# SQL & MYSQL NOTES

## Database



Ashirbad Swain

## SQL

**Structured Query Language** is a standard database programming language used for accessing and manipulating data in a database. **Mainly**, SQL language is used to query a database.

The SQL query to select all records from the user's table:

```
SELECT * from users;
```

The SQL query to delete single records from users table by using where clause:

```
DELETE FROM users WHERE user_id = 299;
```

## Database

A database is systematically organized storage of information, and it allows easy insertion, updating, analysis, and retrieval of data.

### Advantages of using Database

- Database minimizes data redundancy to a great extent
- The database can control inconsistency of data to a large extent
- Sharing of data is also possible using the database
- Use of database can ensure data security
- Integrity can be managed using the database

### Level of database implementation

The database is implemented through three general levels. These levels are:

- Internal level or physical level
- Conceptual level
- External or view level

## Relation

In general, a relation is a table i.e., data is arranged in rows and columns.

## Tuple

The rows of a table in a relationship are generally termed as tuples.

## Attributes

The columns or fields of a table is termed as attributes.

## Degree

The number of attributes in a relation determines the degree of relation.

## Cardinality

The number of tuples or rows in a relation is termed as cardinality.

## SQL Syntax

- SQL Syntax is unique sets of rules and guidelines.
- Syntax rules:
  - SQL statements always start with the keywords.
  - SQL statement ends with a semicolon.
  - SQL is not case sensitive, means the **update** is the same as UPDATE.
- SELECT Statement

```
//Syntax
SELECT column_name(s) FROM table_name
```

- SELECT Statement with WHERE Clause

```
//Syntax
SELECT [*] FROM [TableName] WHERE [Condition1]
```

- SELECT Statement with WHERE AND/OR Clause

```
//Syntax
SELECT [*] FROM [TableName] WHERE [condition1] [AND[OR]] [condition2] ...
```

- SELECT Statement with ORDER BY

```
//Syntax
SELECT column_name () FROM table_name ORDER BY column_name ASC or DESC
```

- INSERT INTO Statement

```
//Syntax
INSERT INTO table_name (column. Column1, column2, ...) VALUES (value, value1, value2, ...)
```

- UPDATE Statement

```
//Syntax
UPDATE table_name SET column=value, column1=value1, ... WHERE someColumn=someValue
```

- DELETE Statement

```
//Syntax
DELETE FROM table_name WHERE someColumn = someValue
```

## SQL Commands

| SQL Command | Description |
|---|---|
| CREATE DATABASE | Creates a new database |
| CREATE TABLE | Creates a new table |
| ALTER DATABASE | Modifies a database |
| ALTER TABLE | Modifies a table |
| DROP TABLE | Deletes a table |
| CREATE INDEX | Creates an index |
| DROP INDEX | Deletes an index |
| SELECT | Fetch data from database tables |
| UPDATE | Modify data in a database table |
| DELETE | Deletes data from a database table |
| INSERT INTO | Inserts new data into a database table |

## SQL CREATE DATABASE

The SQL CREATE DATABASE Statement is used to create a new database.

```
//Syntax
CREATE DATABASE database_name;

//Example
CREATE DATABASE my_database;
```

## SQL CREATE TABLE

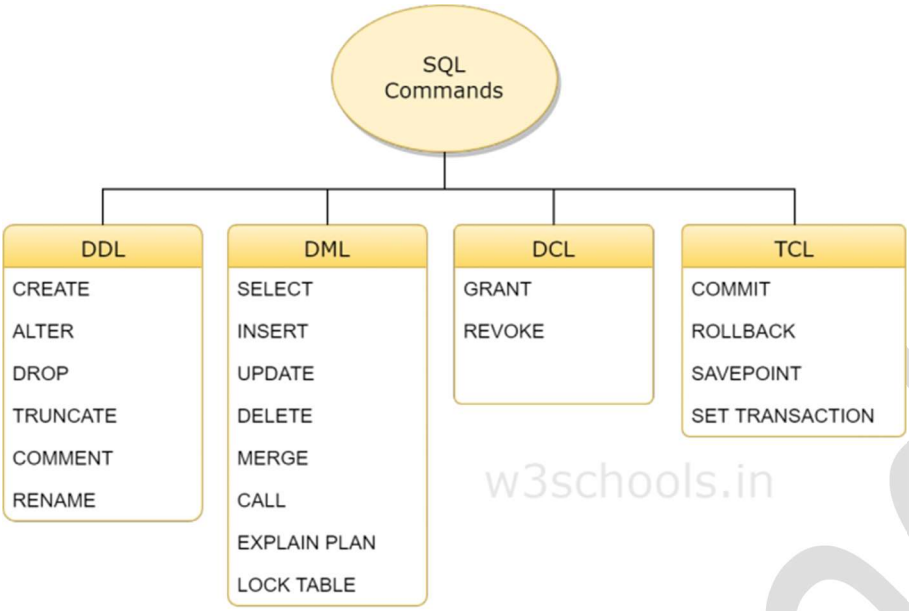
The SQL CREATE TABLE Statement is used to create a new table in a database.

```
//Syntax
CREATE TABLE table_name
(
Column_name1 datatype(size) [NULL | NOT NULL],
Column_name2 datatype(size) [NULL | NOT NULL],
Column_name3 datatype(size) [NULL | NOT NULL],
....
);

//Example
CREATE TABLE tbl_employee
(
employee_id INT NOT NULL,
last_name VARCHAR (100) NOT NULL,
first_name VARCHAR (100) NOT NULL,
address VARCHAR (255) NULL
);
```

## SQL Commands

SQL commands are divided into four subgroups,

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DCL (Data Control Language)
- TCL (Transaction Control Language)






## DDL (Dr. CAT)

**Data Definition Language** deals with database schemas and descriptions, of how the data should reside in the database.

- CREATE – to create a database and its objects like (table, index, views, store, procedures, functions and triggers)
- ALTER – alter the structure of the existing database
- DROP – delete objects from the database
- TRUNCATE – removes all records from a table, including all spaces allocated for the records are removed
- COMMENT – add comments to the data dictionary
- RENAME – rename an object

## DML (Uid)

**Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such as SELECT, INSERT, UPDATE, DELETE etc.

- SELECT – retrieve data from a database
- INSERT – insert data into a table
- UPDATE – updates existing data within a table
- DELETE – delete all records from a database table
- MERGE – merge to tables
- CALL – call a PL/SQL or Java subprogram
- EXPLAIN PLAN – interpretation of the data access path
- LOCK TABLE – concurrency control

## DCL

- **Data Control Language** which includes commands such as GRANT and mostly concerned with rights, permissions and other control of the database system.
- GRANT – allow users access privileges to the database
- REVOKE – withdraw users access privileges given by using the GRANT command

## TCL

- **Transaction Control Language** deals with a transaction within a database.
- COMMIT – commits a transaction
- ROLLBACK – rollback a transaction in case of any error occurs
- SAVEPOINT – to roll back the transaction making points within groups
- SET TRANSACTION – specify characteristics of the transaction

## What is Data?

Data can be facts related to any object in consideration.

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

## What is Database?

A database is a systematic collection of data. They support electronic storage and manipulation of data. Generally, Database make management easy.

**Example**:

- An online telephone directory uses a database to store data of people, phone numbers, other contact details.
- Your electricity service provides uses a database to manage billing, client-related issues, handle fault data etc.
- Consider Facebook, it needs to store, manipulate and present data related to members, their friends, member activities, messages, advertisements and a lot more.

## MySQL

MySQL is an open source relational database management system.

## Difference Between SQL & MySQL

- SQL is a language which is used to operate your database whereas MySQL is one of the open-source relational database available in the market.
- SQL is used in the accessing, updating and manipulating of data in a database while MySQL is an RDBMS that allows keeping the data that exists in a database organized.
- SQL is a query language while MySQL is a database software.

## Why use MySQL?

There are number of relational databases like Microsoft SQL server, Microsoft Access, Oracle etc. Then why to choose MySQL over other database management systems.

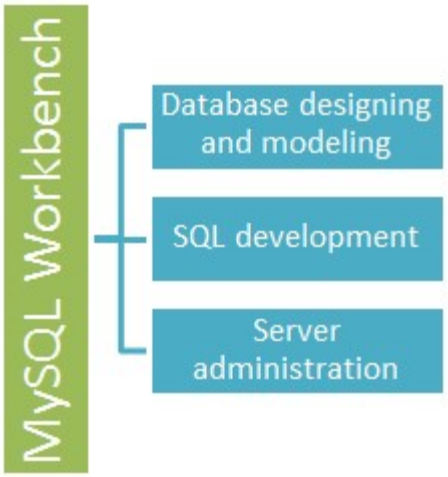
- MySQL supports multiple storage engines (like InnoDB, MyISAM) each with its own specifications while other systems like SQL server only support a single storage engine.
- MySQL has high performance compared to other relational database systems. This is due to its simplicity in design and support multiple-storage engines.
- Cost effective, it's relatively cheaper in terms of cost when compared to other database systems.
- Cross Platform, MySQL works on many platforms which means it can be deployed on most machines.

## What is MySQL Workbench?

**MySQL Workbench** 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.

The purpose of MySQL Workbench is to provide the interface to work with databases more easily and in a more structured way.




## MySQL Workbench – Modeling and Design Tool

- MySQL workbench has tools that allow developers and database administrators visually create physical database design models that can be easily translated into MySQL databases using forward engineering.
- It supports all objects such as tables, views, stored procedures, triggers etc. that make up a database.
- MySQL workbench supports creation of multiple models in the same environment.

- MySQL workbench has built in model validating utility that reports any issues that might be found to the data modeler.
- It also allows for different modeling notations and can be extended by using LUA scripting language.

**MySQL Workbench – SQL Development tool**

- MySQL workbench has built in SQL visual editor.
- The visual SQL editor allows developers to build, edit and run queries against MySQL server databases. It has utilities for viewing data and exporting it.
- The syntax color highlighters help developers easily write and debug SQL statements.
- Multiple queries can be run and results automatically displayed in different tabs.

**MySQL Workbench – Administration tool**

Server administration plays a critical role in securing the data of the company. Workbench MySQL has the following features that simplify the process of MySQL server administration.

- User Administration:

Visual utility for managing users that lets database administrators easily add new and remove existing users if need arises, grant and drop privileges and view profiles0.

- Server Configuration:

allows for advanced configuration of the server and fine tuning for optimal performance.

