**CREATE DB**

CREATE DATABASE movies;

CREATE DATABASE IF NOT EXISTS movies;

**CREATE TABLE**

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

CREATE TABLE IF NOT EXISTS ` movies`.`Members` (

`membership\_number` INT AUTOINCREMENT,

`full\_names` VARCHAR(150) NOT NULL,

`gender` VARCHAR(6),

`date\_of\_birth` DATE,

`physical\_address` VARCHAR(255),

`postal\_address` VARCHAR(255),

`contact\_number` VARCHAR(75),

`email` VARCHAR(255),

PRIMARY KEY (`membership\_number`)

);

**DATA TYPES**

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

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

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

**Numeric Data types**

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

|  |  |
| --- | --- |
| TINYINT( ) | -128 to 127 normal 0 to 255 UNSIGNED. |
| SMALLINT( ) | -32768 to 32767 normal 0 to 65535 UNSIGNED. |
| MEDIUMINT( ) | -8388608 to 8388607 normal 0 to 16777215 UNSIGNED. |
| INT( ) | -2147483648 to 2147483647 normal 0 to 4294967295 UNSIGNED. |
| BIGINT( ) | -9223372036854775808 to 9223372036854775807 normal 0 to 18446744073709551615 UNSIGNED. |
| FLOAT | A small approximate number with a floating decimal point. |
| DOUBLE( , ) | A large number with a floating decimal point. |
| DECIMAL( , ) | A DOUBLE stored as a string , allowing for a fixed decimal point. Choice for storing currency values. |

**Text Data Types**

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

|  |  |
| --- | --- |
| CHAR( ) | A fixed section from 0 to 255 characters long. |
| VARCHAR( ) | A variable section from 0 to 255 characters long. |
| TINYTEXT | A string with a maximum length of 255 characters. |
| TEXT | A string with a maximum length of 65535 characters. |
| BLOB | A string with a maximum length of 65535 characters. |
| MEDIUMTEXT | A string with a maximum length of 16777215 characters. |
| MEDIUMBLOB | A string with a maximum length of 16777215 characters. |
| LONGTEXT | A string with a maximum length of 4294967295 characters. |
| LONGBLOB | A string with a maximum length of 4294967295 characters. |

**Date / Time**

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

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

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

CREATE TABLE`all\_data\_types` (

`varchar` VARCHAR( 20 ) ,

`tinyint` TINYINT ,

`text` TEXT ,

`date` DATE ,

`smallint` SMALLINT ,

`mediumint` MEDIUMINT ,

`int` INT ,

`bigint` BIGINT ,

`float` FLOAT( 10, 2 ) ,

`double` DOUBLE ,

`decimal` DECIMAL( 10, 2 ) ,

`datetime` DATETIME ,

`timestamp` TIMESTAMP ,

`time` TIME ,

`year` YEAR ,

`char` CHAR( 10 ) ,

`tinyblob` TINYBLOB ,

`tinytext` TINYTEXT ,

`blob` BLOB ,

`mediumblob` MEDIUMBLOB ,

`mediumtext` MEDIUMTEXT ,

`longblob` LONGBLOB ,

`longtext` LONGTEXT ,

`enum` ENUM( '1', '2', '3' ) ,

`set` SET( '1', '2', '3' ) ,

`bool` BOOL ,

`binary` BINARY( 20 ) ,

`varbinary` VARBINARY( 20 )

) ENGINE= MYISAM ;

### SQL SELECT statement syntax

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

SELECT [DISTINCT|ALL ] { \* | [fieldExpression [AS newName]} FROM 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 string fields into one.
* **FROM** tableName is mandatory and must contain at least one table, multiple tables must be separated using commas or joined using the JOIN keyword.
* **WHERE** condition is optional, it can be used to specify criteria in the result set returned from the query.
* **GROUP BY** is used to put together records that have the same field values.
* **HAVING** condition is used to specify criteria when working using the GROUP BY keyword.
* **ORDER BY** is used to specify the sort order of the result set.

