full_names`,`gender`,`physical_address`,`contac
t_number`) VALUES ('Sheldon Cooper','Male','Woodcrest', '0976736763');
```

In the above case , zero(0) will not be dropped

Changing the order of the columns has no effect on the INSERT query as long as the correct values have been mapped to the correct columns.

The query shown below demonstrates the above point.

```
INSERT INTO `members` (`contact_number`,`gender`,`full_names`,`physica
l_address`)VALUES ('0938867763','Male','Rajesh Koothrappali','Woodcrest');
```

The above queries skipped the date of birth column, **by default MySQL will insert NULL values in columns that are skipped in the INSERT query.**

Let's now insert the record for Leslie which has the date of birth supplied. **The date value should be enclosed in single quotes using the format 'YYYY-MM-DD'.**

```
INSERT INTO `members` (`full_names`,`date_of_birth`,`gender`,`physical_a
ddress`,`contact_number`) VALUES ('Leslie Winkle','1984-02-14','Male','Wood
crest', '0987636553');
```

All of the above queries specified the columns and mapped them to values in the insert statement.**If we are supplying values for ALL the columns in the table, then we can omit the columns from the insert query.**

Example:-

```
INSERT INTO `members` VALUES (9,'Howard Wolowitz','Male','1981-08-24','So
uthPark','P.O. Box 4563', '0987786553', 'lwolowitz[at]email.me');
```

Let's now use the SELECT statement to view all the rows in the members table.SELECT * FROM `members`;

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL
5	Leonard Hofstadter	Male	NULL	Woodcrest	NULL	845738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL

7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	lwolowitz@email.me

Notice the contact number for Leonard Hofstadter has dropped the zero (0) from the contact number. The other contact numbers have not dropped the zero (0) at the beginning.

Inserting into a Table from another Table

The INSERT command can also be used to insert data into a table from another table. The basic syntax is as shown below.

```
INSERT INTO table_1 SELECT * FROM table_2;
```

Let's now look at a practical example, we will create a dummy table for movie categories for demonstration purposes. We will call the new categories table categories_archive. The script shown below creates the table.

```
CREATE TABLE `categories_archive` (           `category_id` int(11) AUTO_INC
REMENT,            `category_name` varchar(150) DEFAULT NULL,           `re
marks` varchar(500) DEFAULT NULL,            PRIMARY KEY (`category_id`)         )
```

Execute the above script to create the table.

Let's now insert all the rows from the categories table into the categories archive table. The script shown below helps us to achieve that.

```
INSERT INTO `categories_archive` SELECT * FROM `categories`;
```

Executing the above script inserts all the rows from the categories table into the categories archive table. Note the table structures will have to be the same for the script to work. A more robust script is one that maps the column names in the insert table to the ones in the table containing the data.

The query shown below demonstrates its usage.

```
INSERT INTO `categories_archive`(category_id,category_name,remarks)  SELE
CT category_id,category_name,remarks FROM `categories`;
```

Executing the SELECT query

```
SELECT * FROM `categories_archive`
```

gives the following results shown below.

category_id	**category_name**	**remarks**
1	Comedy	Movies with humour
2	Romantic	Love stories
3	Epic	Story acient movies
4	Horror	NULL
5	Science Fiction	NULL
6	Thriller	NULL
7	Action	NULL
8	Romantic Comedy	NULL
9	Cartoons	NULL
10	Cartoons	NULL

Summary

- The INSERT command is used to add new data into a table
- The date and string values should be enclosed in single quotes.
- The numeric values do not need to be enclosed in quotes.
- The INSERT command can also be used to insert data from one table into another.

MySQL UPDATE & DELETE Query with Example

What is the DELETE Keyword?

The SQL DELETE command is used to delete rows that are no longer required from the database tables. It deletes the whole row from the table. Delete command comes in handy to delete temporary or obsolete data from your database. The DELETE command can delete more than one row from a table in a single query. This proves to be advantages when removing large numbers of rows from a database table.

Once a row has been deleted, it cannot be recovered. It is therefore strongly recommended to make database backups before deleting any data from the database. This can allow you to restore the database and view the data later on should it be required.

Delete command syntax

The basic syntax of the delete command is as shown below.

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

HERE

- DELETE FROM `table_name` tells MySQL server to remove rows from the table ..
- [WHERE condition] is optional and is used to put a filter that restricts the number of rows affected by the DELETE query.

If the WHERE clause is not used in the DELETE query, then all the rows in a given table will be deleted. Before we go into more details discussion the DELETE command, let's insert some sample data into the movies table to work with.

```
INSERT INTO  `movies` (`title`, `director`, `year_released`, `category_id`)
VALUES ('The Great Dictator', 'Chalie Chaplie', 1920, 7);
INSERT INTO `movies` (`title`, `director`, `category_id`) VALUES ('sample m
ovie', 'Anonymous', 8);
INSERT INTO  movies (`title`, `director`, `year_released`, `category_id`) VA
LUES ('movie 3', 'John Brown', 1920, 8);
```

Executing the above script adds three (3) movies into the movies table. Before we go any further into our lesson, let's get all the movies in our table. The script shown below does that.

```
SELECT * FROM `movies`;
```

Executing the above script gives us the following results.

movie_id	**itle**	**director**	**year_released**	**category_id**
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2

3	X-Men	NULL	2008	NULL
4	Code Name Black	Edgar Jimz	2010	NULL
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8
16	67% Guilty	NULL	2012	NULL
18	The Great Dictator	Chalie Chaplie	1920	7
19	sample movie	Anonymous	NULL	8
20	movie 3	John Brown	1920	8

Let's suppose that the Myflix video library no longer wishes to be renting out "The Great Dictator" to its members and they want it removed from the database. Its movie id is 18, we can use the script shown below to delete its row from the movies table.

```
DELETE FROM `movies` WHERE `movie_id` = 18;
```

Executing the above script in MySQL WorkBench against the Myflix deletes the movie with id 18 from the database table.

Let's see the current status of movies table.

```
SELECT * FROM `movies`;
```

movie_id	**title**	**director**	**year_released**	**category_id**
1	Pirates of the Caribean 4	Rob Marshall	2011	1
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
3	X-Men	NULL	2008	NULL

4	Code Name Black	Edgar Jimz	2010	NULL
5	Daddy's Little Girls	NULL	2007	8
6	Angels and Demons	NULL	2007	6
7	Davinci Code	NULL	2007	6
9	Honey mooners	John Schultz	2005	8
16	67% Guilty	NULL	2012	NULL
19	sample movie	Anonymous	NULL	8
20	movie 3	John Brown	1920	8

NOTE:

- *the movie with id 18 has not been return in the query result set.*
- *you cannot delete a single column for a table. You can delete an entire row.*

Let's say we have a list of movies we want to delete . We can use the WHERE clause along with IN.

```
DELETE FROM `movies` WHERE `movie_id`  IN (20,21);
```

Executing the above script deletes movies with IDs 20 and 21 from our movies table.

WHAT IS THE UPDATE COMMAND?

The Update command is used to modify rows in a table. The update command can be used to update a single field or multiple fields at the same time. It can also be used to update a table with values from another table .

Update command syntax

The basic syntax of the SQL Update command is as shown below.

```
UPDATE `table_name` SET `column_name` = `new_value' [WHERE condition]
;
```

HERE

- UPDATE `table_name` is the command that tells MySQL to update the data in a table .
- SET `column_name` = `new_value' are the names and values of the fields to be affected by the update query. Note, when setting the update values, strings data types must be in single quotes. Numeric values do not need to be in quotation marks. Date data type must be in single quotes and in the format 'YYYY-MM-DD'.
- [WHERE condition] is optional and can be used to put a filter that restricts the number of rows affected by the DELETE query.

Let's now look at a practical example that updates data in the members table. Let's suppose that our member's membership numbers 1 and 2 have the following updates to be made to their data records.

Membership number	Updates required
1	Changed contact number from 999 to 0759 253 532
2	Change the name to Janet Smith Jones and physical address should be updated to Melrose 123

We will start with making updates for membership number 1 before we make any updates to our data, let's retrieve the record for membership number 1. The script shown below helps us to do that.

```
SELECT * FROM `members` WHERE `membership_number` = 1;
```

Executing the above script gives us the following results.

membership_number	**full_names**	**gender**	**date_of_birth**	**physical_address**	**postal_address**	**contct_number**	**email**
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	999	janetjones@yagoo.cm

Let's now update the contact number using the script shown below.

```
UPDATE `members` SET `contact_number` = '0759 253 542' WHERE `membe
rship_number` = 1;
```

Executing the above script updates the contact number from 999 to 0759 253 532 for membership number 1. Let's now look at the record for membership number 1 after executing the update script.

```
SELECT * FROM `members` WHERE `membership_number` = 1;
```

Executing the above script gives us the following results.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm

Let's now look at the updates required for membership number 2.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email

2	Smith Jones	Female	23-06-1980	Park Street	NULL	NULL	jj@fstreet.com

The following script helps us to do that.

```
UPDATE `members` SET `full_names` = 'Janet Smith Jones', `physical_addres
s` = 'Melrose 123' WHERE `membership_number` = 2;
```

Executing the above script in updates the full names for membership number 2 to Janet Smith Jones and the physical address to Melrose 123.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com

Summary

- The delete command is used to remove data that is no longer required from a table.
- The "WHERE clause" is used to limit the number of rows affected by the DELETE query.
- Once data has been deleted, it cannot be recovered, it is therefore strongly recommend make backups before deleting data.
- The update command is used to modify existing data.
- The "WHERE clause" is used to limit the number of rows affected by the UPDATE query.

ORDER BY in MySQL: DESC & ASC

Sorting Results

We looked at how to get data from our tables using the SELECT command. Results were returned in the same order the records were added into the database. This is the default sort order. In this section, we will be looking at how we can sort our query results. **Sorting is simply re-arranging our query results in a specified way**. Sorting can be performed on a single column or on more than one column. It can be done on number, strings as well as date data types.

Order by clause

The order by clause is used to sort the query result sets in either ascending or descending order. It is used in conjunction with the SELECT query. It has the following basic syntax.

```
SELECT statement... [WHERE condition | GROUP BY `field_name(s)` HAVING c
ondition] ORDER BY `field_name(s)` [ASC | DESC];
```

HERE

- **"SELECT statement..."** is the regular select query
- **" | "** represents alternatives
- **"[WHERE condition | GROUP BY `field_name(s)` HAVING condition"** is the optional condition used to filter the query result sets.
- **"ORDER BY"** performs the query result set sorting
- **"[ASC | DESC]"** is the keyword used to sort result sets in either ascending or descending order. Note ***ASC*** is used as the default.

What are DESC and ASC Keywords?

⬆ **ASC is the short form for ascending**	⬇ **DESC is the short form for descending**
It is used to sort the query results in a top to bottom style.	It is used to sort the query results in a bottom to top style
When working on date data types, the earliest date is shown on top of the list.	. When working on date types, the latest date is shown on top of the list.
When working with numeric data types, the lowest values are shown on top of the list.	When working with numeric data types, the highest values are shown at top of the query result set.
When working with string data types, the query result set is sorted from those starting with the When working with string data types, the query result set is sorted from those starting with the letter Z going down to the letter A	When working with string data types, the query result set is sorted from those starting with the letter Z going down to the letter A.

letter A going up to the letter Z.	

Both the DESC and ASC keywords are used together in conjunction with the SELECT statement and the ORDER BY clause.

DESC and ASC syntax

The DESC sort keyword has the following basic syntax.

```
SELECT {fieldName(s) | *} FROM tableName(s) [WHERE condition] ORDER BY f
ieldname(s) ASC /DESC [LIMIT N]
```

HERE

- **SELECT {fieldName(s) | *} FROM tableName(s)** is the statement containing the fields and table(s) from which to get the result set from.
- **[WHERE condition]** is optional but can be used to filter the data according to the given condition.
- **ORDER BY** fieldname(s) is mandatory and is the field on which the sorting is to be performed. The DESC keyword specifies that the sorting is to be in descending order.
- **[LIMIT]** is optional but can be used to limit the number of results returned from the query result set.

Examples:

Let's now look at a practical example -

```
SELECT * FROM members;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results shown below.