- Database Backup & Restorations:

visual tool for exporting/importing MySQL dump files. MySQL dump files contain SQL scripts for creating databases, tables, views, stored procedures and insertion of data.

- Server logs:

visual tool for viewing MySQL server logs. The logs include error logs, binary logs and InnoDB logs. These logs come in handy when performing diagnosis on the server.

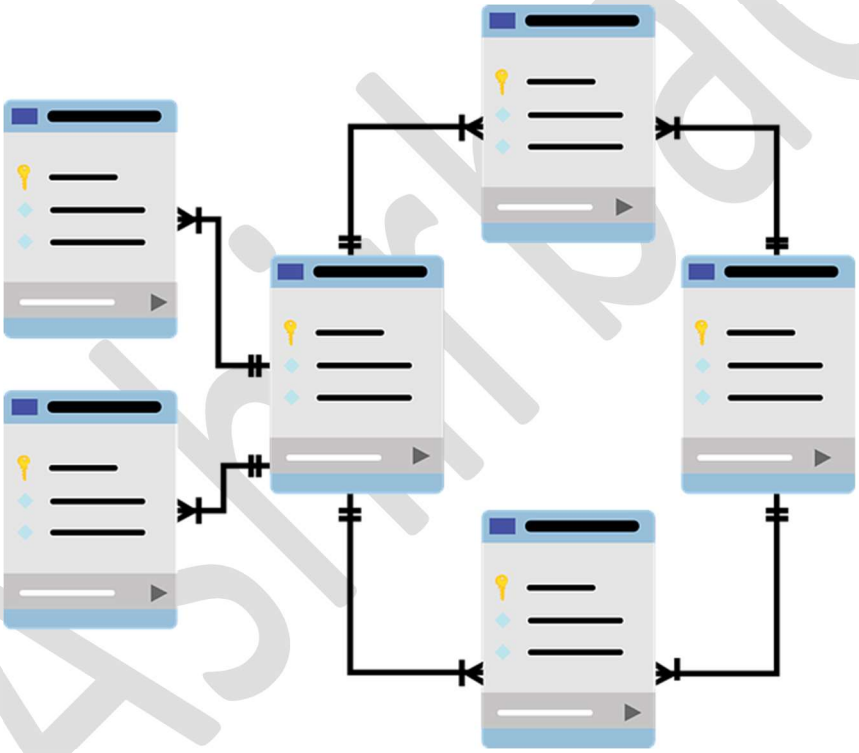
## Database Design

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

The **main objective** of database designing is 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 into physical media using hardware resources and software systems such as database management systems (DBMS).



### Why Database Design is important?

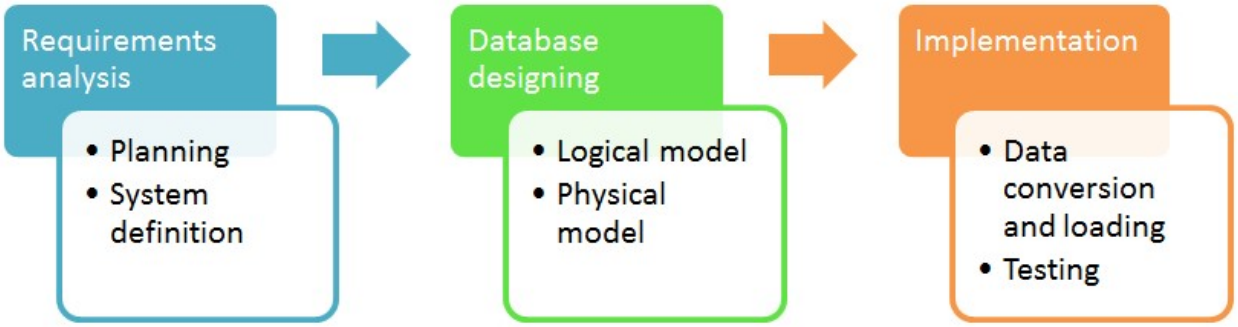
It helps produce database systems:

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

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

**Note**: The genius of database is in its design. Data operations using SQL is relatively simple.

## Database Development Life Cycle




### Requirement analysis

- Planning:

  This stage concerns with planning of entire Database Development Life Cycle. 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 & 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.

## Types of Database Designing

There are 2 types of Database Designing Techniques,

1. Normalization
2. ER Modeling

## Database Creation

We can create a database in **2** ways,

1. By executing a simple SQL query
2. By using forward engineering in MySQL Workbench

## Create Database

CREATE DATABASE is the SQL command for creating a database.

Suppose 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.

## 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 lets 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 & 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.

## Creating Tables MySQL

Tables can be created using CREATE TABLE statement and the syntax is as follows:

```
CREATE TABLE [IF NOT EXISTS] 'Tablename' ('Fieldname' datatype [optional 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 "**datatype**" 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.

**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),

'conatct_number' VARCHAR (75),

'email' VARCHAR (255),

PRIMARY KEY ('membership_number'))

ENGINE = InnoDB;
```

## Data Types

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

MySQL has 3 main categories of data types namely:

1. Numeric
2. Text
3. Date/Time

## Numeric Datatypes

Numeric datatypes 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 datatypes.

| | |
|---|---|
| TINYINT () | -128 to 127 normal<br>0 to 255 unsigned |
| SMALLINT () | -32768 to 32767 normal<br>0 to 65535 unsigned |
| MEDIUMINT () | -8388608 to 8388607 normal<br>0 to 16777215 unsigned |
| INT () | -2147483648 to 2147483647 normal<br>0 to 4294967295 unsigned |
| BIGINT () | -9223372036854775808 to 9223372036854775807 normal<br>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 Datatypes

These are used to store text values.

| | |
|---|---|
| CHAR () | A fixed section from 0 to 255 character 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 |

**NOTE**:

- Use upper case letter 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.
- Creating database involves translating the logical database design model into the physical database.
- MySQL supports a number of datatypes for numeric, dates and string values.
- CREATE DATABASE command is used to create a database.
- CREATE TABLE command is used to create tables in a database.
- MySQL workbench supports forward engineering which involves automatically generating SQL scripts from the logical database model that can be executed to create the physical database.
- A visual data model can be transformed into a physical database on a target MySQL server by executing the **forward engineering** wizard.
- Also, MySQL workbench also enables you to **reverse engineer** an existing database or packaged application to get better insight into its database design.

## MySQL SELECT Query

SELECT Query is used to fetch the data from the MySQL Database. Databases store data for later retrieval.

The purpose of MySQL Select is to return from the database tables, one or more rows that match a given criteria.

Syntax:

```
SELECT [DISTINCT | ALL] {* | [fieldExpression [AS newName]]} FROM tablename [alias] [WHERE condition] [GROUP BY fieldname (s)] [HAVING condition] ORDER 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 strings 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 o put together records that have the same field values.
- **HAVING** condition is used to specify criteria when 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.

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

The above statement selects all the fields from the member's table. The semicolon is a statement terminate.

Example:

Table 1: **members** table

| 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 23 | NULL | NULL | NULL |

Table 2: **movies** table

| movie_id | title | director | year_released | category_id |
|---|---|---|---|---|
| 1 | Pirates of the Caribbean 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 Member listing

Suppose we want to get a list of all the registered library members from our database, we would use the script shown to do that.

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

The result will be,

Our above query has returned all the rows and columns from the member's table.

| 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 23 | NULL | NULL | NULL |

Let's say we are only interested in getting only the full_names, gender, physical_address and email fields only.

Then the script will be:

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

| 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

Let's say we want to get a list of movies 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 script will be,

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

Here,

- The **Concat ()** MySQL function is used to join the columns value 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:

| Concat (`title`, ' (', `director`, ')') | year_released |
|---|---|
| Pirates of the Caribbean 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

Suppose we want to use a more descriptive field name in our result set. We would use the column alias name to achieve that.

```
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 se as the field name.

Suppose,

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

The result will be,

| Concat | year_released |
|---|---|
| Pirates of the Caribbean 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.

```
SELECT 'membership_number', 'full_names', LEFT ('date_of_birth', 4) AS 'year_of_birth' FROM members;
```

Here,

- "LEFT ('date_of_birth', 4)" the LEFT string function accepts the date of birth as the parameter and only return 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.

By executing the above query, the result will be:

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

## Where Clause

When we want to restrict the query results to a specified condition, the SQL WHERE clause comes in handy in such situations.

Syntax:

```
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.

## Example:

Suppose we want to get a member's personal details from members table given the membership number 1,

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

Executing the above query in MySQL Workbench on the "**MyFlixDB**" would produce the following result,

| 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 | janet jones @ya goo.c m |

## WHERE clause combined with – AND logical operator

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

Example:

Suppose we want to get a list of all the movies in category 2 that released in 2008, the query will be,

```
SELECT * FROM 'movies' WHERE 'category_id' = 2 AND 'year_released' = 2008;
```

## 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.

Example:

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;
```

## 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.

Example:

The following query gives rows where membership_number is either 1, 2 or 3.

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

## WHERE clause combined with – NOT IN keyword

The WHERE clause when used together with the NOT IN keyword DOES NOT affect the rows whose values matches the list of values provided in the NOT IN keyword.

Example:

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);
```

## WHERE clause combined with – COMPARISON operators

### Equals To (=)

The following scripts gets all the females from the members table using the equal to comparison operator.

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

### Greater than (>)

The following script gets all the payments that are greater than 2000 from the payments table.

```
SELECT * FROM 'payments' WHERE 'amount_paid' > 2000;
```

### Not Equal To (<>)

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

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

NOTE:

- 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 & 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 keyword IN is used to select rows matching a list of values.

## MySQL INSERT Query

INSERT INTO is used to store data in the tables. The INSERT command creates a new row in the table to store data.

Syntax:

```
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 a new row into a table named 'table_name'.
- **(column_1, column_2, ...)** specifies the columns to be updated in the new MySQL 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:

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

Example:

Suppose, we have the following list of new library members that need to be added to 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 |

Let's insert data one by one. We will start with Leonard Hofstadter. We will treat the contact number as a numeric datatype and not enclose the number in single quotes.

```
INSERT INTO 'members' ('full_names', 'gender', 'physical_address', 'contact_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 zero (0) at the beginning dropped since it's not significant.

To avoid such problems, the value must be enclosed in single quotes as shown below,

```
INSERT INTO 'members' ('full_names', 'gender', 'physical_address', 'contact_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 in MySQL as long as the correct values have been mapped to the correct columns.**

The query shown below demonstrates the above point.

```
INSERT INTO 'members' ('conatct_number', 'gender', 'full_names', 'physical_address') VALUES ('0938867763', 'male', 'Rajesh Koothrappali', 'Woodcrest');
```

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

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

```
INSERT INTO 'members' ('full_names', 'date_of_birth', 'gender', 'physical_address', 'contact_number') VALUES ('Leslie Winkle', '1984-02-14', 'Male', 'Woodcrest', '0987636553');
```

All the above queries specify the columns and mapped them to values in MySQL insert statement.