**\***

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

SELECT \* FROM `members`;

SELECT `full\_names`,`gender`,`physical\_address`, `email` FROM `members`;

SELECT Concat(`title`, ' (', `director`, ')') , `year\_released` FROM `movies`;

SELECT Concat(`title`, ' (', `director`, ')') AS 'Concat', `year\_released` FROM `movies`;

SELECT `membership\_number`,`full\_names`,LEFT(`date\_of\_birth`,4) AS `year\_of\_birth` FROM members;

#### WHERE clause Syntax

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

SELECT \* FROM tableName WHERE condition;

SELECT \* FROM `members` WHERE `membership\_number` = 1;

SELECT \* FROM `movies` WHERE `category\_id` = 2 AND `year\_released` = 2008;

SELECT \* FROM `movies` WHERE `category\_id` = 1 OR `category\_id` = 2;

SELECT \* FROM `members` WHERE `membership\_number` IN (1,2,3);

SELECT \* FROM `members` WHERE `membership\_number` NOT IN (1,2,3);

SELECT \* FROM `payments` WHERE `amount\_paid` > 2000; SELECT \* FROM `movies` WHERE `category\_id`<> 1;

# MySQL query INSERT INTO Table with Examples

INSERT INTO `table\_name`(column\_1,column\_2,...) VALUES (value\_1,value\_2,...);

INSERT INTO `members` (`full\_names`,`gender`,`physical\_address`,`contact\_number`) VALUES ('Leonard Hofstadter','Male','Woodcrest',0845738767);

### Inserting into  a Table from another Table

INSERT INTO table\_1 SELECT \* FROM table\_2;

INSERT INTO `categories\_archive` SELECT \* FROM `categories`;

INSERT INTO `categories\_archive`(category\_id,category\_name,remarks) SELECT category\_id,category\_name,remarks FROM `categories`;

# MySQL UPDATE & DELETE Query with Example

DELETE FROM `table\_name` [WHERE condition];

DELETE FROM `movies` WHERE `movie\_id` = 18;

DELETE FROM `movies` WHERE `movie\_id` IN (20,21);

UPDATE `table\_name` SET `column\_name` = `new\_value' [WHERE condition];

UPDATE `members` SET `contact\_number` = '0759 253 542' WHERE `membership\_number` = 1;

UPDATE `members` SET `full\_names` = 'Janet Smith Jones', `physical\_address` = 'Melrose 123' WHERE `membership\_number` = 2;

# ORDER BY in MySQL: DESC & ASC

SELECT statement... [WHERE condition | GROUP BY `field\_name(s)` HAVING condition] ORDER BY `field\_name(s)` [ASC | DESC];

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

SELECT \* FROM members ORDER BY date\_of\_birth DESC;

SELECT \* FROM members ORDER BY date\_of\_birth ASC;

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

SELECT \* FROM `members` ORDER BY `gender`,`date\_of\_birth` DESC;

# MySQL GROUP BY and HAVING Clause with Examples

SELECT statements... GROUP BY column\_name1[,column\_name2,...] [HAVING condition];

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

SELECT `gender` FROM `members` ;

|  |
| --- |
| **gender** |
| Female |
| Female |
| Male |
| Female |
| Male |
| Male |
| Male |
| Male |
| Male |

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

|  |
| --- |
| **gender** |
| Female |
| Male |

SELECT `category\_id`,`year\_released` FROM `movies` GROUP BY `category\_id`,`year\_released`;

SELECT `gender`,COUNT(`membership\_number`) FROM `members` GROUP BY `gender`;

|  |  |
| --- | --- |
| **gender** | **COUNT('membership\_number')** |
| Female | 3 |
| Male | 5 |

SELECT \* FROM `movies` GROUP BY `category\_id`,`year\_released` HAVING `category\_id` = 8;

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

SELECT \* FROM members WHERE postal\_address = 'Austin , TX' OR postal\_address = Dallas , TX OR postal\_address = Iola,TX OR postal\_adress = Houston ,TX';

SELECT \* FROM members WHERE postal\_address like '% TX';

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

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

### \_ underscore wildcard

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

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 |

### NOT Like

SELECT \* FROM movies WHERE year\_released NOT LIKE '200\_';

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **movie\_id** | **title** | **director** | **year\_released** | **category\_id** |
| 1 | Pirates of the Caribean 4 | Rob Marshall | 2011 | 1 |
| 4 | Code Name Black | Edgar Jimz | 2010 | NULL |
| 8 | Underworld-Awakeninh | Michahel Eal | 2012 | 6 |

### Escape keyword.

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

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

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

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

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

The same query will also work if we use something like

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

# MYSQL Regular Expressions (REGEXP) with Syntax & Examples

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

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

Let's suppose that we want to search for movies that start with a, b, c or d , followed by any number of other characters, how would we go about to achieve that. We can use a regular expression together with the metacharacters to achieve our desired results. Executing the above script in MySQL workbench against the myflixdb gives us the following results.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

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

### ****Types of functions****

**Built-in functions**

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

* **Strings functions** - operate on string data types
* **Numeric functions** - operate on numeric data types
* **Date functions** - operate on date data types
* [**Aggregate functions**](https://www.guru99.com/aggregate-functions.html) - operate on all of the above data types and produce summarized result sets.
* **Other functions** - MySQL also supports other types of built in functions but we will limit our lesson to the above named functions only.
* SELECT `movie\_id`,`title`, UCASE(`title`) FROM `movies`;