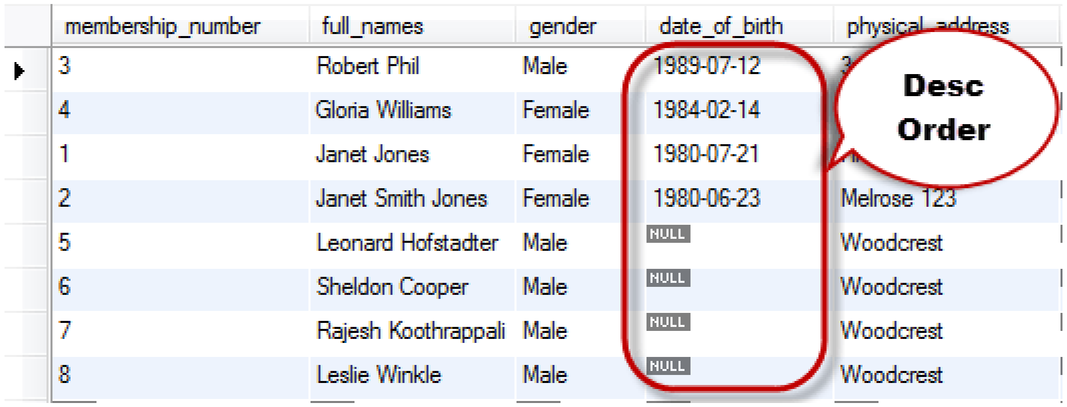
membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL
5	Leonar	Mal	NULL	Woodcrest	NULL	8457387	NULL

	d Hofstadter	e				67	
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	

Let's suppose the marketing department wants the members details arranged in decreasing order of Date of Birth. This will help them send birthday greetings in a timely fashion. We can get the said list by executing a query like below -

```
SELECT * FROM members ORDER BY date_of_birth DESC;
```

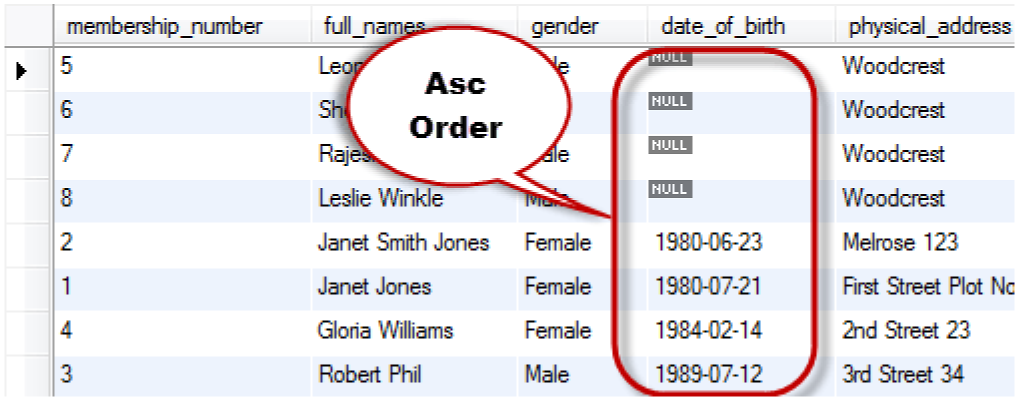
Executing the above script in MySQL workbench against the myflixdb gives us the following results shown below.



membership_number	full_names	gender	date_of_birth	physical_address
3	Robert Phil	Male	1989-07-12	3
4	Gloria Williams	Female	1984-02-14	[illegible]
1	Janet Jones	Female	1980-07-21	[illegible]
2	Janet Smith Jones	Female	1980-06-23	Melrose 123
5	Leonard Hofstadter	Male	NULL	Woodcrest
6	Sheldon Cooper	Male	NULL	Woodcrest
7	Rajesh Koothrappali	Male	NULL	Woodcrest
8	Leslie Winkle	Male	NULL	Woodcrest

The same query in ascending order

SELECT * FROM members ORDER BY date_of_birth ASC



membership_number	full_names	gender	date_of_birth	physical_address
5	Leon	le	NULL	Woodcrest
6	Sh	[illegible]	NULL	Woodcrest
7	Rajes	le	NULL	Woodcrest
8	Leslie Winkle	Ma	NULL	Woodcrest
2	Janet Smith Jones	Female	1980-06-23	Melrose 123
1	Janet Jones	Female	1980-07-21	First Street Plot No
4	Gloria Williams	Female	1984-02-14	2nd Street 23
3	Robert Phil	Male	1989-07-12	3rd Street 34

Note: NULL values means no values (not zero or empty string) . Observe the way they have been sorted.

More examples

Let's consider the following script that lists all the member records.

```
SELECT * FROM `members`;
```

Executing the above script gives the following results shown below.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contct_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL

5	Leonard Hofstadter	Male	NULL	Woodcrest	NULL	845738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	NULL

Suppose we want to get a list that sorts the query result set using the gender field, we would use the script shown below.

```
SELECT * FROM `members` ORDER BY `gender`;
```

membership	full_n	gen	date_of	physical_	postal_a	contct_n	email

_number	ames	der	_birth	address	ddress	umber	
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street 23	NULL	NULL	NULL
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
5	Leonard Hofstadter	Male	NULL	Woodcrest	NULL	845738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL

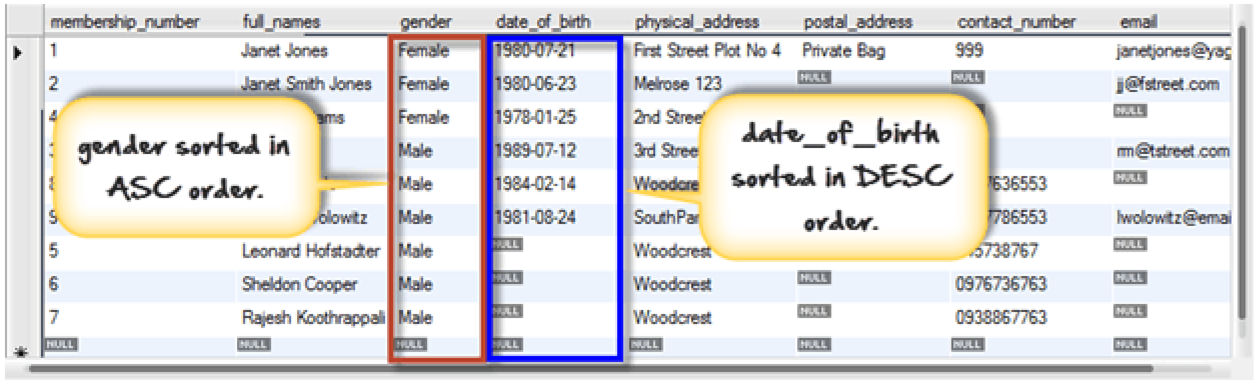
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	NULL

"Female" members have been displayed first followed by "Male" members, this is because when order by clause is used without specifying the ASC or DESC keyword, by default, MySQL has sorted the query result set in an ascending order.

Let's now look at an example that does the **sorting using two columns**; the first one is **sorted** in **ascending order** by default while the second column is **sorted** in **descending order.**

```
SELECT * FROM `members` ORDER BY `gender`,`date_of_birth` DESC;
```

Executing the above script in MySQL workbench against the myflixdb gives the following results.


gender sorted in ASC order.
date_of_birth sorted in DESC order.

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contact_number	email
1	Janet Jones	Female	1980-07-21	First Street Plot No 4	Private Bag	999	janetjones@yag
2	Janet Smith Jones	Female	1980-06-23	Melrose 123	NULL	NULL	jj@fstreet.com
4	ams	Female	1978-01-25	2nd Stree			NULL
3		Male	1989-07-12	3rd Stree			rm@tstreet.com
8		Male	1984-02-14	Woodcre		636553	NULL
9	olowitz	Male	1981-08-24	SouthPar		786553	lwolowitz@emai
5	Leonard Hofstadter	Male	NULL	Woodcrest		5738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	0976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	0938867763	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

The gender column was sorted in ascending order by default while the date of birth column was sorted in descending order explicitly

Why we may use DESC and ASC?

Suppose we want to print a payments history for a video library member to help answer queries from the front desk, wouldn't it be more logical to have the payments printed in a descending chronological order starting with the recent payment to the earlier payment?

The DESC key word comes in handy in such situations. We can write a query that sorts the list in descending order using the payment date.

Suppose the marketing department wants to get a list of movies by category that members can use to decide which movies are available in the library when renting movies, wouldn't it be more logical to look sort the movie category names and title in ascending so that members can quickly lookup the information from the list?

The ASC keyword comes in handy in such situations; we can get the movies list sorted by category name and movie title in an ascending order.

Summary

- Sorting query results is re-arranging the rows returned from a query result set either in ascending or descending order.
- The DESC keyword is used to sort the query result set in a descending order.
- The ASC keyword is used to sort the query result set in an ascending order.
- Both DESC and ASC work in conjunction with the ORDER BY keyword. They can also be used in combination with other keywords such as WHERE clause and LIMIT
- The default for ORDER BY when nothing has been explicitly specified is ASC.

MySQL GROUP BY and HAVING Clause with Examples

What is the Group by Clause?

The GROUP BY clause is a SQL command that is used to **group rows that have the same values**.

The GROUP BY clause is used in the SELECT statement .Optionally it is used in conjunction with aggregate functions to produce summary reports from the database.

That's what it does, **summarizing data** from the database.

The queries that contain the GROUP BY clause are called grouped queries and only return a single row for every grouped item.

GROUP BY Syntax

Now that we know what the GROUP By clause is, let's look at the syntax for a basic group by query.

```
SELECT statements... GROUP BY column_name1[,column_name2,...] [HAVING condition];
```

HERE

- "SELECT statements..." is the standard SQL SELECT command query.

- "**GROUP BY** *column_name1*" is the clause that performs the grouping based on column_name1.
- "[,column_name2,...]" is optional; represents other column names when the grouping is done on more than one column.
- "[HAVING condition]" is optional; it is used to restrict the rows affected by the GROUP BY clause. It is similar to the WHERE clause.

Grouping using a Single Column

In order to help understand the effect of Group By clause, let's execute a simple query that returns all the gender entries from the members table.

```
SELECT `gender` FROM `members` ;
```

gender
Female
Female
Male
Female
Male
Male
Male
Male
Male

Suppose we want to get the unique values for genders. We can use a following query -

```
SELECT `gender` FROM `members` GROUP BY `gender`;
```

Executing the above script in MySQL workbench against the Myflixdb gives us the following results.

gender
Female
Male

Note only two results have been returned. This is because we only have two gender types Male and Female. The GROUP BY clause grouped all the "Male" members together and returned only a single row for it. It did the same with the "Female" members.

Grouping using multiple columns

Suppose that we want to get a list of movie category_id and corresponding years in which they were released.

Let's observe the output of this simple query

```
SELECT `category_id`,`year_released` FROM `movies` ;
```

category_id	year_released
1	2011
2	2008
NULL	2008
NULL	2010
8	2007
6	2007
6	2007
8	2005
NULL	2012
7	1920
8	NULL
8	1920

The above result has many duplicates.

Let's execute the same query using group by -

```
SELECT `category_id`,`year_released` FROM `movies` GROUP BY `category_i
d`,`year_released`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results shown below.

category_id	year_released
NULL	2008
NULL	2010
NULL	2012
1	2011
2	2008
6	2007
7	1920
8	1920
8	2005
8	2007

The GROUP BY clause operates on both the category id and year released to identify **unique** rows in our above example.

If the category id is the same but the year released is different, then a row is treated as a unique one .If the category id and the year released is the same for more than one row, then it's considered a duplicate and only one row is shown.

Grouping and aggregate functions

Suppose we want total number of males and females in our database. We can use the following script shown below to do that.

```
SELECT `gender`,COUNT(`membership_number`)  FROM `members` GROUP
BY `gender`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

gender	COUNT('membership_number')
Female	3
Male	5

The results shown below are grouped by every unique gender value posted and the number of grouped rows is counted using the COUNT aggregate function.

Restricting query results using the HAVING clause

It's not always that we will want to perform groupings on all the data in a given table. There will be times when we will want to restrict our results to a certain given criteria. In such cases , we can use the HAVING clause

Suppose we want to know all the release years for movie category id 8. We would use the following script to achieve our results.

```
SELECT * FROM `movies` GROUP BY `category_id`,`year_released` HAVING `
category_id` = 8;
```

Executing the above script in MySQL workbench against the Myflixdb gives us the following results shown below.

movie_id	title	director	year_released	category_id
9	Honey mooners	John Schultz	2005	8
5	Daddy's Little Girls	NULL	2007	8

Note only movies with category id 8 have been affected by our GROUP BY clause.

Summary

- The GROUP BY Clause is used to group rows with same values .
- The GROUP BY Clause is used together with the SQL SELECT statement.
- The SELECT statement used in the GROUP BY clause can only be used contain column names, aggregate functions, constants and expressions.

- The HAVING clause is used to restrict the results returned by the GROUP BY clause.