If we are supplying values for ALL the columns in the table, then we can omit the columns from the MySQL insert query.

```
INSERT INTO 'members' VALUES (9, 'Howard Wolowitz', 'Male', '1981-08-24', 'SouthPark', 'P.O. Box 4563', '0987786553', 'lwolowitz@email.me');
```

## Inserting into a Table from another Table

The INSERT command can also be used to insert data into a table from another table. The syntax will be:

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

Example:

We will create a dummy table for movie categories for demonstration purposes. We will call the new categories table categories_archive.

```
CREATE TABLE 'categories_archive' (

'category_id' int (11) AUTO_INCREMENT,

'category_name' varchar (150) DEFAULT NULL,

'remarks' 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 up 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 structure 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 will be:

```
INSERT INTO 'categories_archive' (category_id, category_name, remarks) SELECT category_id, category_name, remarks FROM 'categories';
```

NOTE:

- 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 table.

## MySQL DELETE Query

MySQL **DELETE** command is used to delete rows that are no longer required from the database tables.

It deleted the whole row from the table and returns count of deleted rows.

Delete command comes in handy to delete temporary or obsolete data from your database.

The **DELETE** query in MySQL can delete more than one row from a table in a single query.

Once a Delete row in MySQL row has been deleted, it **can not 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.

To delete a row in a MySQL table, use the **DELETE FROM** statement:

```
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 WHERE clause is not used in the DELETE MySQL query, then all the rows in a given table will be deleted.

Example:

Suppose that the MyFlixDB video library no longer wishes to be renting out "The Great Dictator" to its member 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;
```

NOTE:

- You cannot delete a single column for a table. You can delete an entire row.
- The "WHERE clause" is used to limit the number of rows affected by the DELETE Query.
- The delete command is used to remove data that is no longer required from a table.

## MySQL UPDATE Query

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 MySQL table with values from another table.

The basic syntax of the update query in MySQL is:

```
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 datatypes must ne in single quotes. Numeric values do not need to be in quotation marks. Date datatype 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 UPDATE MySQL query.

Example:

Consider the "members" table.

Suppose that our member's membership numbers 1 & 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 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;
```

By executing the above query, the result will be,

| 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 | 999 | janetjones@ yahoo.com |

Let's now update the contact number using the script,

```
UPDATE 'members' SET 'contact_number' = '0759253542' WHERE 'membership_number' = 1;
```

Executing the above script updates the contact number from 999 to 0759253532 for membership number 1.

Now, let's update the membership number 2.

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

| Membership_ number | Full_ names | Gender | Date_of_birth | Physical_ address | Postal_ Address | Contact_ Number | email |
|---|---|---|---|---|---|---|---|
| 2 | Janet Smith Jones | Female | 23-06-1980 | Melrose 123 | Null | Null | jj@fstreet.com |

**NOTE**:

- 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.

## Data Sorting

### ORDER BY in MySQL: DESC & ASC

**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 datatypes.

### MySQL – ORDER BY

**MySQL ORDER** BY is used in conjunction with the SELECT query to sort data in an orderly manner. The order by clause is used to sort the query results sets in either ascending or descending order.

```
SELECT statement... [WHERE condition | GROUP BY 'field_name(s)' HAVING condition] 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 results in either ascending or descending order. (**ASC** is used as the default).

| ASC Keyword | DESC Keyword |
|---|---|
| It is used to sort the query results in a top to bottom style. | It is used to sort query results in a bottom to top style. |
| When working on the date data types, the earliest date is shown on the top of the list. | When working on date data types, the latest date is shown on top of the list. |
| When working on the numeric data types, the lowest values are shown on top of the list. | When working on the 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 letter A going up to the letter Z. | 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. |

## DESC and ASC Syntax

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

Here,

- **SELECT {fieldName(s) | *} FROM tableName(s)** is the statement containing the fields and tables(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 is mandatory and is the field on which the sorting is to be performed.
- **[LIMIT]** is optional but can be used to limit th number of results returned from the query result set.

### Example 1:

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;
```

| 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 | |
| 1 | Janet Jones | Female | 1980-07-21 | |
| 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 |


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

For in ascending order:

```
SELECT * FROM members ORDER BY date_of _birth ASC;
```

| membership_number | full_names | gender | date_of_birth | physical_address |
|---|---|---|---|---|
| 5 | Leo | | NULL | Woodcrest |
| 6 | Sh | | NULL | Woodcrest |
| 7 | Rajes | | NULL | Woodcrest |
| 8 | Leslie Winkle | | 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 |


Example 2:

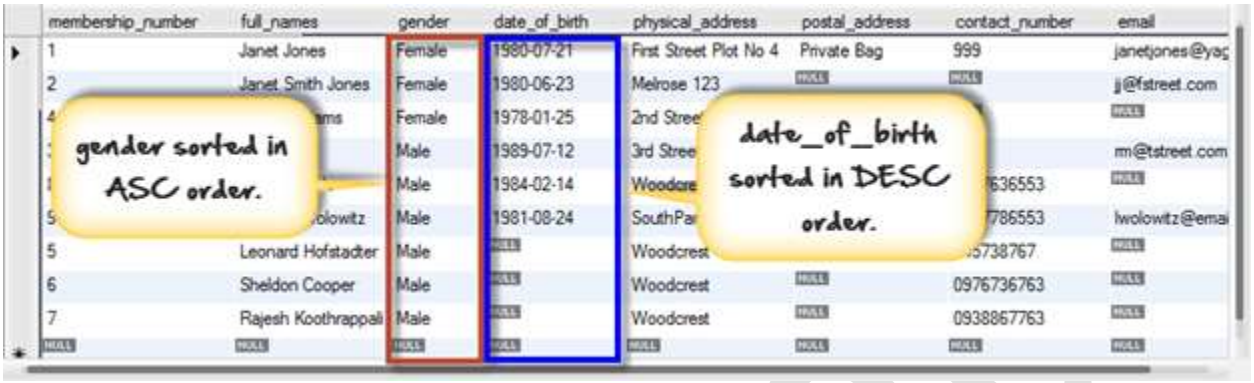
Suppose we want to get a list that sorts the query result using the gender field, we would use the script shown below:

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

"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;
```




The gender column was sorted in ascending order y 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.

## Note:

- Sorting query results 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 in ASC.

## MySQL GROUP BY and HAVING Clause

### GROUP BY Clause

The GROUP BY clause is a SQL command that is used to **group row 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.

```
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

Let's execute a simple query that returns all the gender entries from the member's table.

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

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

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

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

```
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_id', 'year_released';
```

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

NOTE: If the category id is the same but the year released is different, then a row is treated as 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:

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

| gender | COUNT('membership_number') |
|---|---|
| Female | 3 |
| Male | 5 |

## Restricting query results using the HAVING clause

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

Suppose we want to know all the release 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;
```

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

NOTE:

- 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 result returned by the GROUP BY clause.

## MySQL Wildcards: Like, NOT Like, Escape, (%), (/)

### MySQL Wildcards

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

### Why use Wildcards?

If you are familiar with using SQL, you might think you can search for any complex data using SELECT and WHERE clause. They why use Wildcards?

Before we answer that question, let's look at an example. Suppose that the marketing department of MyFlixDB 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 following SELECT statement together with the WHERE clause to get the desired information.

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

As you can see 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';
```

### Types of Wildcards

### % 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…** WHERE fieldname LIKE 'xxx%';
- **"WHERE"** is the keyword 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 single character or more and "%" matches any number of characters starting from zero (0).

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".

The SQL query will be:

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

| 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 keyword "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';
```

| movie_id | title | director | year_released | category_id |
|---|---|---|---|---|
| 7 | Davinci Code | NULL | NULL | 6 |

Similarly, now shift the percentage wildcard to the end of the specified pattern to be matched.

The modified query will be:

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

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

### _ underscore wildcard

The underscore wildcard is used to **match exactly one character**.

Example:

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

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

| 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.

### Example:

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. The query will be:

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

| movie_id | title | director | year_released | category_id |
|---|---|---|---|---|
| 1 | Pirates of the Caribbean 4 | Rob Marshall | 2011 | 1 |
| 4 | Code Name Black | Edgar Jimz | 2010 | NULL |
| 8 | Underworld-Awakening | Michael Eel | 2012 | 6 |

## 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.

Example:

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 that 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 '=';
```

## NOTE

- 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.
- Th percentage wildcard is used to match any number of characters starting from zero and more.
- The underscore wildcard is used to match exactly one character.

## MySQL Regular Expressions (REGEXP)

### Regular Expression

Regular expression help **data matching complex criteria**.

We know we can also do the same thing with wildcards. But as compared to wildcards, regular expressions allow us to search data matching even more complex criterion.

```
SELECT statements... WHERE fieldname REGEXP 'pattern';
```

Here,

- **"SELECT statement..."** is the standard SELECT statement.
- **"WHERE fieldname"** is the name of the column on which the regular expression is to be performed on.
- **"REGEXP 'pattern'"** REGEXP is the regular expression operator and 'pattern' represents the pattern to be matched by REGEXP. **RLIKE** is the **synonym for REGEXP** and achieves the same results as REGEXP. To avoid confusing it with the LIKE operator, it **better to use REGEXP instead**.

**Example**:

```
SELECT * FROM 'movies' WHERE 'title' REGEXP 'code';
```

The above query searches for all the movie titles that have the word code in them. It does not matter whether the "code" is at the beginning, middle or end of the title. As long as it is contained in the title then it will be considered.

Suppose that we want to search for movies that start with a, b, c or d, followed by any number of other characters. For that we can use the regular expression together with the metacharacters to achieve our desired results.