**Arithematic operators**

|  |  |
| --- | --- |
| **Name** | **Description** |
| DIV | Integer division |
| / | Division |
| - | Subtraction |
| + | Addition |
| \* | Multiplication |
| % or MOD | Modulus |

SELECT 23 DIV 6 ;

SELECT 23 / 6 ;

SELECT 23 - 6 ;

SELECT 23 + 6 ;

SELECT 23 \* 6 AS `multiplication\_result`;

SELECT 23 % 6 ;

SELECT 23 MOD 6 ;

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

SELECT FLOOR(23 / 6) AS `floor\_result`;

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

SELECT ROUND(23 / 6) AS `round\_result`;

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

SELECT RAND() AS `random\_result`;

### Stored functions

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

CREATE FUNCTION sf\_name ([parameter(s)])

RETURNS data type

DETERMINISTIC

STATEMENTS

**HERE**

* **"CREATE FUNCTION sf\_name ([parameter(s)]) "** is mandatory and tells MySQL server to create a function named `sf\_name' with optional parameters defined in the parenthesis.
* **"RETURNS data type"** is mandatory and specifies the data type that the function should return.
* **"DETERMINISTIC"** means the function will return the same values if the same arguments are supplied to it.
* **"STATEMENTS"** is the procedural code that the function executes.
* DELIMITER |
* CREATE FUNCTION sf\_past\_movie\_return\_date (return\_date DATE)
* RETURNS VARCHAR(3)
* DETERMINISTIC
* BEGIN
* DECLARE sf\_value VARCHAR(3);
* IF curdate() > return\_date
* THEN SET sf\_value = 'Yes';
* ELSEIF curdate() <= return\_date
* THEN SET sf\_value = 'No';
* END IF;
* RETURN sf\_value;
* END|

SELECT `movie\_id`,`membership\_number`,`return\_date`,CURDATE() ,sf\_past\_movie\_return\_date(`return\_date`) FROM `movierentals`;

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

1. COUNT  
   2) SUM  
   3) AVG  
   4) MIN  
   5) MAX
2. SELECT COUNT(`movie\_id`) FROM `movierentals` WHERE `movie\_id` = 2;
3. SELECT DISTINCT `movie\_id` FROM `movierentals`;
4. SELECT MIN(`year\_released`) FROM `movies`;
5. SELECT MAX(`year\_released`) FROM `movies`;
6. SELECT SUM(`amount\_paid`) FROM `payments`;
7. SELECT AVG(`amount\_paid`) FROM `payments`;
8. SELECT m.`full\_names`,COUNT(p.`payment\_id`) AS `paymentscount`,AVG(p.`amount\_paid`) AS `averagepaymentamount`,SUM(p.`amount\_paid`) AS `totalpayments` FROM members m, payments p WHERE m.`membership\_number` = p.`membership\_number` GROUP BY m.`full\_names`;

# MySQL IS NULL & IS NOT NULL Tutorial with EXAMPLES

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

## NOT NULL Values

## NOT NULL Values

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

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

CREATE TABLE `employees`(

employee\_number int NOT NULL,

full\_names varchar(255) ,

gender varchar(6)

);

`comlumn\_name' IS NULL

`comlumn\_name' NOT NULL

SELECT \* FROM `members` WHERE contact\_number IS NOT NULL;

SELECT \* FROM `members` WHERE contact\_number IS NULL;

### Comparing null value****s****

SELECT 5 =5;

|  |
| --- |
| **5 =5** |
| 1 |

SELECT NULL = NULL;

|  |
| --- |
| **NULL = NULL** |
| NULL |

SELECT 5 > 5;

|  |
| --- |
| **5 > 5** |
| 0 |

SELECT NULL > NULL;

|  |
| --- |
| **NULL > NULL** |
| NULL |

SELECT 5 IS NULL;

|  |
| --- |
| **5 IS NULL** |
| 0 |

SELECT NULL IS NULL;

|  |
| --- |
| **NULL IS NULL** |
| 1 |

# MySQL AUTO\_INCREMENT with Examples

CREATE TABLE `categories` (

`category\_id` int(11) AUTO\_INCREMENT,

`category\_name` varchar(150) DEFAULT NULL,

`remarks` varchar(500) DEFAULT NULL,

PRIMARY KEY (`category\_id`)

);

SELECT LAST\_INSERT\_ID();