Difference between WHERE vs HAVING clause in SQL - GROUP BY Comparison with Example

What is difference between WHERE and HAVING clause in SQL is one of the most popular question asked on SQL and database interviews, especially to beginners. Since programming jobs, required more than one skill, it’s quite common to see couple ofSQL Interview questions in Java and .NET interviews. By the way unlike any other question, not many Java programmers or dot net developers, who is supposed to have knowledge of basic SQL, fail to answer this question. Though almost half of the programmer says that WHERE is used in any SELECT query, while HAVING clause is only used in SELECT queries, which contains aggregate function or group by clause, which is correct. Though both WHERE and HAVING clause is used to specify filtering condition in SQL, there is subtle difference between them. Real twist comes into interview, when they are asked to explain result of a SELECT query, which contains both WHERE and HAVING clause, I have seen many people getting confused there.

Key point, which is also the main difference between WHERE and HAVING clause in SQL is that, condition specified in WHERE clause is used while fetching data (rows) from table, and data which doesn't pass the condition will not be fetched into result set, on the other hand HAVING clause is later used to filter summarized data or grouped data.

In short if both WHERE and HAVING clause is used in a SELECT query with aggregate function or GROUP BY clause, it will execute before HAVING clause. This will be more clear, when we will see an example of WHERE, HAVING, JOIN and GROUP BYclause together.

WHERE vs HAVING Clause Example in SQL

In this example of WHERE and HAVING clause, we have two tables Employee and Department. Employee contains details of employees e.g. id, name, age, salary and department id, while Department contains id and department name. In order to show, which employee works for which department we need to join two tables on DEPT_ID to get the the department name. Our requirement is to find how many employees are working in each department and average salary of department. In order to use WHERE clause, we will only include employees who are earning more than 5000. Before executing our query which contains WHERE, HAVING, and GROUP BY clause, let see data from Employee and Departmenttable:

SELECT * FROM Employee;

EMP_ID	EMP_NAME	EMP_AGE	EMP_SALARY	DEPT_ID
1	Virat	23	10000	1
2	Rohit	24	7000	2
3	Suresh	25	8000	3
4	Shikhar	27	6000	1
5	Vijay	28	5000	2

SELECT * FROM Department;

DEPT_ID	DEPT_NAME

1	Accounting
2	Marketing
3	Sales

```
SELECT d.DEPT_NAME, count(e.EMP_NAME) as NUM_EMPLOYEE, avg(e.EMP_SA
LARY) asAVG_SALARY FROM Employee e,

Department d WHERE e.DEPT_ID=d.DEPT_ID AND EMP_SALARY
> 5000 GROUP BY d.DEPT_NAME;
```

DEPT_NAME	NUM_EMPLOYEE	AVG_SALARY
Accounting	1	8000
Marketing	1	7000
Sales	2	8000

From the number of employee (NUM_EMPLOYEE) column you can see that only Vijay who work for Marketing department is not included in result set because his earning 5000. This example shows that, condition in WHERE clause is used to filter rows before you aggregate them and then HAVING clause comes in picture for final filtering, which is clear from following query, now Marketing department is excluded because it doesn't pass condition in HAVING clause i..e AVG_SALARY > 7000

```
SELECT d.DEPT_NAME, count(e.EMP_NAME) as NUM_EMPLOYEE, avg(e.EMP_SA
LARY) asAVG_SALARY FROM Employee e,

Department d WHERE e.DEPT_ID=d.DEPT_ID AND EMP_SALARY
> 5000 GROUP BY d.DEPT_NAMEHAVING AVG_SALARY > 7000;
```

DEPT_NAME	NUM_EMPLOYEE	AVG_SALARY
Accounting	1	8000
Sales	2	8000

Difference between WHERE and HAVING in SQL
Apart from this key difference we have seen in this article, here are few more differences between WHERE and HAVING clause, which is worth remembering and can be used to compare both of them :

1) Apart from SELECT queries, you can use WHERE clause
with UPDATE and DELETE clause but HAVING clause can only be used with SELECT query. For example following query, which involve WHERE clause will work but other which uses HAVINGclause will not work :

```
update DEPARTMENT set DEPT_NAME="NewSales" WHERE DEPT_ID=1 ;  //
works fine
```

```
update DEPARTMENT set DEPT_NAME="NewSales" HAVING DEPT_ID=1 ; //
error
```

```
Incorrect syntax near the
keyword 'HAVING'.: update DEPARTMENT setDEPT_NAME='NewSales' HAVING
DEPT_ID=1
```

2) WHERE clause is used for filtering rows and it applies on each and every row, while HAVING clause is used to filter groups in SQL.

3) One syntax level difference between WHERE and HAVING clause is that, former is used before GROUP BY clause, while later is used after GROUP BY clause.

4) When WHERE and HAVING clause are used together in a SELECT query with aggregate function, WHERE clause is applied first on individual rows and only rows which pass the condition is included for creating groups. Once group is created, HAVINGclause is used to filter groups based upon condition specified.

That's all on difference between WHERE and HAVING clause in SQL. As I said this is very popular question and you can't afford not to prepare it. Always remember key difference between WHERE and HAVING clause in SQL, if WHERE and HAVING clause is used together, first WHERE clause is applied to filter rows and only after grouping HAVING clause is applied.

MySQL Wildcards Tutorial: Like, NOT Like, Escape, (%), (_)

What are wildcards?

Wildcards are characters that help search data matching complex criteria. Wildcards are used in conjunction with the LIKE comparison operator or the NOT LIKE comparison operator.

Why use WildCards ?

If you are familiar with using the SQL, you may think that you can search for any complex data using SEELCT and WHERE clause . Then why use Wildcards ?

Before we answer that question, let's look at an example. Suppose that the marketing department of Myflix video library carried out marketing promotions in the city of Texas and would like to get some feedback on the number of members

that registered from Texas, you can use the following SELECT statement together with the WHERE clause to get the desired information.

```
SELECT * FROM members WHERE postal_address = 'Austin , TX' OR  postal_ad
dress = Dallas , TX OR postal_address = Iola,TX OR postal_adress = Houston ,
TX';
```

As you can see from the above query, the "WHERE clause" becomes complex. Using wildcards however, simplifies the query as we can use something simple like the script shown below.

```
SELECT * FROM  members  WHERE postal_address  like '% TX';
```

In short, wildcards allow us to develop power search engines into our data driven applications.

Types of wildcards

% the percentage

% the percentage character is used to specify a pattern of **zero (0) or more characters**. It has the following basic syntax.

```
SELECT statements... WHERE fieldname LIKE 'xxx%';
```

HERE

- "SELECT statement..." is the standard SQL SELECT command.
- "WHERE" is the key word used to apply the filter.
- "LIKE" is the comparison operator that is used in conjunction with wildcards
- 'xxx' is any specified starting pattern such as a single character or more and "%" matches any number of characters starting from zero (0).

To fully appreciate the above statement, let's look at a practical example

Suppose we want to get all the movies that have the word "code" as part of the title, we would use the percentage wildcard to perform a pattern match on both sides of the word "code". Below is the SQL statement that can be used to achieve the desired results.

```
SELECT * FROM movies WHERE title LIKE '%code%';
```

Executing the above script in MySQL workbench against the myflixdb gives us the results shown below.

movie_id	title	director	year_released	category_id
4	Code Name Black	Edgar Jimz	2010	NULL
7	Davinci Code	NULL	NULL	6

Notice that even if the search key word "code" appears on the beginning or end of the title, it is still returned in our result set. This is because our code includes any number of characters at the beginning then matches the pattern "code" followed by any number of characters at the end.

Let's now modify our above script to include the percentage wildcard at the beginning of the search criteria only.

```
SELECT * FROM movies WHERE title LIKE '%code';
```

Executing the above script in MySQL workbench against the myflixdb gives us the results shown below.

movie_id	title	director	year_released	category_id
7	Davinci Code	NULL	NULL	6

Notice that only one record has been returned from the database. This is because our code matches any number of characters at the beginning of the movie title and gets only records that end with the pattern "code".

Let's now shift the percentage wildcard to the end of the specified pattern to be matched. The modified script is shown below.

```
SELECT * FROM movies WHERE title LIKE 'code%';
```

Executing the above script in MySQL workbench against the myflixdb gives us the results shown below.

movie_id	title	director	year_released	category_id
4	Code Name Black	Edgar Jimz	2010	NULL

Notice only one record has been returned from the database. This is because our code matches all titles that start with the pattern "code" followed by any number of characters.

_ underscore wildcard

The underscore wildcard is used to **match exactly one character**. Let's suppose that we want to search for all the movies that were released in the years 200x where x is exactly one character that could be any value. We would use the underscore wild card to achieve that. The script below select all the movies that were released in the year "200x"

```
SELECT * FROM movies WHERE year_released LIKE '200_';
```

Executing the above script in MySQL workbench against the myflixdb gives us the results shown below.

movie_id	title	director	year_released	category_id
2	Forgetting Sarah Marshal	Nicholas Stoller	2008	2
9	Honey mooners	Jhon Shultz	2005	8

Notice that only movies that have 200 follows by any character in the field year released have been returned in our result set. This is because the underscore wildcard matched the pattern 200 followed by any single character

NOT Like

The NOT logical operator can be used together with the wildcards to return rows that do not match the specified pattern.

Suppose we want to get movies that were not released in the year 200x. We would use the NOT logical operator together with the underscore wildcard to get our results. Below is the script that does that.

```
SELECT * FROM movies WHERE year_released NOT LIKE '200_';
```

movie_id	title	director	year_released	category_id
1	Pirates of the Caribean 4	Rob Marshall	2011	1

4	Code Name Black	Edgar Jimz	2010	NULL
8	Underworld-Awakeninh	Michahel Eal	2012	6

Notice only movies that do not start with 200 in the year released have been returned in our result set. This is because we used the NOT logical operator in our wildcard pattern search.

Escape keyword.

The ESCAPE keyword is used to **escape pattern matching characters** such as the (%) percentage and underscore (_) if they form part of the data.

Let's suppose that we want to check for the string "67%" we can use;

```
LIKE '67#%%' ESCAPE '#';
```

If we want to search for the movie "67% Guilty", we can use the script shown below to do that.

```
SELECT * FROM movies WHERE title LIKE '67#%%' ESCAPE '#';
```

Note the double "**%**%" in the LIKE clause, the first one in red "**%**" is treated as part of the string to be searched for. The other one is used to match any number of characters that follow.

The same query will also work if we use something like

```
SELECT * FROM movies WHERE title LIKE '67=%%' ESCAPE '=';
```

Summary

- Like & Wildcards powerful tools that help search data matching complex patterns.
- There are a number of wildcards that include the percentage, underscore and charlist(not supported by MySQL) among others
- The percentage wildcard is used to match any number of characters starting from zero (0) and more.
- The underscore wildcard is used to match exactly one character.

MySQL Functions: String, Numeric, User-Defined, Stored

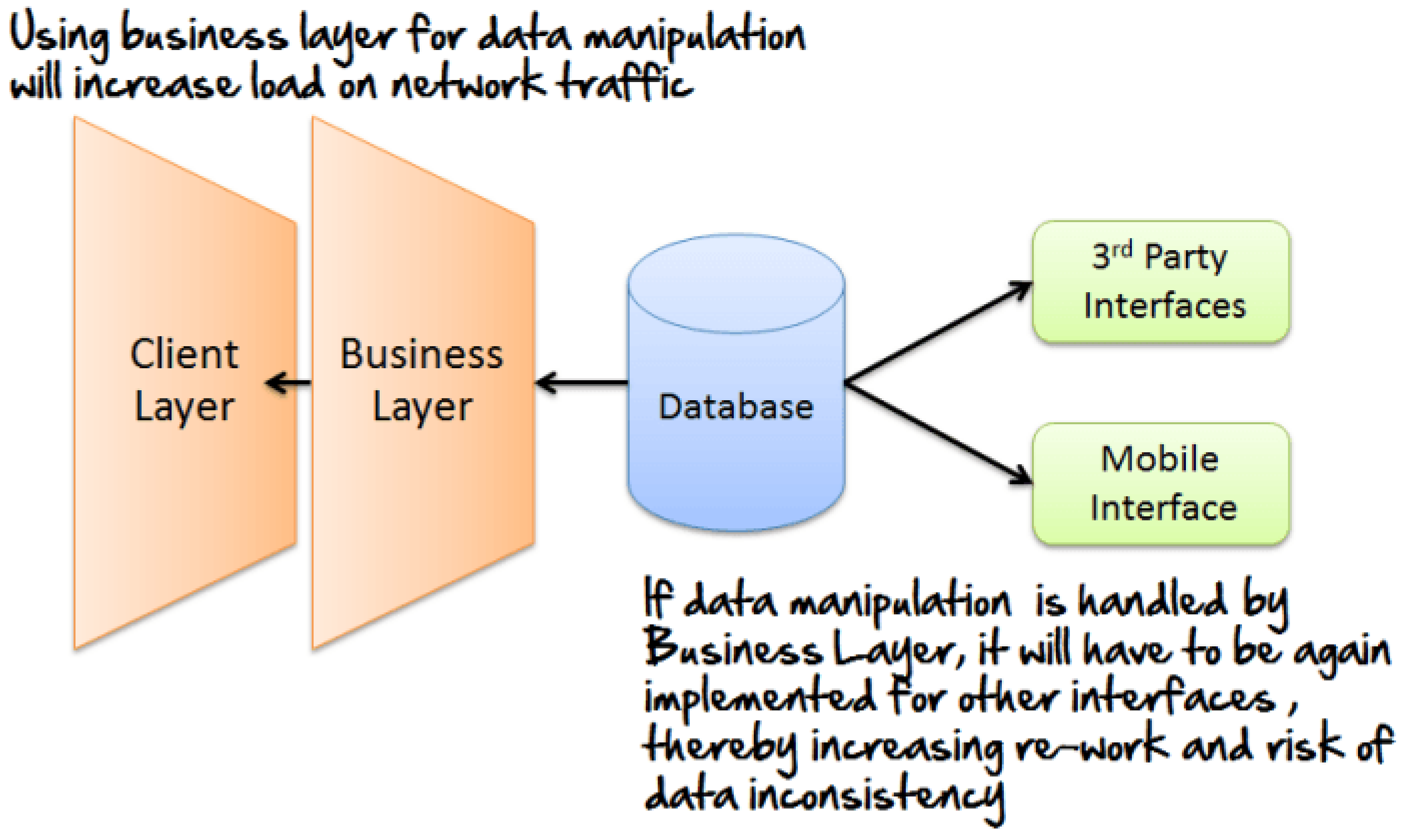
What are functions?