```
SELECT * FROM 'movies' WHERE 'title' REGEXP '^[abcd]';
```

| movie_id | title | director | year_released | category_id |
|---|---|---|---|---|
| 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 |

Here,

'^[abcd]' the caret (^) means that the pattern match should be applied at the beginning and the chalist [abcd] means that only movie titles that start with [a, b, c or d] are returned to our result set.

Let's modify our above script and use the NOT charlist and see what results we will get after executing our query.

```
SELECT * FROM 'movies' WHERE 'title' REGEXP '^[^abcd]';
```

| movie_id | title | director | year_released | category_id |
|---|---|---|---|---|
| 1 | Pirates of the Caribbean 4 | Rob Marshall | 2011 | 1 |
| 2 | Forgetting Sarah Marshal | Nicholas Stoller | 2008 | 2 |
| 3 | X-Men | | 2008 | |
| 9 | Honey mooners | John Schultz | 2005 | 8 |
| 16 | 67% Guilty | | 2012 | |
| 17 | The Great Dictator | Chalie Chaplie | 1920 | 7 |
| 18 | sample movie | Anonymous | | 8 |
| 19 | movie 3 | John Brown | 1920 | 8 |

Here,

'^[^abcd]' the caret (^) means that the pattern match should be applied at the beginning and the charlist [^abcd] means that the movie titles starting with any of the enclosed characters is excluded from the result set.

NOTE

- Regular expressions provide a powerful and flexible pattern match that can help us implement power search utilities for our database systems.
- REGEXP is the operator used when performing regular expression pattern matches. RLIKE is the synonym.
- Regular expressions support a number of metacharacters which allow for more flexibility and control when performing pattern matches.
- The back slash (\) is used as an escape character in regular expressions. It's only considered in the pattern match if double backslash has used.
- Regular expressions are not case sensitive.

## Regular Expression Metacharacters

| Char | Description | Example |
|---|---|---|
| * | The **asterisk (*)** metacharacter is used to match zero (0) or more instances of the strings preceding it. | SELECT * FROM 'movies' WHERE 'title' REGEXP 'da*'; will give all movies containing characters "da". |
| + | The **plus (+)** metacharacter is used to match one or more instances of strings preceding it. | SELECT * FROM 'movies' WHERE 'title' REGEXP 'mon+'; will give all movies containing characters "mon". |
| ? | The **question (?)** metacharacter is used to match zero (0) or one instances of the strings preceding it. | SELECT * FROM 'categories' WHERE 'category_name' REGEXP 'com?'; will give all the categories containing string com. |
| . | The **dot(.)** metacharacter is used to match any single character in exception of a new line. | SELECT * FROM 'movies' WHERE 'year_released' REGEXP '200.'; will give all the movies released in the years starting with characters "200" followed by any single character. |
| [abc] | The **charlist [abc]** is used to match any of the enclosed characters. | SELECT * FROM 'movies' WHERE 'title' REGEXP '[vwxyz]'; will give all the movies containing any single character in 'vwxyz'. |
| [^abc] | The **charlist [^abc]** is used to match any characters excluding the ones enclosed. | SELECT * FROM 'movies' WHERE 'title' REGEXP '[^vwxyz]'; will give all the movies containing characters other than the ones in "vwxyz". |
| [A - Z] | The **[A-Z]** is used to match any upper-case letter. | SELECT * FROM 'members' WHERE 'postal_address' REGEXP ['A-Z']; will give all the members that have postal address containing any character from A to Z. |

| | | |
|---|---|---|
| [a-z] | The **[a-z]** is used to match any lower-case letter. | SELECT * FROM 'members' WHERE 'postal_address' REGEXP 'a-z'; will give all the members that have postal address containing any character from a to z. |
| [0-9] | The **[0-9]** is used to match any digit from 0 through to 9. | SELECT * FROM 'members' WHERE 'contact_number' REGEXP '[0-9]'; will give all the members have submitted contact numbers containing characters 0 to 9. |
| ^ | The **caret (^)** is used to start the match at beginning. | SELECT * FROM 'movies' WHERE 'title' REGEXP '^[cd]'; gives all the movies with the title starting with any of the characters in "cd". |
| \| | The **vertical br (\|)** is used to isolate alternatives. | SELECT * FROM 'movies' WHERE 'tile' REGEXP '^[cd] \| ^[u]'; gives all the movies with the title starting with any of the characters in "cd" or "u". |
| [[:<:]] | The [[:<:]] matches the beginning of words. | SELECT * FROM 'movies' WHERE 'title' REGEXP '[[:<:]] for'; gives all the movies with titles starting with the characters "for". |
| [[:>:]] | The [[:>:]] | SELECT * FROM 'movies' WHERE 'title' REGEXP 'ack [[:>:]]'; gives all the movies with titles ending with the characters "ack". |
| [:class:] | The [:class:] matches a character class i.e. [: alpha:] to match letters, [: space:] to match white space, [: punct:] is match punctuations and [: upper:] for upper class letters. | SELECT * FROM 'movies' WHERE 'title' REGEXP '[: alpha:]'; gives all the movies with titles contain letters only. |

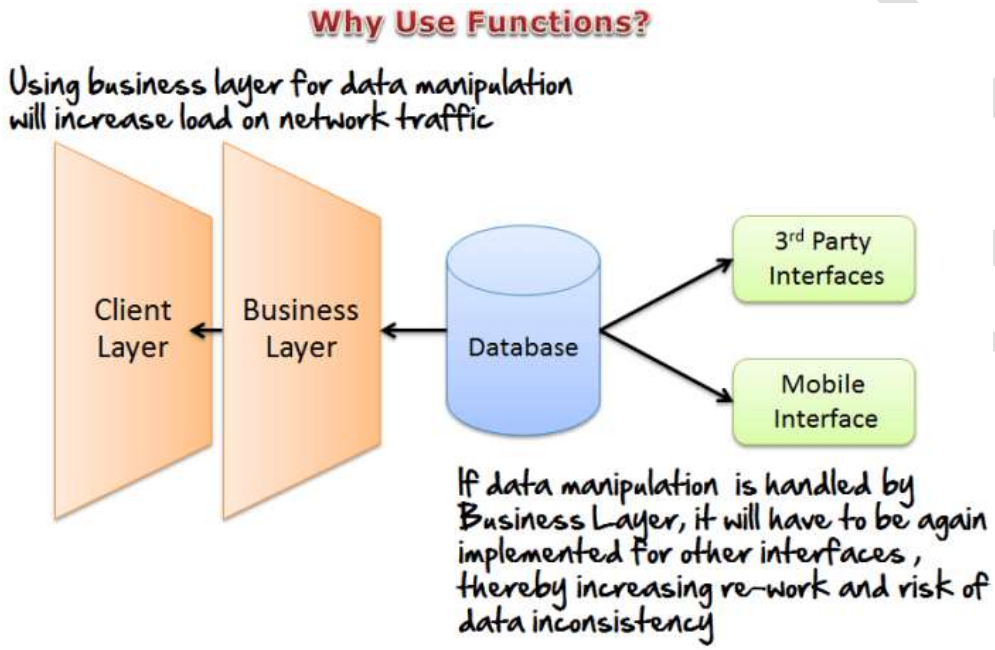
## MySQL Functions – String, Numeric, User-Defined, Stored

### 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 off code that perform some operations and then return a result.

### Why use functions?




For **Example**: By default, MySQL saves date datatypes in the format "YYYY-MM-DD". Suppose we have built an application and our users what 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.

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 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 for using Functions is that**, MySQL functions is the fact that it can help reducing network traffic in client/server applications**.

## Types of Functions

### 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:

- **String Functions** - operate on string datatypes
- **Numeric Functions** – operate on numeric datatypes
- **Date Functions** – operate on date datatypes
- **Aggregate Functions** – operate on all of the above datatypes and produce summarized result sets.

### String Functions

String functions operate on **string data types**.

Example: 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"** functions to do that. It takes a string as parameter and converts all the letters to upper-case.

The query will be:

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

| 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 Caribbean 4 | PIRATES OF THE CARIBEAN 4 |
| 18 | sample movie | SAMPLE MOVIE |
| 17 | The Great Dictator | THE GREAT DICTATOR |
| 3 | X-Men | X-MEN |

## Numeric Functions

These functions operate on **numeric data types**. We can perform mathematic computations on numeric data in the SQL statements.

### Arithmetic Operators

#### Integer Division (DIV)

```
SELECT 23 DIV 6;
```

#### Division Operator (/)

```
SELECT 23 / 6;
```

#### Subtraction Operator (-)

```
SELECT 23 – 6;
```

#### Addition Operator (+)

```
SELECT 23 + 6;
```

#### Multiplication Operator (*)

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

#### Modulo Operator (%)

The modulo operator divides N by M and gives us the remainder.

```
SELECT 23 % 6; OR SELECT 23 MOD 6;
```

#### Floor

This function removes decimals places from a number and round it to the nearest lower number.

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

#### Round

This function rounds a number with decimal places to the nearest whole number.

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

### Rand

This function is used to generate a random number, its value changes every time that the function is called.

```
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 syntax will be:

```
CREATE FUNCTION sf_name ([parameters])
        RETURN data types
        DETERMINISTIC
        STATEMENTS
```

Here,

- **"CREATE FUNCTION sf_name ([parameters])"** is mandatory and tells MySQL server to create a function named 'sf_name' with optional parameters defined in the parenthesis.
- **"RETURN data types"** 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.