# MYSQL - ALTER, DROP, RENAME, MODIFY

ALTER TABLE `table\_name` ADD COLUMN `column\_name` `data\_type`;

SHOW COLUMNS FROM `members`;

ALTER TABLE `members` ADD COLUMN `credit\_card\_number` VARCHAR(25);

### WHAT IS THE DROP COMMAND?

The DROP command is used to

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

ALTER TABLE `members` DROP COLUMN `credit\_card\_number`;

**DROP TABLE**

DROP TABLE `categories\_archive`;

### WHAT IS THE RENAME COMMAND?

RENAME TABLE `current\_table\_name` TO `new\_table\_name`;

RENAME TABLE `movierentals` TO `movie\_rentals`;

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

 We can accomplish this using the change command as follows -

ALTER TABLE `members` CHANGE COLUMN `full\_names` `fullname` char(250) NOT NULL;

### MODIFY KEYWORD

**The MODIFY Keyword allows you to**

1. Modify Column Data Type
2. Modify Column Constraints
3. In the CHANGE example above, we had to change the field name as well other details. **Omitting the field name from the CHANGE statement will generate an error.**Suppose we are only interested in changing the data type and constraints on the field without affecting the field name, we can use the MODIFY keyword to accomplish that.
4. The script below changes the width of "fullname" field from 250 to 50.
5. ALTER TABLE `members`MODIFY `fullname` char(50) NOT NULL;

### AFTER KEYWORD

ALTER TABLE `members` ADD `date\_of\_registration` date NULL AFTER `date\_of\_birth`;

# MySQL LIMIT & OFFSET with Examples

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

SELECT \* FROM members LIMIT 2;

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

# MySQL SubQuery Tutorial with Examples

SELECT category\_name FROM categories WHERE category\_id =( SELECT MIN(category\_id) from movies);

SELECT full\_names,contact\_number FROM members WHERE membership\_number IN (SELECT membership\_number FROM movierentals WHERE return\_date IS NULL );

Select full\_names From members WHERE membership\_number = (SELECT membership\_number FROM payments WHERE amount\_paid = (SELECT MAX(amount\_paid) FROM payments));

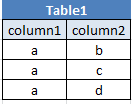
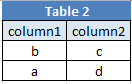
# MySQL UNION

### What is a 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 .

Suppose we have two tables as follows

[[](https://www.guru99.com/images/Table1%281%29.png)](https://www.guru99.com/images/Table1%281%29.png)[](https://www.guru99.com/images/Table2%281%29.png)

Let's now create a UNION query to combines both tables using DISTINCT

SELECT `column1`,` column1 FROM `table1`

UNION DISTINCT

SELECT ` column1`,` column1` 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.

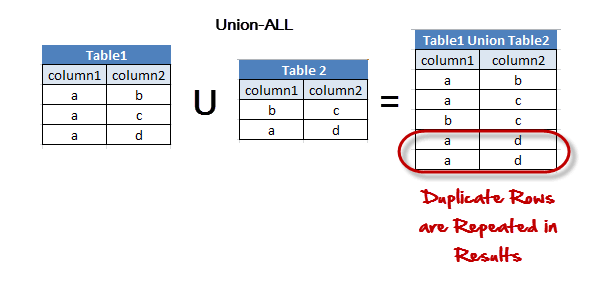
Let's now create a UNION query to combines both tables using ALL

SELECT `column1`,` column1` FROM `table1`

UNION ALL

SELECT ` column1`,` column1` FROM `table2`;

Here duplicate rows are included and since we use ALL.

[](https://www.guru99.com/images/Table1UnionTable2ALL.png)

### Why use unions

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.

### Summary

* 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.
* SELECT `membership\_number`,`full\_names` FROM `members`
* UNION
* SELECT `movie\_id`,`title` FROM `movies`;

# MySQL JOINS Tutorial: INNER, OUTER, LEFT, RIGHT, CROSS

### What are JOINS?

Joins help retrieving data from two or more database tables.

The tables are mutually related using primary and foreign keys.