MySQL can do much more than just store and retrieve data. We can also **perform manipulations on the data** before retrieving or saving it. That's where MySQL Functions come in. Functions are simply pieces of code that perform some operations and then return a result. Some functions accept parameters while other functions do not accept parameters.

Let' briefly look at an example of MySQL function. By default, MySQL saves date data types in the format "YYYY-MM-DD". Suppose we have built an application and our users want the date to be returned in the format "DD-MM-YYYY", we can use MySQL built in function DATE_FORMAT to achieve this. DATE_FORMAT is one of the most used functions in MySQL. We will look at it in more details as we unfold the lesson.

Why use functions?

Why Use Functions?


Using business layer for data manipulation will increase load on network traffic
Client Layer
Business Layer
Database
3rd Party Interfaces
Mobile Interface
If data manipulation is handled by Business Layer, it will have to be again implemented for other interfaces, thereby increasing re-work and risk of data inconsistency

Based on the example given in the introduction, people with experience in computer programming may be thinking *"Why bother MySQL Functions? The same effect can be achieved with scripting/programming language?"* It's true we can achieve that by writing some procedures/function in the application program.

Getting back to our DATE example in the introduction, for our users to get the data in the desired format, business layer will have to do necessary processing.

This becomes a problem when the application has to integrate with other systems. When we use MySQL functions such as the DATE_FORMAT, then we can have that functionality embedded into the database and any application that needs the data gets it in the required format. This **reduces re-work in the business logic and reduce data inconsistencies.**

Another reason why we should consider using **MySQL functions is the fact that it can help reducing network traffic in client/server applications**. Business Layer will only need to make call to the stored functions without the need manipulate data .On average, the use of functions can help greatly improve overall system performance.

Types of functions

Built-in functions

MySQL comes bundled with a number of built in functions. Built in functions are simply functions come already implemented in the MySQL server. These functions allow us to perform different types of manipulations on the data. The built in functions can be basically categorized into the following most used categories.

- **Strings functions** - operate on string data types
- **Numeric functions** - operate on numeric data types
- **Date functions** - operate on date data types
- **Aggregate functions** - operate on all of the above data types and produce summarized result sets.
- **Other functions** - MySQL also supports other types of built in functions but we will limit our lesson to the above named functions only.

Let's now look at each of the functions mentioned above in detail. We will be explaining the most used functions using our "Myflixdb".

String functions

We already looked at what string functions do. We will look at a practical example that uses them. In our movies table, the movie titles are stored using combinations of lower and upper case letters. Suppose we want to get a query list that returns the movie titles in upper case letters. We can use the "UCASE" function to do that. It takes a string as a parameter and converts all the letters to upper case. The script shown below demonstrates the use of the "UCASE" function.

```
SELECT `movie_id`,`title`, UCASE(`title`)  FROM `movies`;
```

HERE

- UCASE(`title`) is the built in function that takes the title as a parameter and returns it in upper case letters with the alias name `upper_case_title`.

Executing the above script in MySQL workbench against the Myflixdb gives us the following results shown below.

movie_id	title	UCASE('title')
16	67% Guilty	67% GUILTY
6	Angels and Demons	ANGELS AND DEMONS
4	Code Name Black	CODE NAME BLACK
5	Daddy's Little Girls	DADDY'S LITTLE GIRLS
7	Davinci Code	DAVINCI CODE
2	Forgetting Sarah Marshal	FORGETTING SARAH MARSHAL
9	Honey mooners	HONEY MOONERS
19	movie 3	MOVIE 3
1	Pirates of the Caribean 4	PIRATES OF THE CARIBEAN 4
18	sample movie	SAMPLE MOVIE
17	The Great Dictator	THE GREAT DICTATOR
3	X-Men	X-MEN

MySQL supports a number of string functions. For a complete list of all the built in string functions, refere to this link http://dev.mysql.com/doc/refman/5.0/en/string-functions.html on MySQL website.

Numeric functions

As earlier mentioned, these functions operate on numeric data types. We can perform mathematic computations on numeric data in the SQL statements.

Arithematic operators

MySQL supports the following arithmatic operators that can be used to perform computations in the SQL statements.

Name	Description
DIV	Integer division
/	Division
-	Subtraction
+	Addition
*	Multiplication

% or MOD	Modulus

Let's now look at examples of each of the above operator

Integer Division (DIV)

```
SELECT 23 DIV 6 ;
```

Executing the above script gives us the following results.

```
3
```

Division operator (/)

Let's now look at the division operator example. We will modify the DIV example.

```
SELECT 23 / 6 ;
```

Executing the above script gives us the following results.

```
3.8333
```

Subtraction operator (-)

Let's now look at the subtraction operator example. We will use the same values as in the previous two examples

```
SELECT 23 - 6 ;
```

Executing the above script gives us 17

Addition operator (+)

Let's now look at the addition operator example. We will modify the previous example.

```
SELECT 23 + 6 ;
```

Executing the above script gives us 29

Multiplication operator (*)

Let's now look at the multiplication operator example. We will use the same values as in the previous examples.

```
SELECT 23 * 6 AS `multiplication_result`;
```

Executing the above script gives us the following results.

multiplication_result
138

Modulo operator (-)

The modulo operator divides N by M and gives us the reminder. Let's now look at the modulo operator example. We will use the same values as in the previous examples.

```
SELECT 23 % 6 ;
```

OR

```
SELECT 23 MOD 6 ;
```

Executing the above script gives us 5

Let's now look at some of the common numeric functions in MySQL.

Floor - this function removes decimals places from a number and rounds it to the nearest lowest number. The script shown below demonstrates its usage.

```
SELECT FLOOR(23 / 6) AS `floor_result`;
```

Executing the above script gives us the following results.

Floor_result
3

Round - this function rounds a number with decimal places to the nearest whole number. The script shown below demonstrates its usage.

```
SELECT ROUND(23 / 6) AS `round_result`;
```

Executing the above script gives us the following results.

Round_result
4

Rand - this function is used to generate a random number, its value changes every time that the function is called. The script shown below demonstrates its usage.

```
SELECT RAND() AS `random_result`;
```

Stored functions

Stored functions are just like built in functions except that you have to define the stored function yourself. Once a stored function has been created, it can be used in SQL statements just like any other function. The basic syntax for creating a stored function is as shown below

```
CREATE FUNCTION sf_name ([parameter(s)])
```

```
RETURNS data type
DETERMINISTIC
STATEMENTS
```

HERE

- **"CREATE FUNCTION sf_name ([parameter(s)]) "** is mandatory and tells MySQL server to create a function named `sf_name' with optional parameters defined in the parenthesis.
- **"RETURNS data type"** is mandatory and specifies the data type that the function should return.
- **"DETERMINISTIC"** means the function will return the same values if the same arguments are supplied to it.
- **"STATEMENTS"** is the procedural code that the function executes.

Let's now look at a practical example that implements a built in function. Suppose we want to know which rented movies are past the return date. We can create a stored function that accepts the return date as the parameter and then compares it with the current date in MySQL server. If the current date is less than the return movie date, then we return "No" else we return "Yes". The script shown below helps us to achieve that.

```
DELIMITER |
CREATE FUNCTION sf_past_movie_return_date (return_date DATE)
  RETURNS VARCHAR(3)
   DETERMINISTIC
    BEGIN
     DECLARE sf_value VARCHAR(3);
        IF curdate() > return_date
            THEN SET sf_value = 'Yes';
        ELSEIF  curdate() <= return_date
            THEN SET sf_value = 'No';
        END IF;
     RETURN sf_value;
    END|
```

Executing the above script created the stored function `` `sf_past_movie_return_date` ``.

Let's now test our stored function.

```
SELECT `movie_id`,`membership_number`,`return_date`,CURDATE() ,sf_past
_movie_return_date(`return_date`)  FROM `movierentals`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

movie_id	membership_number	return_date	CURDATE()	sf_past_movie_return_date('return_date')
1	1	NULL	04-08-2012	NULL
2	1	25-06-2012	04-08-2012	yes
2	3	25-06-2012	04-08-2012	yes
2	2	25-06-2012	04-08-2012	yes
3	3	NULL	04-08-2012	NULL

User-defined functions

MySQL also supports user defined functions that extend MySQL. User defined functions are functions that you can create using a programming language such as C, C++ etc. and then add them to MySQL server. Once added, they can be used just like any other function.

Summary

- Functions allow us to enhance the capabilities of MySQL.
- Functions always return a value and can optionally accept parameters.
- Built in functions are functions that are shipped with MySQL. They can be categorized according to the data types that they operate on i.e. strings, date and numeric built in functions.
- Stored functions are created by the user within MySQL server and can be used in SQL statements.
- User defined functions are created outside MySQL and can be incorporated into MySQL server.

MySQL Aggregate Functions Tutorial : SUM, AVG, MAX, MIN , COUNT, DISTINCT

Aggregate Functions are all about

- Performing calculations on multiple rows
- Of a single column of a table
- And returning a single value.

The ISO standard defines five (5) aggregate functions namely;

1) COUNT
2) SUM
3) AVG
4) MIN
5) MAX

Why use aggregate functions.

From a business perspective, different organization levels have different information requirements. Top levels managers are usually interested in knowing whole figures and not necessary the individual details.

>Aggregate functions allow us to easily produce summarized data from our database.

For instance, from our myflix database , management may require following reports

- Least rented movies.
- Most rented movies.

- Average number that each movie is rented out in a month.

We easily produce above reports using aggregate functions.

Let's look into aggregate functions in detail.

COUNT Function

The COUNT function returns the total number of values in the specified field. It works on both numeric and non-numeric data types. **All aggregate functions by default exclude nulls values before working on the data.**

COUNT (*) is a special implementation of the COUNT function that returns the count of all the rows in a specified table. COUNT (*) also considers Nulls and duplicates.

The table shown below shows data in movierentals table

reference_ number	transaction_ date	return_date	membership_ number	movie_id	movie_returned
11	20-06-2012	NULL	1	1	0

12	22-06-2012	25-06-2012	1	2	0
13	22-06-2012	25-06-2012	3	2	0
14	21-06-2012	24-06-2012	2	2	0
15	23-06-2012	NULL	3	3	0

Let's suppose that we want to get the number of times that the movie with id 2 has been rented out

```
SELECT COUNT(`movie_id`)  FROM `movierentals` WHERE `movie_id` = 2;
```

Executing the above query in MySQL workbench against myflixdb gives us the following results.

```
COUNT('movie_id')
3
```

DISTINCT Keyword

The DISTINCT keyword that allows us to omit duplicates from our results. This is achieved by grouping similar values together .

To appreciate the concept of Distinct, lets execute a simple query

```
SELECT `movie_id` FROM `movierentals`;
```

```
movie_id
1
2
2
2
3
```

Now let's execute the same query with the distinct keyword -

```
SELECT DISTINCT `movie_id` FROM `movierentals`;
```

As shown below , distinct omits duplicate records from the results.

movie_id
1
2
3

MIN function

The MIN function **returns the smallest value in the specified table field**.

As an example, let's suppose we want to know the year in which the oldest movie in our library was released, we can use MySQL's MIN function to get the desired information.

The following query helps us achieve that

```
SELECT MIN(`year_released`) FROM `movies`;
```

Executing the above query in MySQL workbench against myflixdb gives us the following results.