Example:

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 query will be:

```
1.  DELIMITER |
2.  CREATE FUNCTION sf_past_movie_return_date (return_date DATE)
3.   RETURNS VARCHAR (3)
4.    DETERMINISTIC
5.     BEGIN
6.      DECLARE sf_value VARCHAR (3);
7.        IF curdate () > return_date
8.           THEN SET sf_value = 'Yes';
9.        ELSEIF curdate () <= return_date
10.          THEN SET sf_value = 'No';
11.       END IF;
12.     RETURN sf_value;
13.    END|
```

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

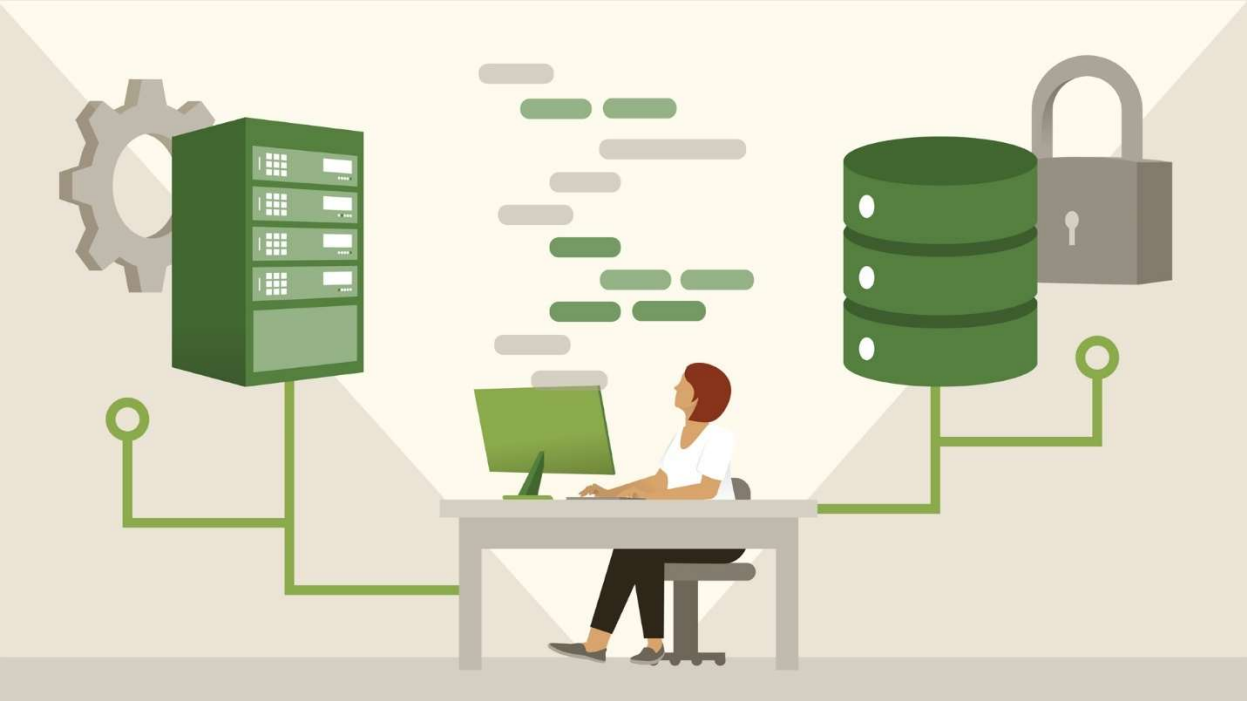
Let's now test our stored function:

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

| 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.



**NOTE:**

- 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: SUM, AVG, MAX, MIN, COUNT, DISTINCT

### Aggregate Function

Aggregate functions are all about:

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

There are 5 aggregate function according to ISO standard:

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.

### 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 null values before working on 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.

Example

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 |

Suppose 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
```

## DISTINST Keyword

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

```
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';
```

| movie_id |
| --- |
| 1 |
| 2 |
| 3 |

## MIN Function

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

Example: suppose we want to know the year in which the oldest movie in our library released, we can use the MIN function to get the desired information.

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

| MIN('year_released') |
| --- |
| 2005 |

## MAX Function

The **MAX** function is the opposite of the MIN function. It returns the largest value from the specified table field.

Example: suppose we want to get the year that the latest movie in our databases was released. We can easily use the MAX function to achieve that.

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

| 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.

**Example**: 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 |

So, the query to get all payments made and sums them up to return a single result is:

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

| SUM('amount_paid') |
| --- |
| 10500 |

## AVG Function

MySQL **AVG** function **returns the average of the values in a specified column**. It also only on **numeric data types**.

**Example**: Suppose we want to find the average amount pai. The query will be:

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

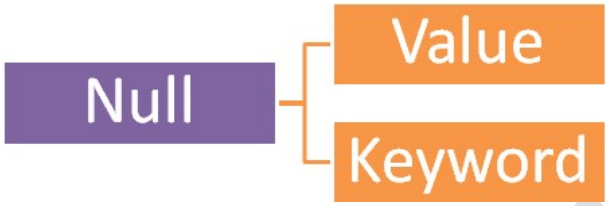
| AVG('amount_paid') |
| --- |
| 3500 |

NOTE:

- MySQL supports all the five (5) ISO standard aggregate functions COUNT, SUM, AVG, MIN, and MAX.
- SUM & 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.

## MySQL – NULL & NOT NULL

In SQL, NULL I both a **value** as well as a **keyword**.




### Null as a Value

NULL is simply a place holder for data that does not exist.

When performing insert operations on tables, there will be time when some field values will not be available.

In order to meet the requirement of true relational database management systems, MySQL uses NULL as the place holder for the values that have not been submitted or assigned.

| 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 |

### NOTE

- **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, 56 + NULL = NULL.
- All **aggregate functions affect only rows that do not have NULL values**.

### Why uses 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 values returns null results. In order to avoid such situations from happening, we can employ the use of the NOT NULL clause to limit the result on which our data operates.

## NOT NULL Values

Suppose 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.

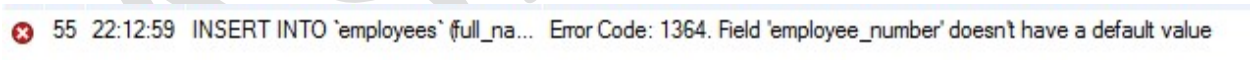
The example shown below creates a new table that contains employee's data. The employee number should always be supplied:

```
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 query in MySQL Workbench gives the following error,




## 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.

```
'column_name' IS NULL

'column_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.

### Example:

Suppose we need details of members whose contact number is not null. Then the query will be:

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

Suppose we want member records where contact number is null. Then the query will be:

```
SELECT * FROM 'members' WHERE contact_numbers IS NULL;
```

## Comparing null values

**Three-value logic:** performing Boolean operations on condition 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 return **"Unknown" (NULL).**

### Example:

Suppose compare number 5 with 5.

```
SELECT 5 = 5;
```

The query result 1 which means TRUE.

Let's do the same operation with NULL.

```
SELECT NULL = NULL;
```

Result: NULL = NULL is NULL

Suppose,

```
SELECT 5 > 5;
```

The query result is 0 which means FALSE.

Let's use IS NULL keyword,

```
SELECT 5 IS NULL;
```

The query result is 0 which is FALSE.

```
SELECT NULL IS NULL;
```

The query result is 1 which is TRUE.

## NOTE

- 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

### 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?

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.

### Syntax

```
1.  CREATE TABLE `categories` (
2.    `category_id` int (11) AUTO_INCREMENT,
3.    `category_name` varchar (150) DEFAULT NULLS,
4.    `remarks` varchar (500) DEFAULT NULLS,
5.    PRIMARY KEY (`category_id`)
```

```
6.  );
```

Notice the "AUTO_INCREMENT" on the category_id field. This cause 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.

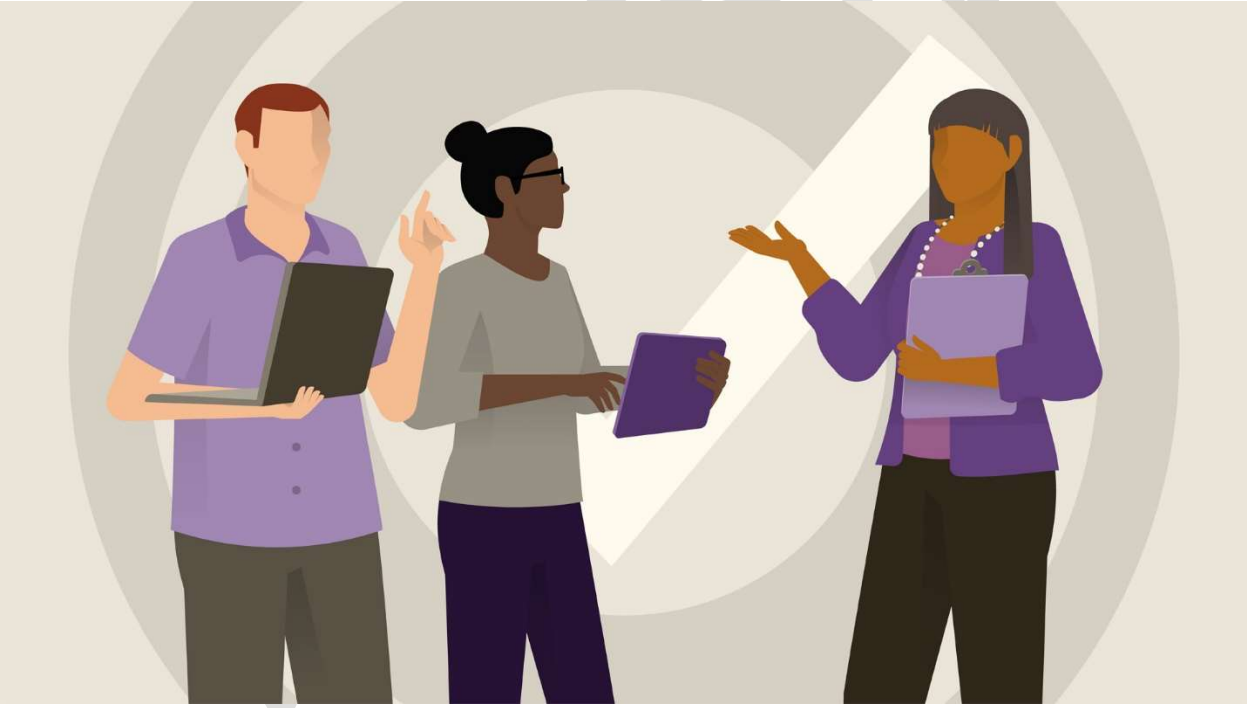
By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

Suppose we want to insert a new category into table:

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

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.

```
SELECT LAST_INSERT_ID ();
```



## NOTE

- Auto increment attribute when specified on a column with a numeric data type, generates numbers sequentially whenever a new row is added into the database.
- The Auto increment is commonly used to generate primary keys.
- The defined datatype 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 increment id is not re-used. MySQL continues generating new numbers sequentially.
- By default, the string value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.
- To let AUTO_INCREMENT sequences start with another value, use AUTO_INCREMENT = 10.

## MySQL – ALTER, DROP, RENAME, MODIFY

### ALTER COMMAND

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

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

```
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'.

### Example:

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.

