SQL Refresher

Creating Tables in MySQL using SQL commands

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

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

"CREATE TABLE" is the one responsible for the creation of the table in the database.
"[IF NOT EXISTS]" is optional and only create the table if no matching table name is found.
"fieldName" is the name of the field and "data Type" defines the nature of the data to be stored in the field.
"[optional parameters]" additional information about a field such as
" AUTO_INCREMENT" , NOT NULL etc

Create Table

- Click on the SQL tab
- Use the following SQL to create a new table in your database as seen below

```
CREATE TABLE IF NOT EXISTS Members (
membership_number INT,
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 the 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 a Table carrying all data types – an example

- Use the following SQL to create a table which has data of all data types

```
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)
);
```

SQL SELECT – WHERE – AND WILDCARDS

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)
```
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, field expression 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.

- Download the file MovieDB.sql from the Blackboard page to a local directory of your choice
- Select Open SQL script from the File menu and open the MovieDB.sql file.
- Execute it and refresh your schema.
- Your database - MovieDB will be created, and you will be able to see it in the schema navigator.

Open a new query tab and write:

```
SELECT * FROM members;
```

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.

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

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

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

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

Getting movies listing

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

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

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

Alias field names

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

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

Getting members listing showing the year of birth

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

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

SELECT with WHERE Clause

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

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

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

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

• **"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.

WHERE clause combined with - AND LOGICAL Operator

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

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

```
SELECT * FROM `movies` WHERE `category_id` = 2 AND `year_released` = 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.

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 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 affects the rows whose values matches the list of values provided in the NOT IN keyword. The following query gives rows where membership_number is NOT 1 , 2 or 3

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

WHERE clause combined with - COMPARISON OPERATORS

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

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

> Greater than

The following script gets all the payments that are greater than 2,000 from the payments table.

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

< > Not Equal To

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

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

Let's suppose that we want to get a list of rented movies that have not been returned on time 24/06/2012. We can use the WHERE clause together with the less than comparison operator and AND logical operator to achieve that.

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

SQL wildcards

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

% the percentage

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

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

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

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

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

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

Notice that only one record has been returned from the database. This is because our code matches any number of characters at the beginning of the movie title and gets only records that end with the pattern "code".
Let's now shift the percentage wildcard to the end of the specified pattern to be matched. The modified script is shown below.

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

_ underscore wildcard

The underscore wildcard is used to **match exactly one character**. Let's suppose that we want to search for all the movies that were released in the years 200x where x is exactly one

character that could be any value. We would use the underscore wild card to achieve that. The script below select all the movies that were released in the year "200x"

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

NOT Like

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

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

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.

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

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

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

The same query will also work if we use something like

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

SQL INSERT – UPDATE – GROUP

SQL INSERT statement syntax

Basic syntax

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

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

- **INSERT INTO `table_name`** is the command that tells MySQL server to add new row into a table named `table_name`.
- **(column_1,column_2,...)** specifies the columns to be updated in the new row
- **VALUES (value_1,value_2,...)** specifies the values to be added into the new row

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

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

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

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

Inspect the inserted record using in the members table.

In order to avoid dropping a zero at the beginning of a numerical value that should be treated as a textual input (string), the value must be enclosed in single quotes

Now re-insert the record using the following statement

```
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

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

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

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

Example:-

```
INSERT INTO `members` VALUES (9,'Howard
Wolowitz','Male','1981/08/24','SouthPark','P.O. Box 4563', '0987786553',
'lwolowitz[at]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. Let's now look at a practical example, we will create a dummy table for movie categories for demonstration purposes. We will call the new categories table categories_archive. The script shown below creates the table.

```
CREATE TABLE `categories_archive` ( `category_id` int(11)
AUTO_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 us to achieve that.

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

Executing the above script inserts all the rows from the categories table into the categories archive table. Note the table structures will have to be the same for the script to work

SQL DELETE statement syntax

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

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

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

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

```
INSERT INTO `movies` (`title`, `director`, `year_released`, `category_id`)
VALUES ('The Great Dictator', 'Chalie Chaplie', 1920, 7);

INSERT INTO `movies` (`title`, `director`, `category_id`) VALUES ('sample
movie', 'Anonymous', 8);

INSERT INTO movies (`title`, `director`, `year_released`, `category_id`)
VALUES ('movie 3', 'John Brown', 1920, 8);
```

Let's suppose that the MovieDB video library no longer wishes to be renting out "The Great Dictator" to its members and they want it removed from the database. Its movie ID is 18, we

can use the script shown below to delete its row from the movies table.