```
MIN('year_released')
2005
```

MAX function

Just as the name suggests, the MAX function is the opposite of the MIN function. It **returns the largest value from the specified table field**.

Let's assume we want to get the year that the latest movie in our database was released. We can easily use the MAX function to achieve that.

The following example returns the latest movie year released.

```
SELECT MAX(`year_released`)  FROM `movies`;
```

Executing the above query in MySQL workbench using myflixdb gives us the following results.

```
MAX('year_released')
2012
```

SUM function

Suppose we want a report that gives total amount of payments made so far. We can use the MySQL **SUM** function which **returns the sum of all the values in the specified column**. **SUM works on numeric fields only**. **Null values are excluded from the result returned.**

The following table shows the data in payments table-

payment_id	**membership_number**	**payment_date**	**description**	**amount_paid**	**external_reference_number**
1	1	23-07-2012	Movie rental payment	2500	11
2	1	25-07-2012	Movie rental payment	2000	12
3	3	30-07-2012	Movie rental payment	6000	NULL

The query shown below gets the all payments made and sums them up to return a single result.

```
SELECT SUM(`amount_paid`) FROM `payments`;
```

Executing the above query in MySQL workbench against the myflixdb gives the following results.

```
SUM('amount_paid')
10500
```

AVG function

MySQL AVG function **returns the average of the values in a specified column**. Just like the SUM function, it **works only on numeric data types**.

Suppose we want to find the average amount paid. We can use the following query -

```
SELECT AVG(`amount_paid`)  FROM `payments`;
```

Executing the above query in MySQL workbench, gives us the following results.

```
AVG('amount_paid')
3500
```

Summary

- MySQL supports all the five (5) ISO standard aggregate functions COUNT, SUM, AVG, MIN and MAX.
- SUM and AVG functions only work on numeric data.
- If you want to exclude duplicate values from the aggregate function results, use the DISTINCT keyword. The ALL keyword includes even duplicates. If nothing is specified the ALL is assumed as the default.
- Aggregate functions can be used in conjunction with other SQL clauses such as GROUP BY

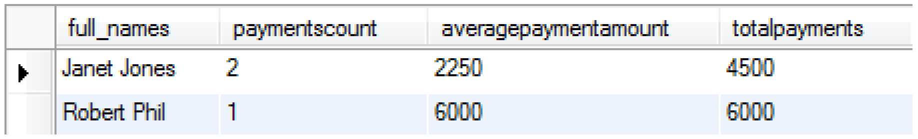
Brain Teaser

You think aggregate functions are easy. Try this!

The following example groups members by name, counts the total number of payments, the average payment amount and the grand total of the payment amounts.

```
SELECT m.`full_names`,COUNT(p.`payment_id`) AS  `paymentscount`,AVG(p
.`amount_paid`)  AS `averagepaymentamount`,SUM(p.`amount_paid`)  AS `
totalpayments` FROM members m, payments p WHERE m.`membership_numb
er` = p.`membership_number` GROUP BY m.`full_names`;
```

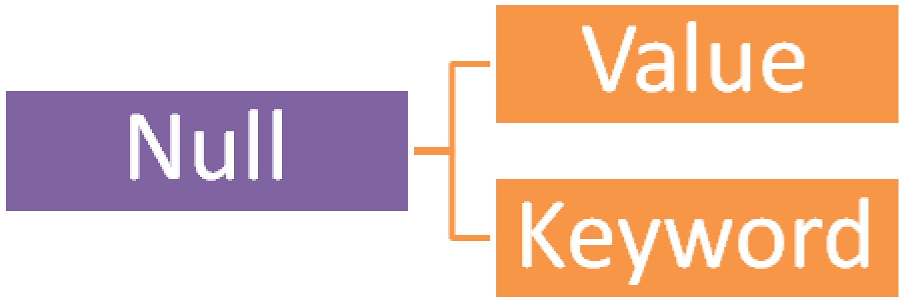
Executing the above example in MySQL workbench gives us the following results.



full_names	paymentscount	averagepaymentamount	totalpayments
Janet Jones	2	2250	4500
Robert Phil	1	6000	6000

MySQL IS NULL & IS NOT NULL Tutorial with Examples

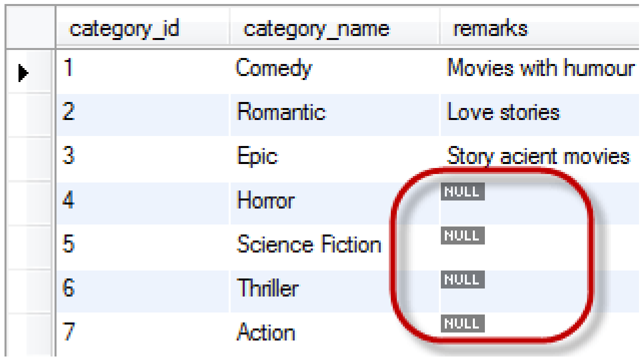
In SQL Null is both a value as well as a keyword. Let's look into NULL value first -



Null as a Value

In simple terms, NULL is simply a place holder for data that does not exist. When performing insert operations on tables, they will be times when some field values will not be available.

In order to meet the requirements of true relational database management systems, MySQL uses NULL as the place holder for the values that have not been submitted. The screenshot below shows how NULL values look in database.



category_id	category_name	remarks
1	Comedy	Movies with humour
2	Romantic	Love stories
3	Epic	Story acient movies
4	Horror	NULL
5	Science Fiction	NULL
6	Thriller	NULL
7	Action	NULL

Let's now look at some of the basics for NULL before we go further into the discussion.

- **NULL is not a data type** - this means it is not recognized as an "int", "date" or any other defined data type.
- **Arithmetic operations** involving**NULL** always **return NULL** for example, 69 + NULL = NULL.
- All **aggregate functions affect only rows that do not have NULL values**.

Let's now demonstrate how the count function treats null values. Let's see the current contents of the members table-

```
SELECT * FROM `members`;
```

Executing the above script gives us the following results

membership_number	full_names	gender	date_of_birth	physical_address	postal_address	contact_number	email
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm
2	Janet Smith Jones	Female	23-06-1980	Melrose 123	NULL	NULL	jj@fstreet.com
3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
4	Gloria Williams	Female	14-02-1984	2nd Street	NULL	NULL	NULL

				23			
5	Leonard Hofstadter	Male	NULL	Woodcrest	NULL	845738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	lwolowitz[at]email.me

Let's count all members who have updated their contact_number

```
SELECT COUNT(contact_number)  FROM `members`;
```

Executing the above query gives us the following results.

COUNT(contact_number)
7

Note: Values that are NULL have not been included

What is NOT?

The NOT logical operator is used to test for Boolean conditions and returns true if the condition is false. The NOT operator returns false if the condition been tested is true

Condition	NOT Operator Result
True	False
False	True

Why use NOT null?

There will be cases when we will have to perform computations on a query result set and return the values. Performing any arithmetic operations on columns that have the NULL value returns null results. In order to avoid such situations from happening, we can employ the use of the NOT NULL clause to limit the results on which our data operates.

NOT NULL Values

Let's suppose that we want to create a table with certain fields that should always be supplied with values when inserting new rows in a table. We can use the NOT NULL clause on a given field when creating the table.

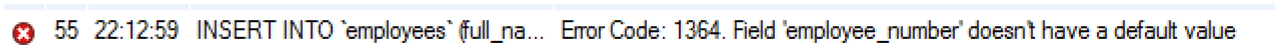
The example shown below creates a new table that contains employee's data. The employee number should always be supplied

```
CREATE TABLE `employees`(
  employee_number int NOT NULL,
  full_names varchar(255) ,
  gender varchar(6)
);
```

Let's now try to insert a new record without specifying the employee name and see what happens.

```
INSERT INTO `employees` (full_names,gender) VALUES ('Steve Jobs', 'Male');
```

Executing the above script in MySQL workbench gives the following error -



```
55  22:12:59  INSERT INTO `employees` (full_na...  Error Code: 1364. Field 'employee_number' doesn't have a default value
```

NULL Keywords

NULL can also be used as a keyword when performing Boolean operations on values that include NULL. The "IS/NOT" keyword is used in conjunction with the NULL word for such purposes. The basic syntax when null is used as a keyword is as follows

```
`comlumn_name'  IS NULL
`comlumn_name' NOT NULL
```

HERE

- ***"IS NULL"*** is the keyword that performs the Boolean comparison. It returns true if the supplied value is NULL and false if the supplied value is not NULL.
- ***"NOT NULL"*** is the keyword that performs the Boolean comparison. It returns true if the supplied value is not NULL and false if the supplied value is null.

Let's now look at a practical example that uses the NOT NULL keyword to eliminate all the column values that have null values.

Continuing with the example above , suppose we need details of members whose contact number is not null . We can execute a query like

```
SELECT * FROM `members` WHERE contact_number IS NOT NULL;
```

Executing the above query gives only records where contact number is not null.

Suppose we want member records where contact number is null. We can use following query

```
SELECT * FROM `members` WHERE contact_number IS NOT NULL;
```

Executing the above query gives member details whose contact number is NULL

membership_number	**full_names**	**gender**	**date_of_birth**	**physical_address**	**postal_address**	**contact_number**	**email**
1	Janet Jones	Female	21-07-1980	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm

3	Robert Phil	Male	12-07-1989	3rd Street 34	NULL	12345	rm@tstreet.com
5	Leonard Hofstadter	Male	NULL	Woodcrest	NULL	845738767	NULL
6	Sheldon Cooper	Male	NULL	Woodcrest	NULL	976736763	NULL
7	Rajesh Koothrappali	Male	NULL	Woodcrest	NULL	938867763	NULL
8	Leslie Winkle	Male	14-02-1984	Woodcrest	NULL	987636553	NULL
9	Howard Wolowitz	Male	24-08-1981	SouthPark	P.O. Box 4563	987786553	lwolowitz[at]email.me

Comparing null values

Three-value logic - performing Boolean operations on conditions that involve NULL can either return **"Unknown", "True" or "False".**

For example, **using the "IS NULL" keyword** when doing comparison operations **involving NULL** can either return **true** or **false**. Using other comparison operators returns **"Unknown"(NULL).**

Suppose you compare number five with 5

```
SELECT 5 =5;
```

The query result is 1 which means TRUE

5 =5
1

Let's do the same operation with NULL

```
SELECT NULL = NULL;
```

NULL = NULL

NULL

Let's look at another example

```
SELECT 5 > 5;
```

5 > 5
0

The query result is 0 which means FALSE

Let's look at same example using NULL

```
SELECT NULL > NULL;
```

NULL > NULL
NULL

Lets use the IS NULL keyword

```
SELECT 5 IS NULL;
```

5 IS NULL
0

The query result is 0 which is FALSE

```
SELECT NULL IS NULL;
```

NULL IS NULL
1

The query result is 1 which is TRUE

Summary

- NULL is a value place holder for optional table fields.
- MySQL treats the NULL value differently from other data types. The NULL values when used in a condition evaluates to the false Boolean value.
- The NOT logical operate is used to test for Boolean values and evaluates to true if the Boolean value is false and false if the Boolean value is true.
- The NOT NULL clause is used to eliminate NULL values from a result set
- Performing arithmetic operations on NULL values always returns NULL results.

- The comparison operators such as [, =, etc.] cannot be used to compare NULL values.

MySQL AUTO_INCREMENT with Examples

What is auto increment?

Auto Increment is a function that operates on numeric data types. It automatically generates sequential numeric values every time that a record is inserted into a table for a field defined as auto increment.

When use auto increment?

In the lesson on database normalization, we looked at how data can be stored with minimal redundancy, by storing data into many small tables ,related to each other using primary and foreign keys.

A primary key must be unique as it uniquely identifies a row in a database. But, how can we ensure that the primary key is always unique? One of the possible solutions would be, to use a formula to generate the primary key, which checks for existence of the key, in the table, before adding data. This may work well but as you can see the approach is complex and not foolproof. In order to avoid such complexity and to ensure that the primary key is always unique, we can use MySQL's Auto increment feature to generate primary keys. Auto increment is used with the INT data type. The INT data type supports both signed and unsigned values. Unsigned data types can only contain positive numbers. As a best practice, it is recommended to define the unsigned constraint on the auto increment primary key.

Auto increment syntax

Let's now look at the script used to create the movie categories table.