Note: JOIN is the most misunderstood topic amongst SQL leaners. For sake of simplicity and ease of understanding , we will be using a new Database to practice sample.  As shown below

|  |  |  |  |
| --- | --- | --- | --- |
| **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 Tin | | 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 joins

**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.

[](https://www.guru99.com/images/CrossJoin.png)

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

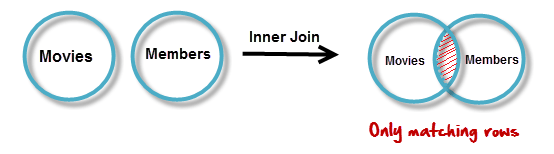
Executing the above script in MySQL workbench gives us the following results.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **id** | **title** |  | **id** | **first\_name** | **last\_name** | **movie\_id** |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 1 | Adam | Smith | 1 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 2 | Ravi | Kumar | 2 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 3 | Susan | Davidson | 5 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 4 | Jenny | Adrianna | 8 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 6 | Lee | Pong | 10 |
| 2 | Real Steel(2012) | Animations | 1 | Adam | Smith | 1 |
| 2 | Real Steel(2012) | Animations | 2 | Ravi | Kumar | 2 |
| 2 | Real Steel(2012) | Animations | 3 | Susan | Davidson | 5 |
| 2 | Real Steel(2012) | Animations | 4 | Jenny | Adrianna | 8 |
| 2 | Real Steel(2012) | Animations | 6 | Lee | Pong | 10 |
| 3 | Alvin and the Chipmunks | Animations | 1 | Adam | Smith | 1 |
| 3 | Alvin and the Chipmunks | Animations | 2 | Ravi | Kumar | 2 |
| 3 | Alvin and the Chipmunks | Animations | 3 | Susan | Davidson | 5 |
| 3 | Alvin and the Chipmunks | Animations | 4 | Jenny | Adrianna | 8 |
| 3 | Alvin and the Chipmunks | Animations | 6 | Lee | Pong | 10 |
| 4 | The Adventures of Tin Tin | Animations | 1 | Adam | Smith | 1 |
| 4 | The Adventures of Tin Tin | Animations | 2 | Ravi | Kumar | 2 |
| 4 | The Adventures of Tin Tin | Animations | 3 | Susan | Davidson | 5 |
| 4 | The Adventures of Tin Tin | Animations | 4 | Jenny | Adrianna | 8 |
| 4 | The Adventures of Tin Tin | Animations | 6 | Lee | Pong | 10 |
| 5 | Safe (2012) | Action | 1 | Adam | Smith | 1 |
| 5 | Safe (2012) | Action | 2 | Ravi | Kumar | 2 |
| 5 | Safe (2012) | Action | 3 | Susan | Davidson | 5 |
| 5 | Safe (2012) | Action | 4 | Jenny | Adrianna | 8 |
| 5 | Safe (2012) | Action | 6 | Lee | Pong | 10 |
| 6 | Safe House(2012) | Action | 1 | Adam | Smith | 1 |
| 6 | Safe House(2012) | Action | 2 | Ravi | Kumar | 2 |
| 6 | Safe House(2012) | Action | 3 | Susan | Davidson | 5 |
| 6 | Safe House(2012) | Action | 4 | Jenny | Adrianna | 8 |
| 6 | Safe House(2012) | Action | 6 | Lee | Pong | 10 |
| 7 | GIA | 18+ | 1 | Adam | Smith | 1 |
| 7 | GIA | 18+ | 2 | Ravi | Kumar | 2 |
| 7 | GIA | 18+ | 3 | Susan | Davidson | 5 |
| 7 | GIA | 18+ | 4 | Jenny | Adrianna | 8 |
| 7 | GIA | 18+ | 6 | Lee | Pong | 10 |
| 8 | Deadline(2009) | 18+ | 1 | Adam | Smith | 1 |
| 8 | Deadline(2009) | 18+ | 2 | Ravi | Kumar | 2 |
| 8 | Deadline(2009) | 18+ | 3 | Susan | Davidson | 5 |
| 8 | Deadline(2009) | 18+ | 4 | Jenny | Adrianna | 8 |
| 8 | Deadline(2009) | 18+ | 6 | Lee | Pong | 10 |
| 9 | The Dirty Picture | 18+ | 1 | Adam | Smith | 1 |
| 9 | The Dirty Picture | 18+ | 2 | Ravi | Kumar | 2 |
| 9 | The Dirty Picture | 18+ | 3 | Susan | Davidson | 5 |
| 9 | The Dirty Picture | 18+ | 4 | Jenny | Adrianna | 8 |
| 9 | The Dirty Picture | 18+ | 6 | Lee | Pong | 10 |
| 10 | Marley and me | Romance | 1 | Adam | Smith | 1 |
| 10 | Marley and me | Romance | 2 | Ravi | Kumar | 2 |
| 10 | Marley and me | Romance | 3 | Susan | Davidson | 5 |
| 10 | Marley and me | Romance | 4 | Jenny | Adrianna | 8 |
| 10 | Marley and me | Romance | 6 | Lee | Pong | 10 |

### INNER JOIN

The inner JOIN is used to return rows from both tables that satisfy the given condition.