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 member's table. We can use the ALTER command to do that.

| 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 | |

To add a new field the query will be:

```
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.

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

| 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 | |
| credit_card_number | varchar (25) | YES | | | |

## 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.**

Example:

Suppose the online billing functionality will take some time and we want to DROP the credit card column. The query will be:

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

Executing the above query drops the column credit_card_number from the member's table.

## DROP TABLE

The syntax to DROP a table from Database is:

```
DROP TABLE 'sample_table';

Example: DROP TABLE 'categories_archive';
```

## 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:

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

Example: RENAME TABLE 'movierentals' TO 'movie_rentals';
```

## CHANGE KEYWORD

Change keyword allows you to,

1. Change name of the column
2. Change column data types
3. Change column constraints

Example:

The full names field in the members table is of varchar data type and has a width of 150.

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

Then the query will be:

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

## MODIFY KEYWORD

The MODIFY keyword allows you to,

1. Modify column data types
2. Modify column constraints

Example:

In the CHANGE example, we had to change the field name as well other details. **Omitting 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 query below changes the width of "fullname" field from 250 to 50.

```
ALTER TABLE 'members' MODIFY 'fullname' char (50) NOT 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 query below adds "date_of_registration" just after the date of birth in the member's table.

```
ALTER TABLE 'members' ADD 'date_of_registration' date NULL AFTER 'date_of_birth';
```

## NOTE

- 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 database rom MySQL 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 **ALTER** keyword is used to specify position of a column in a table.

## MySQL – LIMIT & KEYWORD

### LIMIT Keyword

The LIMIY 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.

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

Here,

- **"SELECT {fieldnames(s) | *} FROM tableNames(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.

Example:

Suppose that we want to get a list of the first 10 registered members from the MyFlixDB database.

The query will be:

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

## OFF SET

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

Suppose we want to get a limited number of members starting from the middle of the rows, we can use the LIMIT keyword with the offset value to achieve that.

The script shown below gets data starting the second row and limits the result to 2.

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

**NOTE**: 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 has 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.

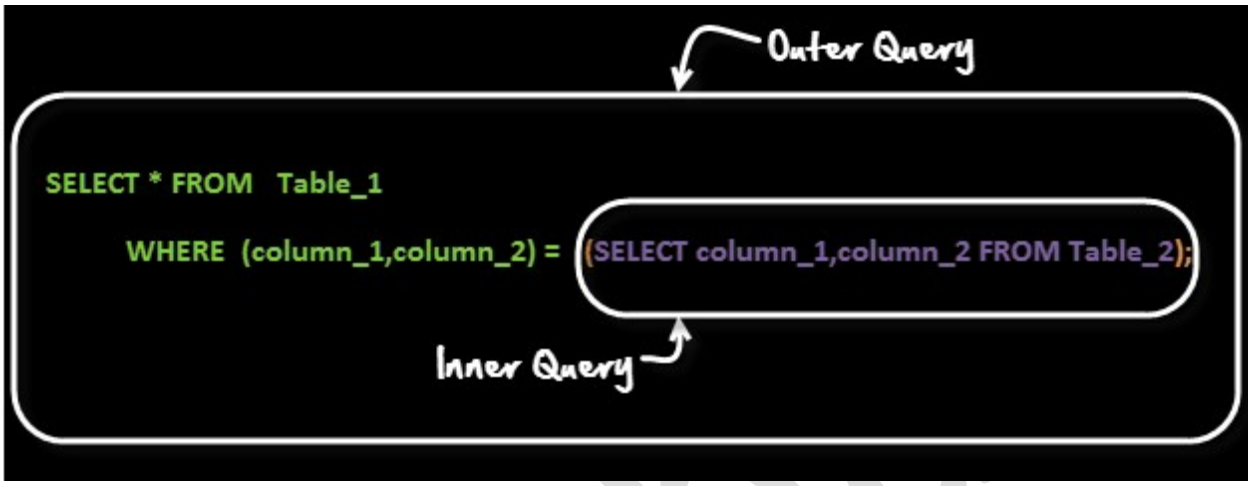
## NOTE

- The LIMIT keyword 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 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

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.

Syntax:



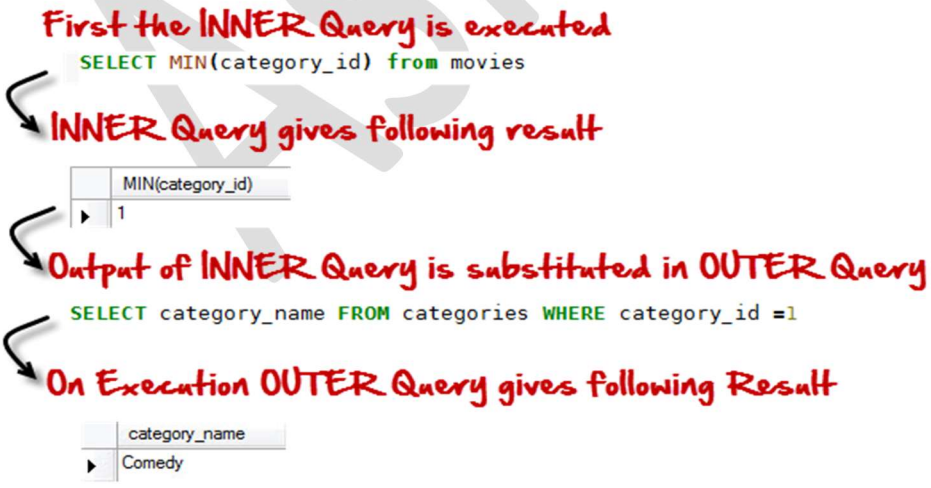

### Example:

Suppose, 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.

The query will be:

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

Let's see how this works:



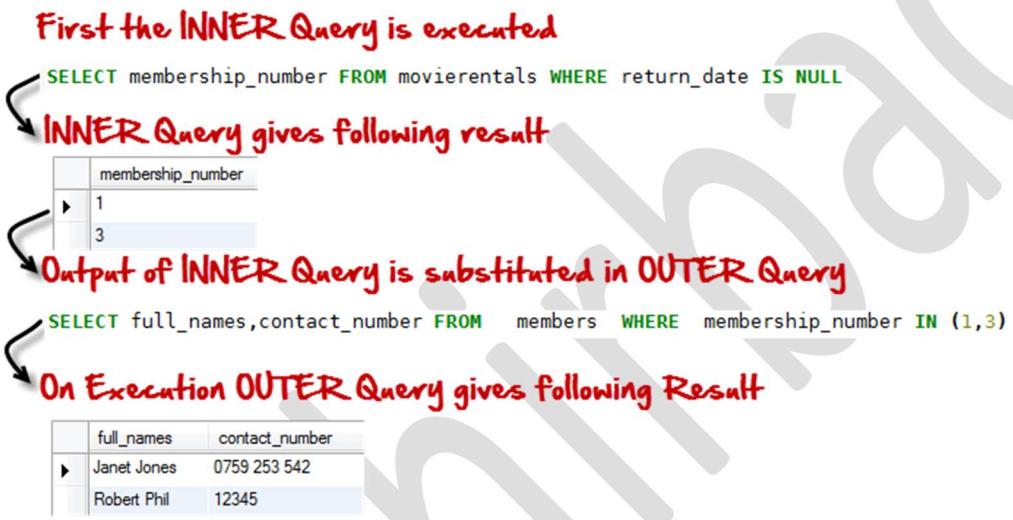
The above is a form of **Row Sub-Query**. In such sub-queries the, inner query can give only ONE result. The permissible operators when work with row subqueries are [=, >=, <=, !=].

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 remainder.

The query will be:

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

Let's see how this query works:




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

Suppose. The management wants to reward the highest paying member. Then the query will be:

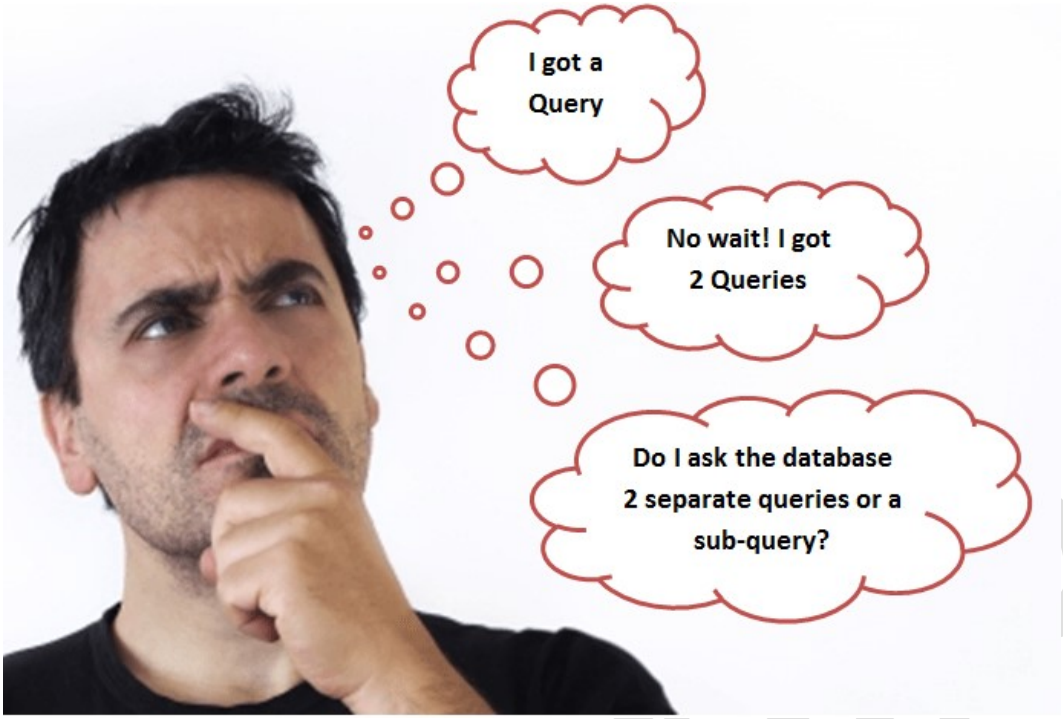
```
SELECT full_names FROM members WHERE membership_number = (SELECT membership_number FROM payments
WHERE amount_paid = (SELECT MAX (amount_paid) FROM payments));
```

The above query is an example of Triple **query**!

## 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.




## NOTE

- 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 subqueries 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 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 JOINs – INNER, OUTER, LEFT, RIGHT, CROSS

### JOINS

Joins help **retrieving** data from two or more database tables.

The tables are mutually related using primary and foreign keys.

Example:

| id | first_name | last_name | movie_id |
|---|---|---|---|
| 1 | Adam | Smith | 1 |
| 2 | Ravi | Kumar | 2 |
| 3 | Susan | Davidson | 5 |
| 4 | Jenny | Adrianna | 8 |
| 6 | Lee | Pong | 10 |

| id | title | category |
|---|---|---|
| 1 | ASSASSIN'S CREED: EMBERS | Animations |
| 2 | Real Steel (2012) | Animations |
| 3 | Alvin and the Chipmunks | Animations |
| 4 | The Adventures of Tin Tins | Animations |
| 5 | Safe (2012) | Action |
| 6 | Safe House (2012) | Action |
| 7 | GIA | 18+ |
| 8 | Deadline 2009 | 18+ |
| 9 | The Dirty Picture | 18+ |
| 10 | Marley and me | Romance |

## Types of JOIN

### CROSS JOIN

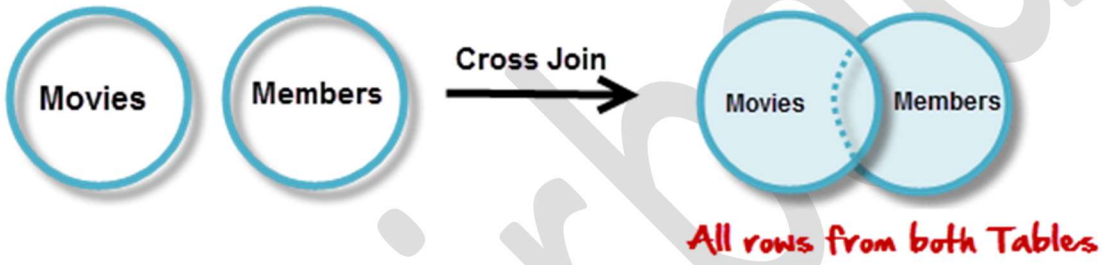
CROSS JOIN is a simplest form of joins which matches each row from one database table to all rows of another.

In other words, it gives us combinations of each row of first table with all records in second table.

Suppose we want to get all member records against all the movie records, we can use the script shown below to get our desired results.

Then the query will be:

```
SELECT * FROM 'movies' CROSS JOIN 'members'
```




### INNER JOIN

The **INNER JOIN** keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e. value of the common field will be same.

We can also write **JOIN** instead of INNER JOIN. JOIN is same as INNER JOIN.

Syntax:

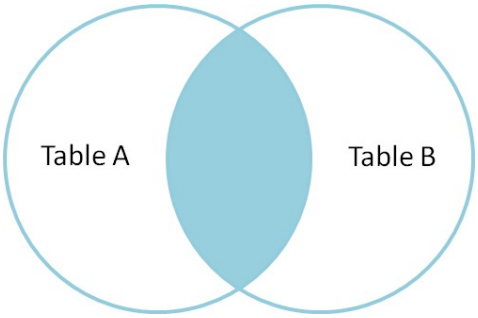
```
SELECT table1.column1, table1.column2, table2.column1, ..... FROM table1 INNER JOIN table2.matching_column = table2.matching_column;
```

Where,

table1 = first table

table2 = second table

matching_column = column common to both the tables




Consider the two tables below:

### Student

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | HARSH | DELHI | xxxxxxxxxx | 18 |
| 2 | PRATIK | BIHAR | xxxxxxxxxx | 19 |
| 3 | RIYANKA | SILIGURI | xxxxxxxxxx | 20 |
| 4 | DEEP | RAMNAGAR | xxxxxxxxxx | 18 |
| 5 | SAPTARHI | KOLKATA | xxxxxxxxxx | 19 |
| 6 | DHANRAJ | BARABAJAR | xxxxxxxxxx | 20 |
| 7 | ROHIT | BALURGHAT | xxxxxxxxxx | 18 |
| 8 | NIRAJ | ALIPUR | xxxxxxxxxx | 19 |

### StudentCourse

| COURSE_ID | ROLL_NO |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 3 | 4 |
| 1 | 5 |
| 4 | 9 |
| 5 | 10 |
| 4 | 11 |

This query will show the names and age of students enrolled in different courses:

```
SELECT StudentCourse. COURSE_ID, Student.NAME, Student. AGE FROM Student INNER JOIN StudentCourse ON Student.ROLL_NO = StudentCourse. ROLLNO;
```

Output:

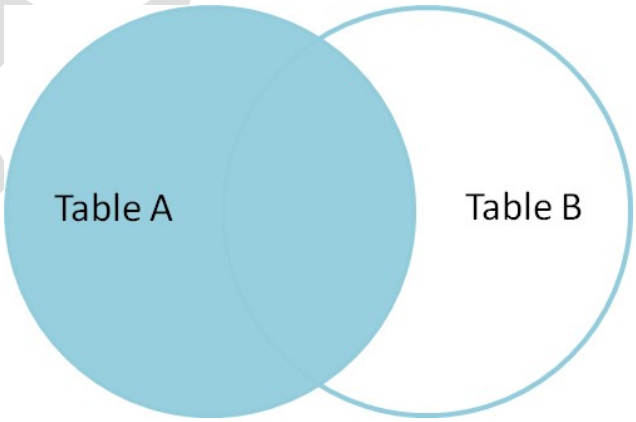
| COURSE_ID | NAME | Age |
|---|---|---|
| 1 | HARSH | 18 |
| 2 | PRATIK | 19 |
| 2 | RIYANKA | 20 |
| 3 | DEEP | 18 |
| 1 | SAPTARHI | 19 |

## LEFT JOIN

This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain NULL.

LEFT JOIN is also known as **LEFT OUTER JOIN**.

```
SELECT table1.column1, table1.column2, table2.column1, ..... FROM table1 LEFT JOIN table2 ON table1.matching _column = table2.matching _column;
```



Example:

```
SELECT Student.NAME, StudentCourse.COURSE_ID FROM Student LEFT JOIN StudentCourse ON
StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |

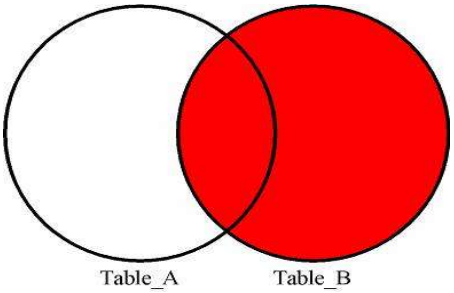
## RIGHT JOIN

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain NULL.

RIGHT JOIN is also known as **RIGHT OUTER JOIN**.

Syntax:

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



Example:

```
SELECT Student.NAME, StudentCourse.COURSE_ID FROM Student RIGHT JOIN StudentCourse ON
StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| *NULL* | 4 |
| *NULL* | 5 |
| *NULL* | 4 |

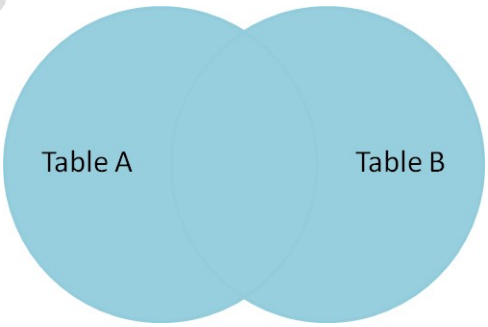
## FULL JOIN

FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN, the result-set will contain all the rows from both the tables.

The rows for which there is no matching, the result-set will contain NULL values.

Syntax:

```
SELECT table1.column1, table1.column2, table2.coumn1, ..... FROM table1 FULL JOIN table2 ON
table1.matching_column = table2.matching_column;
```



Example:

```
SELECT Student.NAME, StudentCourse.COURSE_ID FROM Student FULL JOIN StudentCourse ON
StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |
| *NULL* | 9 |
| *NULL* | 10 |
| *NULL* | 11 |

## MySQL UNION

### What is UNION?

Unions combine the results from multiple SELECT queries into a consolidated result set.

The only requirements for this to work is that the number of columns should be the same from all the SELECT queries which needs to be combined.

Example:

Suppose we have two tables,

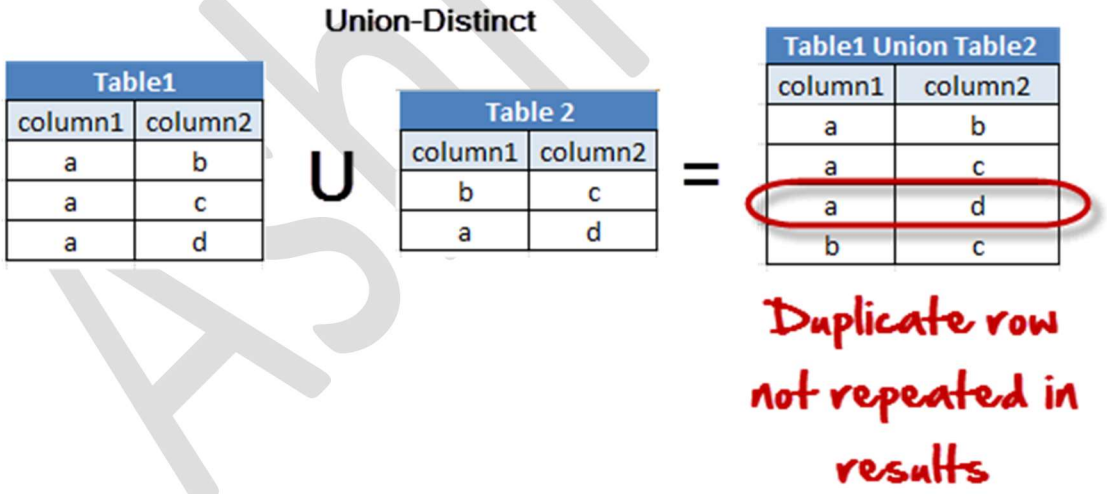
| Table1 | |
|---|---|
| column1 | column2 |
| a | b |
| a | c |
| a | d |

| Table 2 | |
|---|---|
| column1 | column2 |
| b | c |
| a | d |

Let's now create a UNION query to combine both tables using **DISTINCT**.

```
SELECT column1, column2 FROM 'table1' UNION DISTINCT SELECT column1, column2 FROM 'table2';
```

Here, duplicate rows are removed and only unique rows are returned.