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

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

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

SQL UPDATE statement syntax

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

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

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

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

The following script helps us to do that.

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

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

SQL ORDER BY clause

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

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

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

descending order. Note ***ASC*** is used as the default.

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;
```

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

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

"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;
```

GROUP BY Syntax

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

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

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

- "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.

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

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

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

Let's execute the query using group by –

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

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

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

Grouping and aggregate functions

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

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

Restricting query results using the HAVING clause

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

Suppose we want to know the number of movies per category that were released since 2007. We would use the following script to achieve our results.

```
SELECT `category_id`, COUNT(`movie_id`),`year_released` FROM `movies`
GROUP BY `category_id`,`year_released` HAVING `year_released` >= 2007;
```

SQL Unions, Joins and Subqueries

Joins help retrieving data from two or more database tables. The tables are mutually related using primary and foreign keys.

Add a Foreign Key References to Table Members

Add a new field on the table **members** to hold a foreign key reference. Use the following SQL statement

```
ALTER TABLE `members` ADD `movie_id` INT( 11 ) NULL ;
```

Use the following statement to add a foreign key reference to each row of the members table. You will have to repeat the statement several times until all rows of the member table have a foreign key reference to a movie_id . Make sure your statements maintain the **referential**

integrity of the database

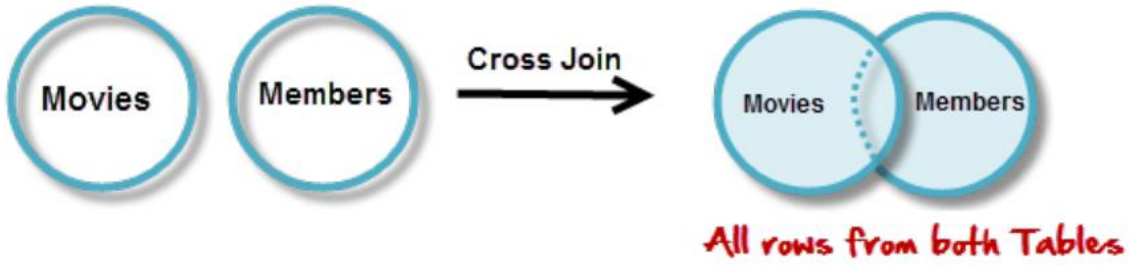
```
UPDATE `members` SET `movie_id` = '2' WHERE `members`.`membership_number` =9;
```

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.

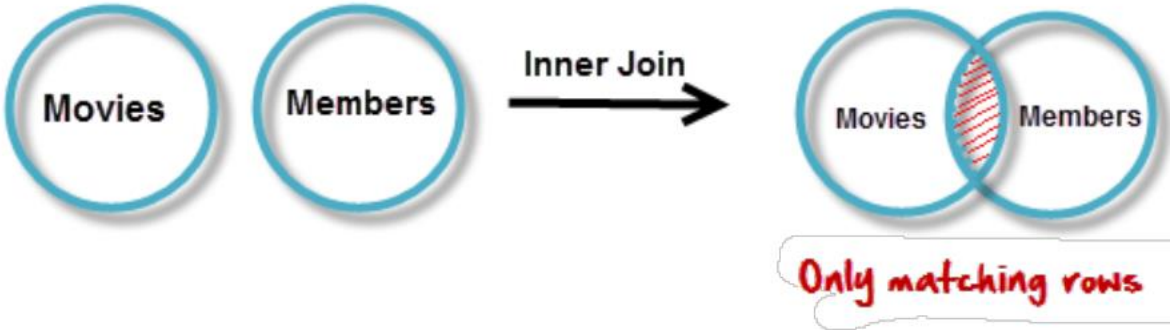
```
SELECT * FROM `movies` CROSS JOIN `members`
```



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.

```
SELECT members.`full_names` , movies.`title`
FROM members ,movies
WHERE movies.`movie_id` = members.`movie_id`
```