Suppose , you want to get list of members who have rented movies together with titles of movies rented by them. You can simply use an INNER JOIN for that, which returns rows from both tables that satisfy with given conditions.

[](https://www.guru99.com/images/InnerJoin.png)

SELECT members.`first\_name` , members.`last\_name` , movies.`title`

FROM members ,movies

WHERE movies.`id` = members.`movie\_id`

Executing the above script give

|  |  |  |
| --- | --- | --- |
| **first\_name** | **last\_name** | **title** |
| Adam | Smith | ASSASSIN'S CREED: EMBERS |
| Ravi | Kumar | Real Steel(2012) |
| Susan | Davidson | Safe (2012) |
| Jenny | Adrianna | Deadline(2009) |
| Lee | Pong | Marley and me |

Note the above results script can also be written as follows to achieve the same results.

SELECT A.`first\_name` , A.`last\_name` , B.`title`

FROM `members`AS A

INNER JOIN `movies` AS B

ON B.`id` = A.`movie\_id`

### Outer JOINs

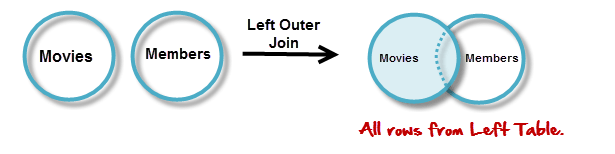
MySQL Outer JOINs return all records matching from both tables .

It can detect records having no match in joined table. It returns **NULL** values for records of joined table if no match is found.

Sounds Confusing ? Let's look into an example -

### LEFT JOIN

Assume now you want to get titles of all movies together with names of members who have rented them. It is clear that some movies have not being rented by any one. We can simply use **LEFT JOIN**for the purpose.

[](https://www.guru99.com/images/LeftOuterJoin.png)

The LEFT JOIN returns all the rows from the table on the left even if no matching rows have been found in the table on the right. **Where no matches have been found in the table on the right, NULL is returned.**

SELECT A.`title` , B.`first\_name` , B.`last\_name`

FROM `movies` AS A

LEFT JOIN `members` AS B

ON B.`movie\_id` = A.`id`

Executing the above script in MySQL workbench gives.You can see that in the returned result which is listed below that for movies which are not rented, member name fields are having NULL values. That means no matching member found members table for that particular movie.

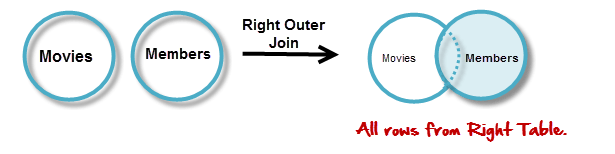
|  |  |  |
| --- | --- | --- |
| **title** | **first\_name** | **last\_name** |
| ASSASSIN'S CREED: EMBERS | Adam | Smith |
| Real Steel(2012) | Ravi | Kumar |
| Safe (2012) | Susan | Davidson |
| Deadline(2009) | Jenny | Adrianna |
| Marley and me | Lee | Pong |
| Alvin and the Chipmunks | NULL | NULL |
| The Adventures of Tin Tin | NULL | NULL |
| Safe House(2012) | NULL | NULL |
| GIA | NULL | NULL |
| The Dirty Picture | NULL | NULL |

Note: Null is returned for non-matching rows on right

### RIGHT JOIN

RIGHT JOIN is obviously the opposite of LEFT JOIN. The RIGHT JOIN returns all the columns from the table on the right even if no matching rows have been found in the table on the left. Where no matches have been found in the table on the left, NULL is returned.

In our example,  let's assume that you need to get names of members and movies rented by them. Now we have a new member who has not rented any movie yet

[](https://www.guru99.com/images/RightOuterJoin.png)

SELECT A.`first\_name` , A.`last\_name`, B.`title`

FROM `members` AS A

RIGHT JOIN `movies` AS B

ON B.`id` = A.`movie\_id`

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

|  |  |  |
| --- | --- | --- |
| **first\_name** | **last\_name** | **title** |
| Adam | Smith | ASSASSIN'S CREED: EMBERS |
| Ravi | Kumar | Real Steel(2012) |
| Susan | Davidson | Safe (2012) |
| Jenny | Adrianna | Deadline(2009) |
| Lee | Pong | Marley and me |
| NULL | NULL | Alvin and the Chipmunks |
| NULL | NULL | The Adventures of Tin Tin |
| NULL | NULL | Safe House(2012) |
| NULL | NULL | GIA |
| NULL | NULL | The Dirty Picture |

Note: Null is returned for non-matching rows on left

### "ON" and "USING" clauses

In above JOIN query examples, we have used ON clause to match the records between table.