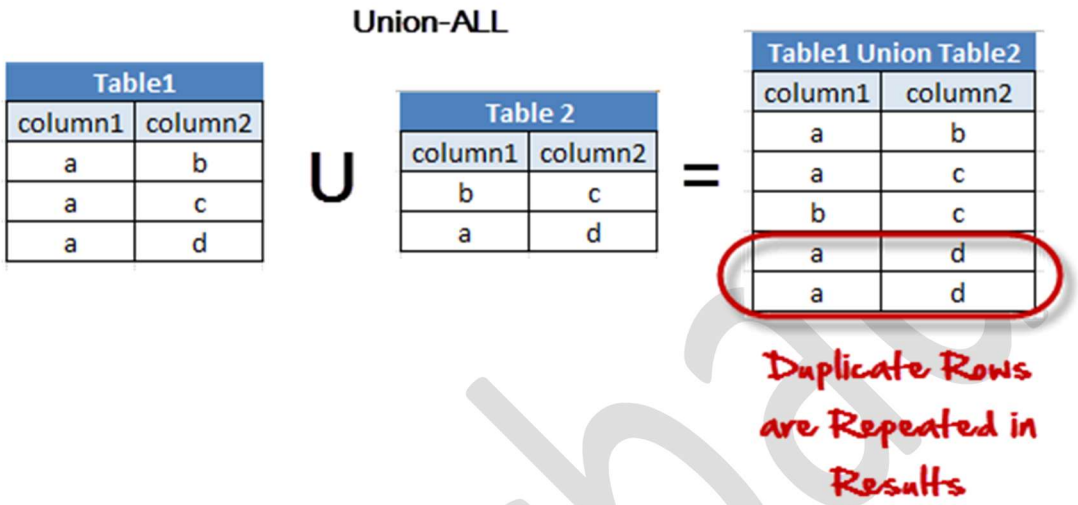



NOTE: MySQL uses the DISTINCT clause as default when executing UNION queries if nothing is specified.

Let's now create a UNION query to combine both tables using **ALL**.

```
SELECT 'column1', 'column2' FROM 'table1' UNION ALL SELECT 'column1', 'column2' FROM 'table2';
```

Here duplicate rows are included and since we use ALL.




## Why use UNION?

Suppose there is a flaw in your database design and you are using two different tables meant for the same purpose. You want to consolidate these two tables into one while omitting any duplicate records from creeping into the new table. You can use UNION in such cases.

## NOTE

- The UNION command is used to combine more than one SELECT query results into a single query contain rows from all the select queries.
- The number of columns and data types in the SELECT statements must be the same in order for the UNION command to work.
- The DISTINCT clause is used to eliminate duplicate values from the UNION query result set. MySQL uses the DISTINCT clause as the default when executing UNION queries if nothing is specified.
- The ALL clause is used to return all even the duplicate rows in the UNION query.

## MySQL VIEWs – CREATE, JOIN & DROP

**VIEWS** are virtual tables that do not store any data of their own but display data stored in other tables. In other words, VIEWS are nothing but SQL Queries. A view can contain all or a few rows from a table. A MySQL view can show data from one table or many tables.

### Views syntax

Let's now look at the basic syntax used to create a view in MySQL.

```
CREATE VIEW `view_name` AS SELECT statement;
```

### WHERE

- **"CREATE VIEW `view_name`"** tells MySQL server to create a view object in the database named `view_name`
- **"AS SELECT statement"** is the SQL statements to be packed in the views. It can be a SELECT statement can contain data from one table or multiple tables.

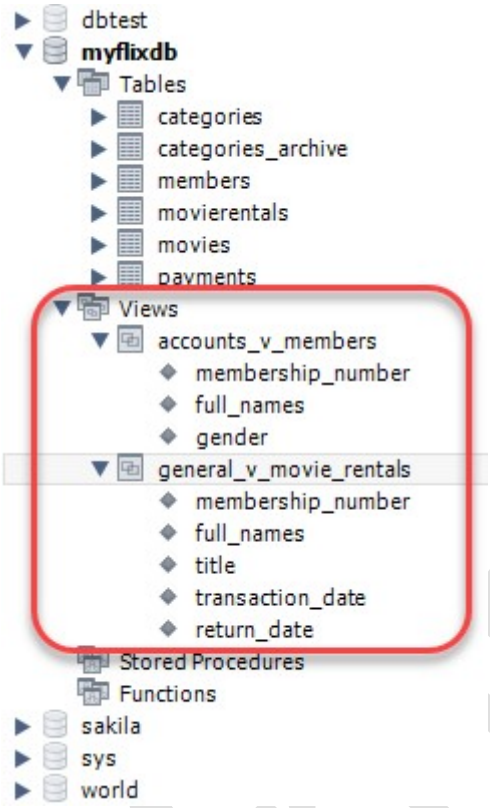
Example:

Let's now create our first view using the "MyFlixDB" we will create a simple view that restricts the columns seen in the member's table.

Suppose authorization requirements state that the accounts department can only see member's number, name and gender from the member's table. To achieve this, you can create a VIEW -

```
CREATE VIEW 'accounts_v_members' AS SELECT 'membership_number', 'full_names', 'gender' FROM 'members';
```

Executing the above query in MySQL workbench against the MyFlixDB and expanding the views node in the database explorer gives us the following results.




Note the accounts_v members object is now visible in the database views objects. Let's now execute a SELECT statement that selects all the fields from the view.

```
SELECT * FROM 'accounts_v_members';
```

Output:

Only the authorized columns for accounts department have been returned. Other details found in the members table have been hidden.

If we want to see the SQL statements that make up a particular view, we can use the script below:

```
SHOW CREATE VIEW 'accounts_v_members';
```

Executing the above script gives you the view name and the SQL SELECT statements used to create the view.

## Why use views?

You may want to use views primarily for following 3 reasons:

- Ultimately, you will use your SQL knowledge, to create applications, which will use a database for data requirements. It's recommended that you use VIEWS of the original table structure in your application instead of using the tables themselves. This ensures that when you refactor your DB, your legacy code will see the original schema via the view without breaking the application.
- VIEWS increase re-usability. You will not have to create complex queries involving joins repeatedly. All the complexity is converted into a single line of query use VIEWS. Such condensed code will be easier to integrate in your application. This will eliminate chances of typos and your code will be more readable.
- VIEWS help in data security. You can use views to show only authorized information to users and hide sensitive data like credit card numbers.

## NOTE

- Views are virtual tables; they do not contain the data that is returned. The data is stored in the tables referenced in the SELECT statement.
- Views improve security of the database by showing only intended data to authorized users. They hide sensitive data.
- Views make life easy as you do not have write complex queries time and again.
- It's possible to use INSERT, UPDATE and DELETE on a VIEW. These operations will change the underlying tables of the VIEW. The only consideration is that VIEW should contain all NOT NULL columns of the tables it references. Ideally, you should not use VIEWS for updating.
- The DROP command is used to delete a view from the database that is no longer required.

Syntax:

```
DROP VIEW 'general_v_movie_rentals';
```

## MySQL Index – Create, Add, Drop

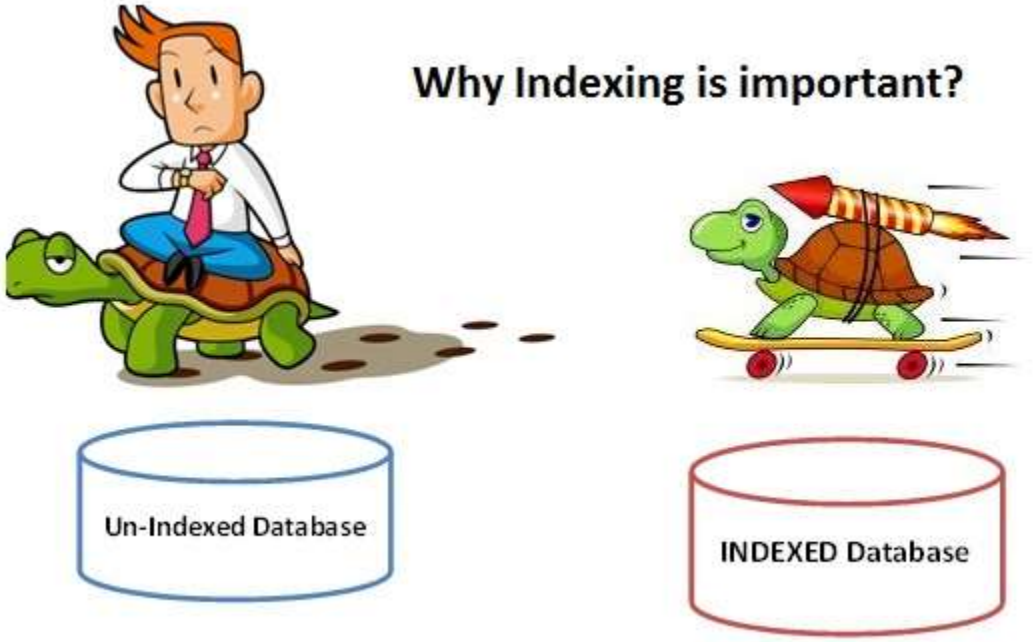
### What is an Index?

Indexes in MySQL sort data in an organized sequential way. They are created on the column(s) that will be used to filter the data. 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.

Using an index 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.

### What uses an 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.




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. Hence, Indexes are used that sort data to make it easier to search.

Ashirbad Swain

**Syntax: Create Index**

Indexes can be defined in 2 ways

1. At the time of table creation
2. After table has been created

**Example:**

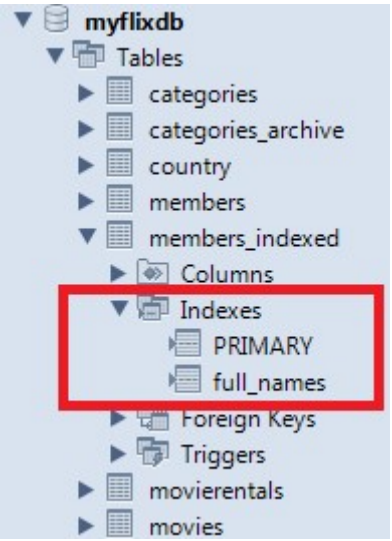
For our MyFlixDB we expect lots of searches to the database on full name.

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 index's 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 results on index created.

**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.

**Syntax: Drop index**

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.

**NOTE**

- 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.