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

```
SELECT A.`full_names` , B.`title`
FROM `members`AS A
INNER JOIN `movies` AS B
ON B.`movie_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.

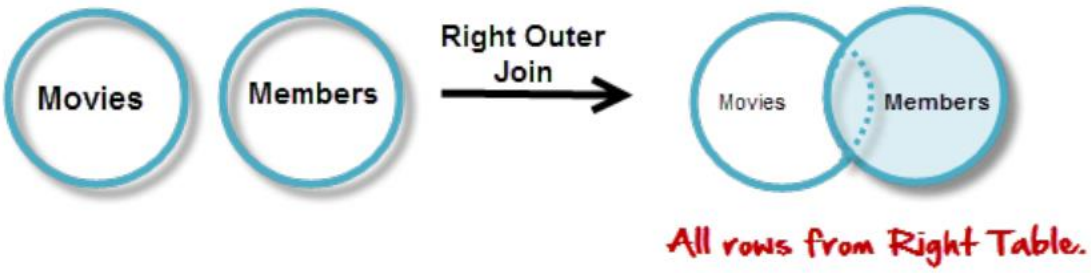
The LEFT JOIN returns all the rows from the table on the left even if no matching rows have been found in the table on the right. **Where no matches have been found in the table on the right, NULL is returned.**

```
SELECT A.`full_names`, B.`title`
FROM `members`AS A
LEFT JOIN `movies` AS B
ON B.`movie_id` = A.`movie_id`
```

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.