USING clause can also be used for the same purpose. The difference with **USING**is it **needs to have identical names for matched columns in both tables.**

In "movies" table so far we used its primary key with the name "id". We referred to same in  "members" table with the name "movie\_id".

Let's rename "movies" tables "id" field to have the name "movie\_id". We do this in order to have identical matched field names.

ALTER TABLE `movies` CHANGE `id` `movie\_id` INT( 11 ) NOT NULL AUTO\_INCREMENT;

Next let's use USING with above LEFT JOIN example.

SELECT A.`title` , B.`first\_name` , B.`last\_name`

FROM `movies` AS A

LEFT JOIN `members` AS B

USING ( `movie\_id` )

Apart from using **ON**and **USING with JOINs**you can use many other MySQL clauses like **GROUP BY, WHERE**and even functions like **SUM**, **AVG**, etc.

### Why should we use joins?

Now you may think, why we use JOINs when we can do the same task running queries. Especially if you have some experience in database programming you know we can run queries one by one, use output of each in successive queries. Of course, that is possible. But using JOINs, you can get the work done by using only a one query with any search parameters. On the other hand **MySQL can achieve better performance** with JOINs as it can use Indexing. Simply use of single JOIN query instead running multiple queries do reduce server overhead. Using multiple queries instead that leads more data transfers between MySQL and applications (software). Further it requires more data manipulations in application end also.

**It is clear that we can achieve better MySQL and application performances by use of JOINs.**

### ****Summary****

* JOINS allow us to combine data from more than one table into a single result set.
* JOINS have better performance compared to sub queries
* INNER JOINS only return rows that meet the given criteria.
* OUTER JOINS can also return rows where no matches have been found. The unmatched rows are returned with the NULL keyword.
* The major JOIN types include Inner, Left Outer, Right Outer, Cross JOINS etc.
* The frequently used clause in JOIN operations is "ON". "USING" clause requires that matching columns be of the same name.
* JOINS can also be used in other clauses such as GROUP BY, WHERE, SUB QUERIES, AGGREGATE FUNCTIONS etc.

# Views in MySQL Tutorial: Create, Join & Drop with Examples

### What are views?

Simply put, VIEWS are virtual tables .By virtual, we mean, the tables do not store any data of their own but display data stored in other tables.

In other words, VIEWS are nothing but SELECT Queries.

### 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**

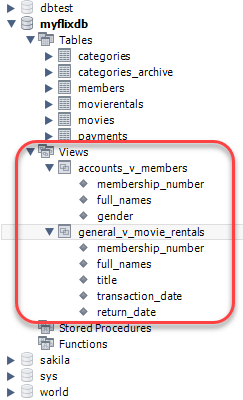
* **"CREATE VIEW `view\_name`"** tells MySQL server to create a view object in the database named `view\_name`
* **"AS SELECT statement"** is the SQL statements to be packed in the views. It can be a SELECT statement can contain data from one table or multiple tables.

Let's now create our first view using the "myflixdb" we will create a simple view that restricts the columns seen in the members 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 script in MySQL workbench against the myflixdb and expanding the views node in the database explorer gives us the following results.

[](https://www.guru99.com/images/database_explorer.png)

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`;

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

|  |  |  |
| --- | --- | --- |
| **membership\_number** | **full\_names** | **gender** |
| 1 | Janet Jones | Female |
| 2 | Janet Smith Jones | Female |
| 3 | Robert Phil | Male |
| 4 | Gloria Williams | Female |
| 5 | Leonard Hofstadter | Male |
| 6 | Sheldon Cooper | Male |
| 7 | Rajesh Koothrappali | Male |
| 8 | Leslie Winkle | Male |
| 9 | Howard Wolowitz | Male |

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 shown below to do that.

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.

### Joins and views

Let's now look at a fairly complex example which involves multiple tables and uses joins.

We will package the JOIN created that gets information from three (3) tables namely members, movies and movie rentals. Below is the script that helps us to achieve that.

CREATE VIEW `general\_v\_movie\_rentals` AS SELECT mb.`membership\_number`,mb.`full\_names`,mo.`title`,mr.`transaction\_date`,mr.`return\_date` FROM `movierentals` AS mr INNER JOIN `members` AS mb ON mr.`membership\_number` = mb.`membership\_number` INNER JOIN `movies` AS mo ON mr.`movie\_id` = mo.`movie\_id`;

Executing the above scripts creates the view named general\_v\_movie\_rentals in our myflixdb

Let's now select all the fields from a table named general\_v\_movie\_rentals.