```
CREATE TABLE `categories` (
  `category_id` int(11) AUTO_INCREMENT,
  `category_name` varchar(150) DEFAULT NULL,
  `remarks` varchar(500) DEFAULT NULL,
  PRIMARY KEY (`category_id`)
);
```

Notice the "AUTO_INCREMENT" on the category_id field. This causes the category Id to be automatically generated every time a new row is inserted into the table. It is not supplied when inserting data into the table, MySQL generates it.

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

Let's examine the current contents of the categories table.

```
SELECT * FROM `categories`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

category_id	category_name	remarks
1	Comedy	Movies with humour
2	Romantic	Love stories
3	Epic	Story acient movies
4	Horror	NULL
5	Science Fiction	NULL
6	Thriller	NULL
7	Action	NULL
8	Romantic Comedy	NULL

Let's now insert a new category into the categories table .

```
INSERT INTO  `categories` (`category_name`) VALUES ('Cartoons');
```

Executing the above script against the myflixdb in MySQL workbench gives us the following results shown below.

category_id	category_name	remarks
1	Comedy	Movies with humour
2	Romantic	Love stories
3	Epic	Story acient movies
4	Horror	NULL
5	Science Fiction	NULL

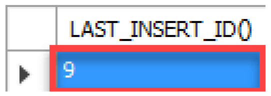
6	Thriller	NULL
7	Action	NULL
8	Romantic Comedy	NULL
9	Cartoons	NULL

Note we didn't supply the category id. MySQL automatically generated it for us because the category id is defined as auto increment.

If you want to get the last insert id that was generated by MySQL, you can use the LAST_INSERT_ID function to do that. The script shown below gets the last id that was generated.

```
SELECT LAST_INSERT_ID();
```

Executing the above script gives the last Auto increment number generated by the INSERT query. The results are shown below.



```
LAST_INSERT_ID()
9
```

Summary

- Auto increment attribute when specified on a column with a numeric data types, generates numbers sequentially whenever a new row is added into the database.
- The Auto increment is commonly used to generate primary keys.

- The defined data type on the Auto increment should be large enough to accommodate many records. Defining TINYINT as the data type for an auto increment field limits the number of records that can be added to the table to 255 only since any values beyond that would not be accepted by the TINYINT data type.
- It is considered a good practice to specify the unsigned constraint on auto increment primary keys to avoid having negative numbers.
- When a row is deleted from a table, its auto incremented id is not re-used. MySQL continues generating new numbers sequentially.
- By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record
- To let AUTO_INCREMENT sequence start with another value , use AUTO_INCREMENT = 10

MYSQL - ALTER, DROP, RENAME, MODIFY

WHAT IS THE ALTER COMMAND?

As the saying goes **Change is the only constant**

With time business requirements change as well. As business requirements change, Database designs need changing as well.

MySQL provides the **ALTER** function that helps us **incorporate the changes to the already existing database design**.

The alter command is used to modify an existing database, table, view or other database objects that might need to change during the life cycle of a database.

Let's suppose that we have completed our database design and it has been implemented. Our database users are using it and then they realize some of the vital information was left out in the design phase. They don't want to lose the existing data but just want to incorporate the new information. The alter command comes in handy in such situations. We can use the alter command to change the data type of a field from say string to numeric, change the field name to a new name or even add a new column in a table.

Alter- syntax

The basic syntax used to add a column to an already existing table is shown below

```
ALTER TABLE `table_name` ADD COLUMN `column_name` `data_type`;
```

HERE

- **"ALTER TABLE `table_name`"** is the command that tells MySQL server to modify the table named `table_name`.
- **"ADD COLUMN `column_name` `data_type`"** is the command that tells MySQL server to add a new column named `column_name` with data type `data_type'.

Let's suppose that Myflix has introduced online billing and payments. Towards that end, we have been asked to add a field for the credit card number in our members table. We can use the ALTER command to do that. Let's first look at the structure of the members table before we make any amendments. The script shown below helps us to do that.

```
SHOW COLUMNS FROM `members`;
```

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
membership_number	int(11)	NO	PRI	NULL	auto_increment
full_names	varchar(350)	NO		NULL	
gender	varchar(6)	YES		NULL	
date_of_birth	date	YES		NULL	
physical_address	varchar(255)	YES		NULL	

postal_address	varchar(255)	YES		NULL	
contact_number	varchar(75)	YES		NULL	
email	varchar(255)	YES		NULL	

We can use the script shown below to add a new field to the members table.

```
ALTER TABLE `members` ADD COLUMN `credit_card_number` VARCHAR(25);
```

Executing the above script in MySQL against the Myflixdb adds a new column named credit card number to the members table with VARCHAR as the data type. Executing the show columns script gives us the following results.

```
SHOW COLUMNS FROM `members`;
```

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
membership_number	int(11)	NO	PRI	NULL	auto_increment

```
full_names          varchar(350) NO          NULL
gender              varchar(6)   YES         NULL
date_of_birth       date         YES         NULL
physical_address    varchar(255) YES         NULL
postal_address      varchar(255) YES         NULL
contact_number      varchar(75)  YES         NULL
email               varchar(255) YES         NULL
credit_card_number  varchar(25)  YES
```

As you can see from the results returned, credit card number has been added to the members table. The data contained in the members' data is not affected by the addition of the new column.

WHAT IS THE DROP COMMAND?

The DROP command is used to

1. Delete a database from MySQL server
2. Delete an object (like Table , Column)from a database.

Let's now look at practical examples that make use of the DROP command.

In our previous example on the Alter Command, we added a column named credit card number to the members table.

Suppose the online billing functionality will take some time and we want to DROP the credit card column

We can use the following script

```
ALTER TABLE `members` DROP COLUMN `credit_card_number`;
```

Executing the above script drops the column credit_card_number from the members table

Let's now look at the columns in the members table to confirm if our column has been dropped.

```
SHOW COLUMNS FROM `members`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

Field	Type	Null	Key	Default	Extra
membership_number	int(11)	NO	PRI	NULL	auto_increment
full_names	varchar(350)	NO		NULL	

```
gender              varchar(6)   YES      NULL
date_of_birth       date         YES      NULL
physical_address    varchar(255) YES      NULL
postal_address      varchar(255) YES      NULL
contact_number      varchar(75)  YES      NULL
email               varchar(255) YES      NULL
```

Notice that the credit card number has been dropped from the fields list.

DROP TABLE

The syntax to DROP a table from Database is as follow -

```
DROP TABLE `sample_table`;
```

Let'look at an example

```
DROP TABLE `categories_archive`;
```

Executing the above script deletes the table named ` categories_archive ` from our database.

WHAT IS THE RENAME COMMAND?

The rename command is used to **change the name of an existing database object(like Table,Column) to a new name**.

Renaming a table does not make it to lose any data is contained within it.

Syntax:-

The rename command has the following basic syntax.

```
RENAME TABLE `current_table_name` TO `new_table_name`;
```

Let's suppose that we want to rename the movierentals table to movie_rentals, we can use the script shown below to achieve that.

```
RENAME TABLE `movierentals` TO `movie_rentals`;
```

Executing the above script renames the table `movierentals` to `movie_rentals`.

We will now rename the movie_rentals table back to its original name.

```
RENAME TABLE `movie_rentals` TO `movierentals`;
```

CHANGE KEYWORD

Change Keywords allows you to

1. Change Name of Column
2. Change Column Data Type
3. Change Column Constraints

Let's look at an example. The full names field in the members table is of varchar data type and has a width of 150.

```
SHOW COLUMNS FROM `members`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

Field	Type	Null	Key	Default	Extra
membership_number	int(11)	NO	PRI	NULL	auto_increment
full_names	varchar(150)	NO		NULL	

```
gender              varchar(6)    YES    NULL
date_of_birth       date          YES    NULL
physical_address    varchar(255)  YES    NULL
postal_address      varchar(255)  YES    NULL
contact_number      varchar(75)   YES    NULL
email               varchar(255)  YES    NULL
```

Suppose we want to

1. Change the field name from "full_names" to "fullname
2. Change it to char data type with a width of 250
3. Add a NOT NULL constraint

We can accomplish this using the change command as follows -

```
ALTER TABLE `members` CHANGE COLUMN `full_names` `fullname` char(250
) NOT NULL;
```

Executing the above script in MySQL workbench against myflixdb and then executing the show columns script given above gives the following results.

Field	Type	Null	Key	Default	Extra
membership_number	int(11)	NO	PRI	NULL	auto_increment
fullnames	char(250)	NO		NULL	
gender	varchar(6)	YES		NULL	
date_of_birth	date	YES		NULL	
physical_address	varchar(255)	YES		NULL	
postal_address	varchar(255)	YES		NULL	
contact_number	varchar(75)	YES		NULL	
email	varchar(255)	YES		NULL	

MODIFY KEYWORD

The MODIFY Keyword allows you to

1. Modify Column Data Type
2. Modify Column Constraints

In the CHANGE example above, we had to change the field name as well other details. **Omitting the field name from the CHANGE statement will generate an error.** Suppose we are only interested in changing the data type and constraints on the field without affecting the field name, we can use the MODIFY keyword to accomplish that.

The script below changes the width of "fullname" field from 250 to 50.

```
ALTER TABLE `members`MODIFY `fullname` char(50) NOT NULL;
```

Executing the above script in MySQL workbench against myflixdb and then executing the show columns script given above gives the following results shown below.

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
membership_number	int(11)	NO	PRI	NULL	auto_increment
fullnames	char(50)	NO		NULL	
gender	varchar(6)	YES		NULL	
date_of_birth	date	YES		NULL	
physical_address	varchar(255)	YES		NULL	

postal_address	varchar(255)	YES		NULL	
contact_number	varchar(75)	YES		NULL	
email	varchar(255)	YES		NULL	

AFTER KEYWORD

Suppose that we want to add a new column at a specific position in the table.

We can use the alter command together with the AFTER keyword.

The script below adds "date_of_registration" just after the date of birth in the members table.

```
ALTER TABLE `members` ADD `date_of_registration` date NULL AFTER `dat
e_of_birth`;
```

Executing the above script in MySQL workbench against myflixdb and then executing the show columns script given above gives the following results shown below.

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
membership_number	int(11)	NO	PRI	NULL	auto_increment

fullnames	char(50)	NO	NULL
gender	varchar(6)	YES	NULL
date_of_birth	date	YES	NULL
date_of_registration	date	YES	NULL
physical_address	varchar(255)	YES	NULL
postal_address	varchar(255)	YES	NULL
contact_number	varchar(75)	YES	NULL
email	varchar(255)	YES	NULL

Note: The Hilighted row is **added after date_of_birth** cloumn

Summary

- The alter command is used when we want to modify a database or any object contained in the database.
- The drop command is used to delete databases from MySQL server or objects within a database.
- The rename command is used to change the name of a table to a new table name.

- The Change keyword allows you to change a column name , data type and constraints
- The Modify Keyword allows you to modify a column data type and constraints
- The After keyword is used to specify position of a column in a table

MySQL LIMIT & OFFSET with Examples

What is the LIMIT keyword?

The limit keyword is used to limit the number of rows returned in a query result.

It can be used in conjunction with the SELECT, UPDATE OR DELETE commands LIMIT keyword syntax

The syntax for the LIMIT keyword is as follows

```
SELECT {fieldname(s) | *} FROM tableName(s) [WHERE condition] LIMIT  N;
```

HERE

- **"SELECT {fieldname(s) | *} FROM tableName(s)"** is the SELECT statement containing the fields that we would like to return in our query.

- **"[WHERE condition]"** is optional but when supplied, can be used to specify a filter on the result set.
- **"LIMIT N"** is the keyword and **N** is any number starting from 0, putting 0 as the limit does not return any records in the query. Putting a number say 5 will return five records. If the records in the specified table are less than N, then all the records from the queried table are returned in the result set.

Let's look at an example -

```
SELECT *  FROM members LIMIT 2;
```

membership_number	**full_names**	**gender**	**date_of_birth**	**date_of_registration**	**physical_address**	**postal_address**	**contact_number**	**email**	**credit_card_number**
1	Janet Jones	Female	21-07-1980	NULL	First Street Plot No 4	Private Bag	0759 253 542	janetjones@yagoo.cm	NULL
2	Janet Smit	Female	23-06-1980	NULL	Melrose 123	NULL	NULL	jj@fstreet.com	NULL

	h Jones								

As you can see from the above screenshot, only two members have been returned.