SELECT \* FROM `general\_v\_movie\_rentals`;

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **membership\_number** | **full\_names** | **title** | **transaction\_date** | **return\_date** |
| 1 | Janet Jones | Pirates of the Caribean 4 | 20-06-2012 | 28-06-2012 |
| 1 | Janet Jones | Forgetting Sarah Marshal | 22-06-2012 | 25-06-2012 |
| 3 | Robert Phil | Forgetting Sarah Marshal | 22-06-2012 | 25-06-2012 |
| 2 | Janet Smith Jones | Forgetting Sarah Marshal | 21-06-2012 | 24-06-2012 |
| 3 | Robert Phil | X-Men | 23-06-2012 | 28-06-2012 |

Note we didn't have to write the complex JOIN query to get information about members, movies and movie rental details. We simply used the view in a regular SELECT statement as any other ordinary table. The view can be called from anywhere in the application system running on top of the myflixdb.

### Dropping views

The DROP command can be used to delete a view from the database that is no longer required. The basic syntax to drop a view is as follows.

DROP VIEW ` general\_v\_movie\_rentals `;

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 orignal 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 eliminates 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.

### Summary

* 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.

# MySQL Index Tutorial - Create, Add & Drop

### What are Index?

Nobody likes slow systems.

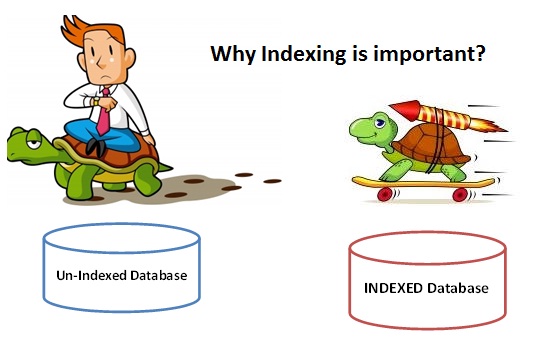
High system performance is of prime importance in almost all database systems .

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

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

Optimizing your database is a cheaper and better solution.

Towards this end we can use INDEXES.

[](https://www.guru99.com/images/Index.jpg)

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

### Create index basic syntax

Indexes can be defined in 2 ways

1.       At the time of table creation

2.       After table has been created

Example:

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

We will add the "full\_names" column to Index in a new table "members\_indexed".

The script shown below helps us to achieve that.

CREATE TABLE `members\_indexed` (

`membership\_number` int(11) NOT NULL AUTO\_INCREMENT,

`full\_names` varchar(150) DEFAULT NULL,

`gender` varchar(6) DEFAULT NULL,

`date\_of\_birth` date DEFAULT NULL,

`physical\_address` varchar(255) DEFAULT NULL,

`postal\_address` varchar(255) DEFAULT NULL,

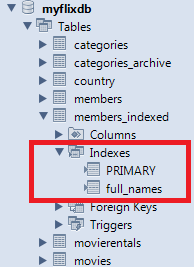
`contact\_number` varchar(75) DEFAULT NULL,

`email` varchar(255) DEFAULT NULL,

PRIMARY KEY (`membership\_number`),INDEX(full\_names)

) ENGINE=InnoDB;

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

[](https://www.guru99.com/images/tables_explorer.png)

Refreshing the myflixdb shows the newly created table named members\_indexed.

"Note" members\_indexed table has "full\_names" in the indexes node.

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

### Add index basic syntax

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

 We can use following query to add index

CREATE INDEX id\_index ON table\_name(column\_name);

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

CREATE INDEX `title\_index` ON `movies`(`title`);

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

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

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

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

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

SHOW INDEXES FROM table\_name;

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

SHOW INDEXES FROM `movies`;

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

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

### Drop index basic syntax

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

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

DROP INDEX `index\_id` ON `table\_name`;

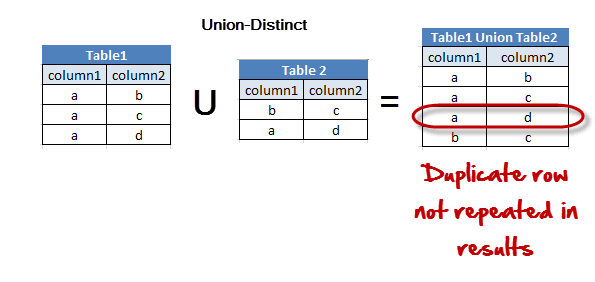
Let's now look at a practical example.

DROP INDEX ` full\_names` ON `members\_indexed`;

Executing the above command drops the index with id ` full\_names ` from the members\_indexed table.

### Summary

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

[](https://www.guru99.com/images/Table1UnionTable2Distinct.png)