Getting a list of ten (10) members only from the database

Let's suppose that we want to get a list of the first 10 registered members from the Myflix database. We would use the following script to achieve that.

```
SELECT *  FROM members LIMIT 10;
```

Executing the above script gives us the results shown below

membership_number	full_names	gender	date_of_birth	date_of_registration	physical_address	postal_address	contact_number	email	credit_card_num

									ber
1	Janet Jones	Female	21-07-1980	NULL	First Street Plot No 4	Private Bag	0759253542	janetjones@yagoo.cm	NULL
2	Janet Smith Jones	Female	23-06-1980	NULL	Melrose 123	NULL	NULL	jj@fstreet.com	NULL
3	Robert Phil	Male	12-07-1989	NULL	3rd Street 34	NULL	12345	rm@tstreet.com	NULL
4	Gloria Williams	Female	14-02-1984	NULL	2nd Street 23	NULL	NULL	NULL	NULL
5	Leonard Hofstadter	Male	NULL	NULL	Woodcrest	NULL	845738767	NULL	NULL
6	Sheldon Cooper	Male	NULL	NULL	Woodcrest	NULL	976736763	NULL	NULL

7	Rajesh Koothrappali	Male	NULL	NULL	Woodcrest	NULL	938867763	NULL	NULL
8	Leslie Winkle	Male	14-02-1984	NULL	Woodcrest	NULL	987636553	NULL	NULL
9	Howard Wolowitz	Male	24-08-1981	NULL	South Park	P.O. Box 4563	987786553	lwolowitz[at]email.me	NULL

Note only 9 members have been returned in our query since N in the LIMIT clause is greater than the number of total records in our table.

Re-writing the above script as follows

```
SELECT *  FROM members LIMIT 9;
```

Only returns 9 rows in our query result set.

Using the OFF SET in the LIMIT query

The **OFF SET** value is also most often used together with the LIMIT keyword. The OFF SET value allows us to specify which row to start from retrieving data

Let's suppose that we want to get a limited number of members starting from the middle of the rows, we can use the LIMIT keyword together with the offset value to achieve that. The script shown below gets data starting the second row and limits the results to 2.

```
SELECT * FROM `members` LIMIT 1, 2;
```

Executing the above script in MySQL workbench against the myflixdb gives the following results.

membership_number	**full_names**	**gender**	**date_of_birth**	**date_of_registration**	**physical_address**	**postal_address**	**contact_number**	**email**	**credit_card_number**
2	Janet Smith	Female	23-06-1980	NULL	Melrose 123	NULL	NULL	jj@fstreet.com	NULL

	Jones								
3	Robert Phil	Male	12-07-1989	NULL	3rd Street 34	NULL	12345	rm@tstreet.com	NULL

Note that here **OFFSET = 1** Hence row#2 is returned & **Limit = 2**, Hence only 2 records are returned

When should we use the LIMIT keyword?

Let's suppose that we are developing the application that runs on top of myflixdb. Our system designer have asked us to limit the number of records displayed on a page to say 20 records per page to counter slow load times. How do we go about implementing the system that meets such user requirements? The LIMIT keyword comes in handy in such situations. We would be able to limit the results returned from a query to 20 records only per page.

Summary

- The LIMIT keyword of is used to limit the number of rows returned from a result set.

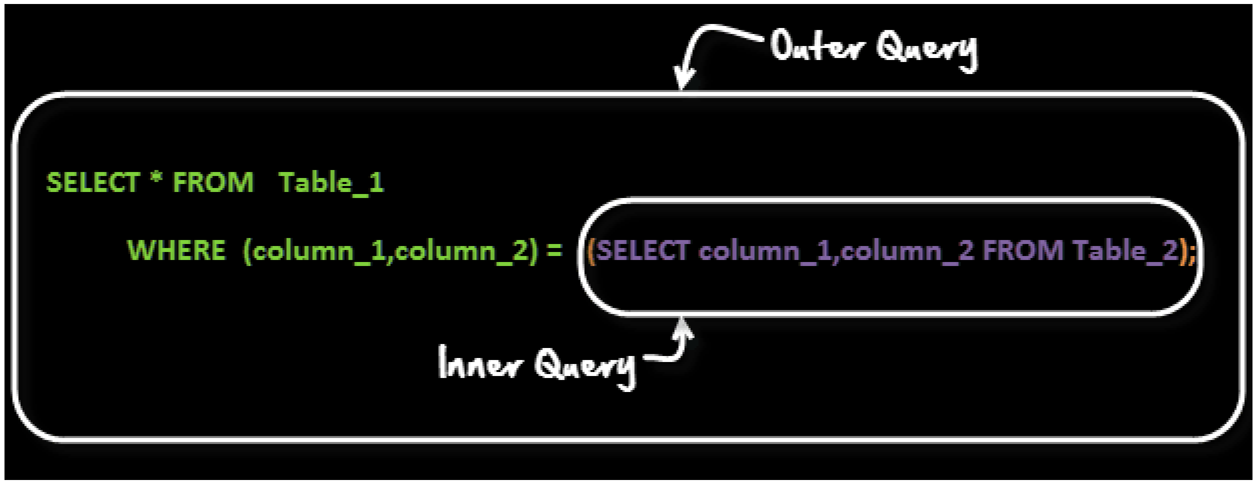
- The LIMIT number can be any number from zero (0) going upwards. When zero (0) is specified as the limit, no rows are returned from the result set.
- The OFF SET value allows us to specify which row to start from retrieving data
- It can be used in conjunction with the SELECT, UPDATE OR DELETE commands LIMIT keyword syntax

MySQL SubQuery Tutorial with Examples

What are sub queries?

A sub query is a select query that is contained inside another query. The inner select query is usually used to determine the results of the outer select query.

Let's look into the sub query syntax -



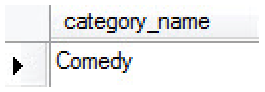
```
SELECT * FROM   Table_1
      WHERE  (column_1,column_2) = (SELECT column_1,column_2 FROM Table_2);
```

A common customer complaint at the MyFlix Video Library is the low number of movie titles. The management wants to buy movies for a category which has least number of titles.

You can use a query like

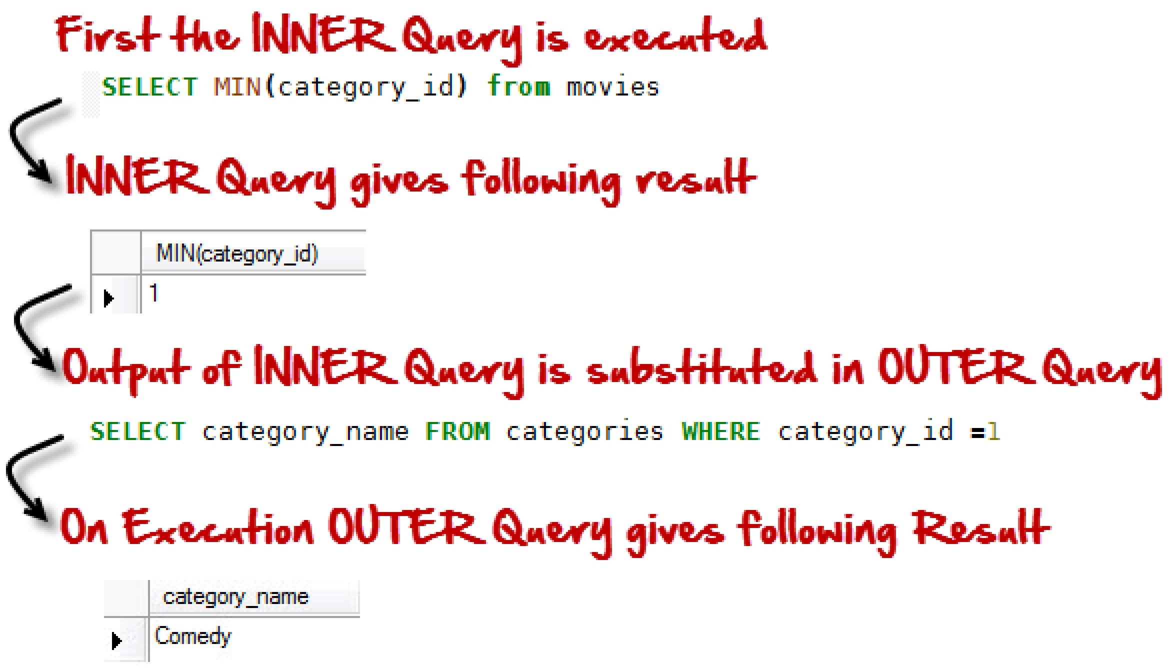
```
SELECT category_name FROM categories WHERE category_id =( SELECT MIN(category_id) from movies);
```

It gives a result



category_name
Comedy

Let's see how this query works



First the INNER Query is executed

```
SELECT MIN(category_id) from movies
```

INNER Query gives following result

	MIN(category_id)
▶	1

Output of INNER Query is substituted in OUTER Query

```
SELECT category_name FROM categories WHERE category_id =1
```

On Execution OUTER Query gives following Result

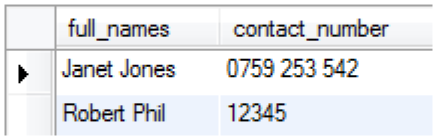
	category_name
▶	Comedy

The above is a form of **Row Sub-Query**. In such sub-queries the , inner query can give only ONE result. The permissible operators when work with row subqueries are [=, >, =, <=, ,!=,]

Let's look at another example ,

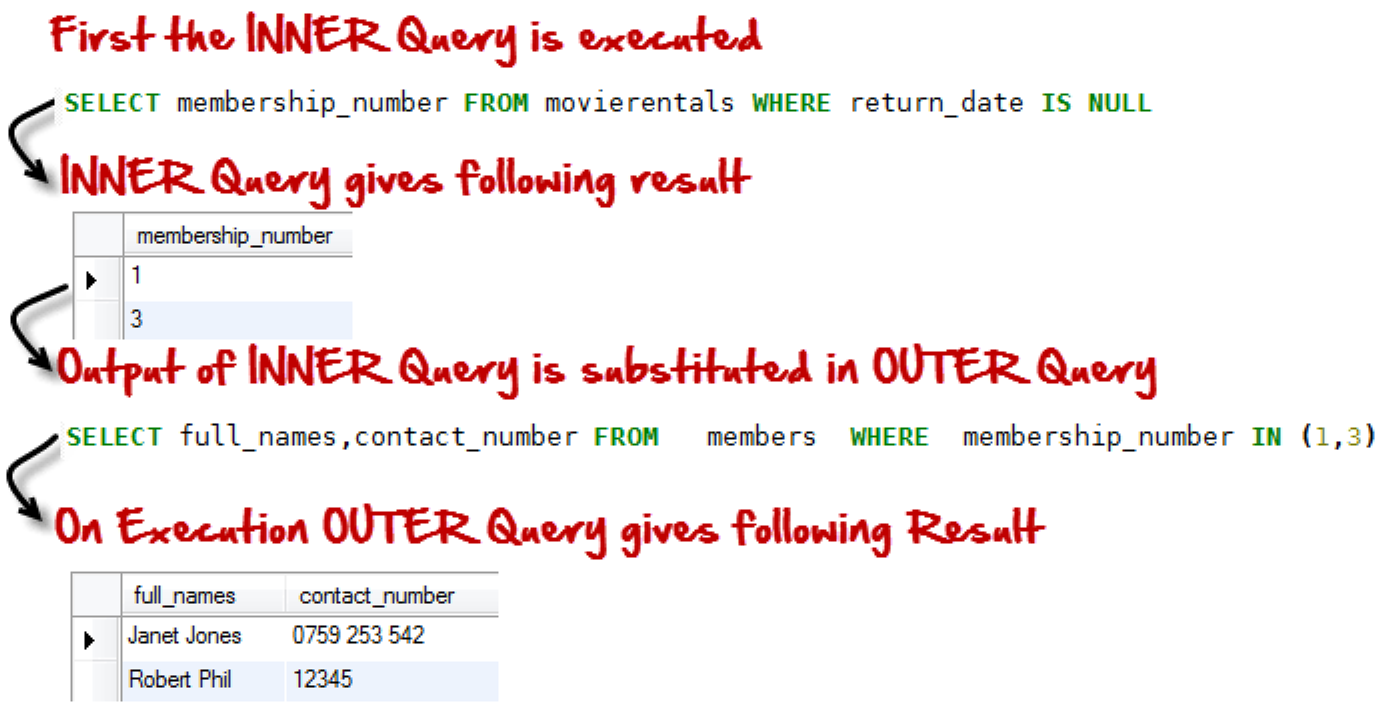
Suppose you want Names and Phone numbers of members of people who have rented a movie and are yet to return them. Once you get Names and Phone Number you call them up to give a reminder. You can use a query like

```
SELECT full_names,contact_number FROM   members  WHERE  membership_number IN (SELECT membership_number FROM movierentals WHERE return_date IS NULL );
```



full_names	contact_number
Janet Jones	0759 253 542
Robert Phil	12345

Let's see how this query works


First the INNER Query is executed

```
SELECT membership_number FROM movierentals WHERE return_date IS NULL
```

INNER Query gives following result

membership_number
1
3

Output of INNER Query is substituted in OUTER Query

```
SELECT full_names,contact_number FROM   members   WHERE   membership_number IN (1,3)
```

On Execution OUTER Query gives following Result

full_names	contact_number
Janet Jones	0759 253 542
Robert Phil	12345

In this case, the inner query returns more than one results. The above is type of T**able sub-quer**y.

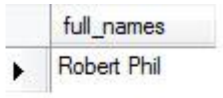
Till now we have seen two queries , lets now see an example of **triple query**!!!

Suppose the management wants to reward the highest paying member.

We can run a query like

```
Select full_names From members WHERE membership_number = (SELECT me
mbership_number FROM payments WHERE amount_paid = (SELECT MAX(amou
nt_paid) FROM payments));
```

The above query gives the following result -



```
full_names
Robert Phil
```

Sub-Queries Vs Joins!

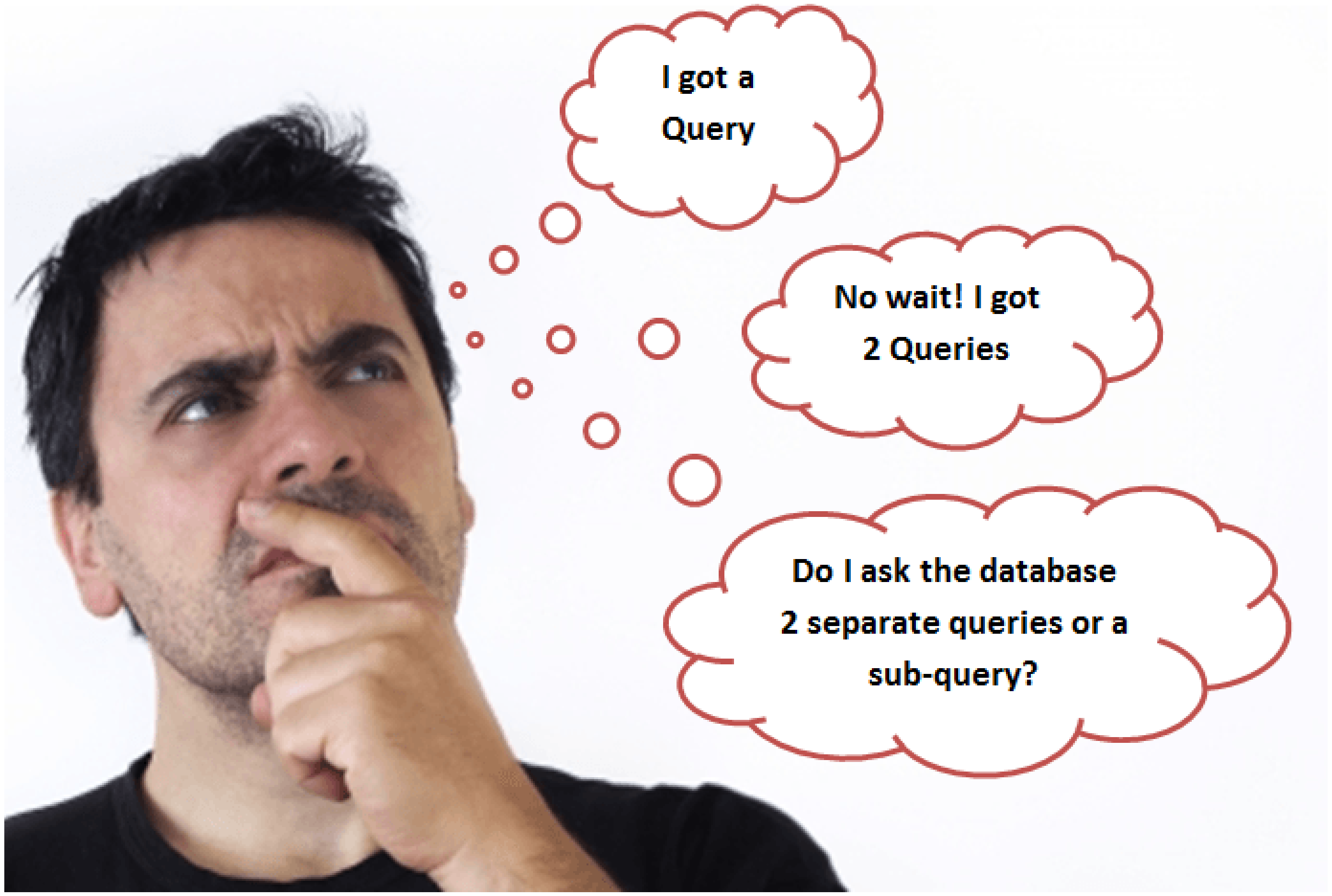
When compare with Joins , sub-queries are simple to use and easy to read. They are not as complicated as Joins

Hence there are frequently used by SQL beginners.

But sub-queries have performance issues. Using a join instead of a sub-query can at times give you upto 500 times performance boost.

Given a choice, it is recommended to use a JOIN over a sub query.

Sub-Queries should only be used as a fallback solution when you cannot use a JOIN operation to achieve the above



Summary

- Subqueries are embedded queries inside another query. The embedded query is known as the inner query and the container query is known as the outer query.

- Sub queries are easy to use, offer great flexibility and can be easily broken down into single logical components making up the query which is very useful when Testing and debugging the queries.
- MySQL supports three types of subqueries, scalar, row and table subqueries.
- Scalar sub queries only return a single row and single column.
- Row sub queries only return a single row but can have more than one column.
- Table subqueries can return multiple rows as well as columns.
- Subqueries can also be used in INSERT, UPDATE and DELETE queries.
- For performance issues, when it comes to getting data from multiple tables, it is strongly recommended to use JOINs instead of subqueries. Sub queries should only be used with good reason.

MySQL Index Tutorial - Create, Add & Drop

What are Index?

Nobody likes slow systems.

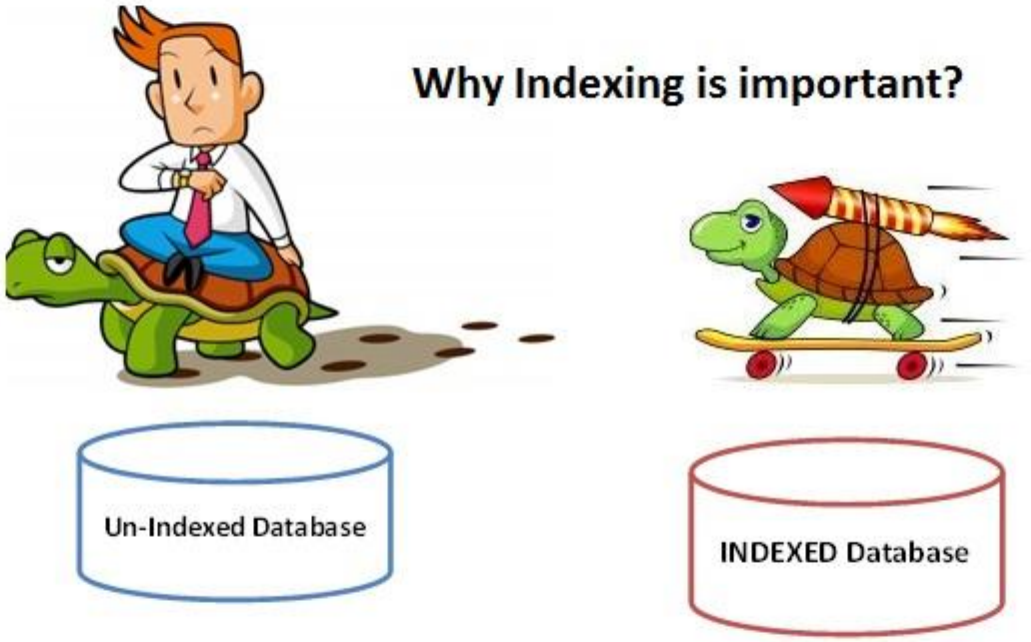
High system performance is of prime importance in almost all database systems
.

Most businesses invest heavily in hardware so that data retrievals and manipulations can be faster.

But there is limit to hardware investments a business can make.

Optimizing your database is a cheaper and better solution.

Towards this end we can use INDEXES.



- The slowness in the response time is usually due to the records being stored randomly in database tables.

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

Create index basic syntax

Indexes can be defined in 2 ways

1. At the time of table creation

2. After table has been created

Example:

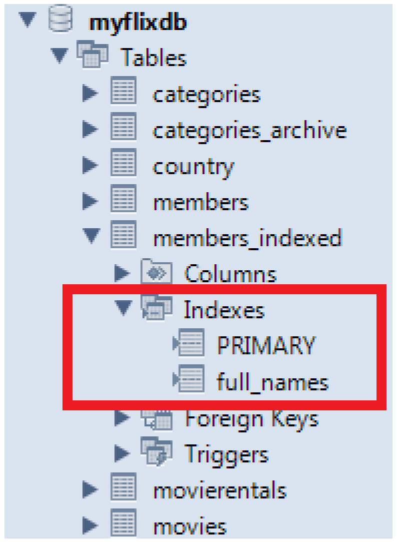
For our myflixdb we expect lots of searches to the database on full name.

We will add the "full_names" column to Index in a new table "members_indexed".

The script shown below helps us to achieve that.

```
CREATE TABLE `members_indexed` (
  `membership_number` int(11) NOT NULL AUTO_INCREMENT,
  `full_names` varchar(150) DEFAULT NULL,
  `gender` varchar(6) DEFAULT NULL,
  `date_of_birth` date DEFAULT NULL,
  `physical_address` varchar(255) DEFAULT NULL,
  `postal_address` varchar(255) DEFAULT NULL,
  `contact_number` varchar(75) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`membership_number`),INDEX(full_names)
) ENGINE=InnoDB;
```

Execute the above SQL script in MySQL workbench against the "myflixdb".



Refreshing the myflixdb shows the newly created table named members_indexed.

"Note" members_indexed table has "full_names" in the indexes node.

As the members base expand and the number of records increases , search queries on the members_indexed table that use the WHERE and ORDER BY clauses will be much faster compared to the ones performed the members table without the index defined.

Add index basic syntax

The above example created the index when defining the database table. Suppose we already have a table defined and search queries on it are very slow. They take too long to return the results. After investigating the problem, we discover that we can greatly improve the system performance by creating INDEX on the most commonly used column in the WHERE clause.

We can use following query to add index

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

Let's suppose that search queries on the movies table are very slow and we want to use an index on the "movie title" to speed up the queries, we can use the following script to achieve that.

```
CREATE INDEX `title_index` ON `movies`(`title`);
```

Executing the above query creates an index on the title field in the movies table.

This means all the search queries on the movies table using the "title" will be faster.

Search queries on other fields in the movies table will however still are slower compared to the ones based on the indexed field.

Note you can create indexes on multiple columns if necessary depending on the fields that you intend to use for your database search engine.

If you want to view the indexes defined on a particular table, you can use the following script to do that.

```
SHOW INDEXES FROM table_name;
```

Let's now take a look at all the indexes defined on the movies table in the myflixdb.

```
SHOW INDEXES FROM `movies`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results shown below.

Note the primary and foreign keys on the table have already been indexed by MySQL. Each index has its own unique name and the column on which it is defined is shown as well.

Drop index basic syntax

The drop command is used to remove already defined indexes on a table.

There may be times when you have already defined an index on a table that is frequently updated. You may want to remove the indexes on such a table to improve the UPDATE and INSERT queries performance. The basic syntax used to drop an index on a table is as follows.

```
DROP INDEX `index_id` ON `table_name`;
```

Let's now look at a practical example.

```
DROP INDEX ` full_names` ON `members_indexed`;
```

Executing the above command drops the index with id ` full_names ` from the members_indexed table.

Summary

- Indexes are very powerful when it comes to greatly improving the performance of MySQL search queries.
- Indexes can be defined when creating a table or added later on after the table has already been created.
- You can define indexes on more than one column on a table.
- The SHOW INDEX FROM table_name is used to display the defined indexes on a table.
- The DROP command is used to remove a defined index on a given table.

```
mysqlslap --user=root --password --host=localhost  --concurrency=50 --
iterations=2 --create-schema=testcourse --query="SELECT * FROM
testcourse.departments WHERE MANAGER_ID > 201;" --verbose
```

```
SELECT * FROM demodemo.jobs

INTO OUTFILE 'C:/ProgramData/MySQL/MySQL Server 5.6/Uploads/orders.csv'

FIELDS TERMINATED BY ','

ENCLOSED BY '"'

LINES TERMINATED BY '\n';
```

```
SHOW VARIABLES LIKE "secure_file_priv";
```