```
SELECT A.`full_names`, B.`title`
FROM `members`AS A
RIGHT JOIN `movies` AS B
ON B.`movie_id` = A.`movie_id`
```



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

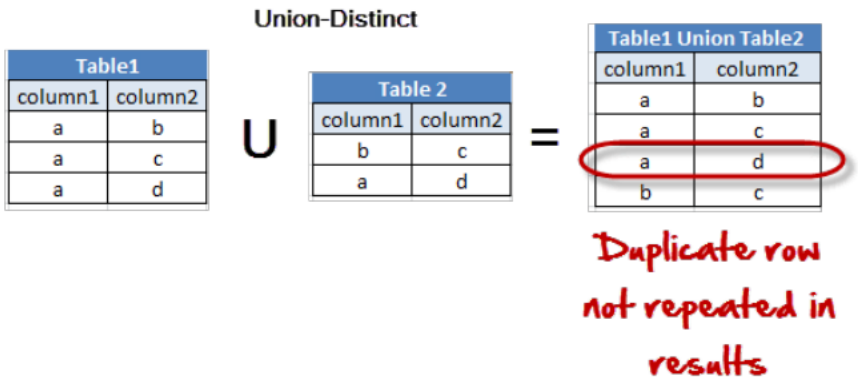
Table1	
column1	column2
a	b
a	c
a	d

Table 2	
column1	column2
b	c
a	d

Let's now create a UNION query to 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.



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.

In our MovieDB lets combine

membership_number and full_names from Members table with movie_id and title from movies table We can use the following query:

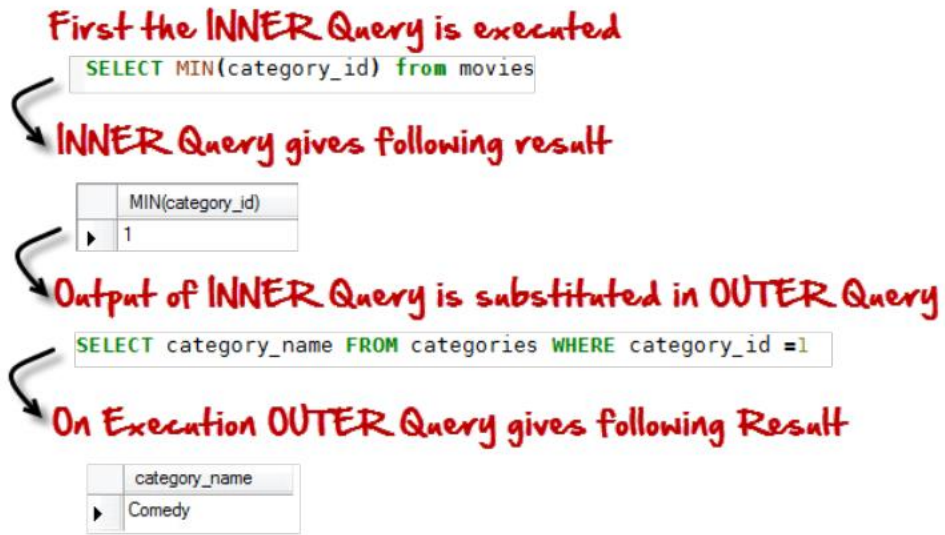
```
SELECT `membership_number`,`full_names` FROM `members`
UNION
SELECT `movie_id`,`title` FROM `movies`;
```

What are sub queries?

A sub query is a select query that is contained inside another query. The inner select query is usually used to determine the results of the outer select query.
A common customer complaint at the MyFlix Video Library is the low number of movie titles. The management wants to buy movies for a category which has least number of titles. You can use a query like

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

Let's see how this query works



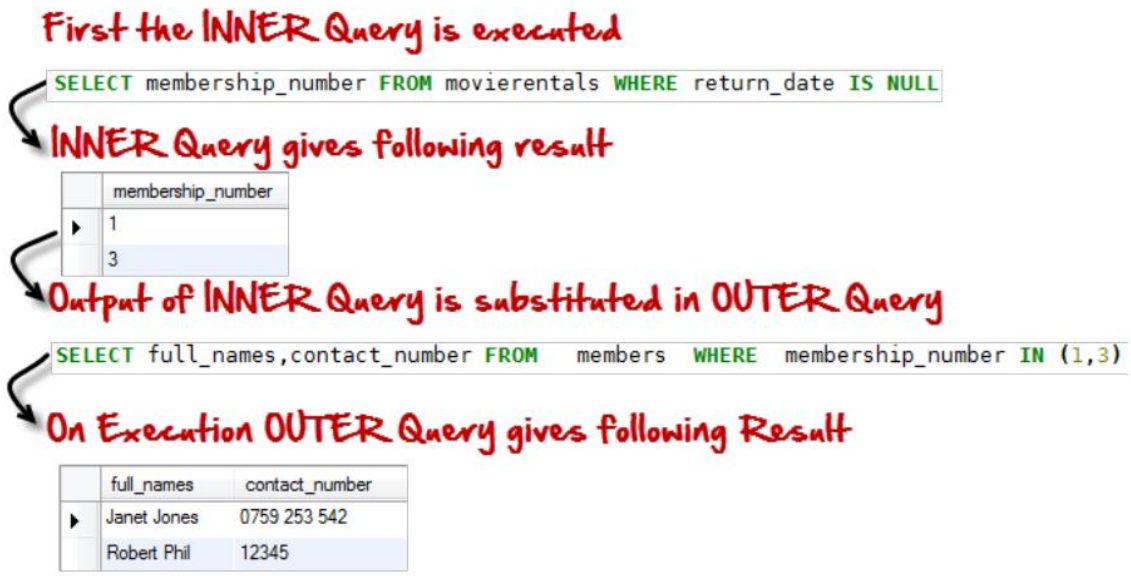
The above is a form of **Row Sub-Query**. In such sub-queries the , inner query can give only ONE result. The permissible operators when work with row subqueries are [=, >, =, <=, ,!=,]

Let's look at another example ,

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

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

Let's see how this query works



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

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

Suppose the management wants to reward the highest paying member.
We can run a query like

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

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 up to 500 times a performance boost.

Given a choice, it is recommended to use a JOIN over a sub-query.
Sub-Queries should only be used as a fallback solution when you cannot use a JOIN operation to achieve the above

SQL FUNCTIONS

Types of functions

Built-in functions

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

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

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

String functions

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

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

Numeric functions

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

Integer Division (DIV)

```
SELECT 23 DIV 6 ;
```

Division operator (/)

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

```
SELECT 23 / 6 ;
```

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

```
SELECT 23 - 6 ;
```

Addition operator (+)
Let's now look at the addition operator example. We will modify the previous example.

```
SELECT 23 + 6 ;
```

Multiplication operator (*)

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

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

Modulo operator (-)

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

```
SELECT 23 % 6 ;
```

```
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
```

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

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

Copy the following code in the SQL Query box (Note: This works well in MySQL 8.0 and above. Some adjustments in syntax may be required for other versions)

```
DELIMITER $$

CREATE FUNCTION sf_past_movie_return_date (return_date DATE)
RETURNS VARCHAR(3)
DETERMINISTIC
BEGIN
    DECLARE sf_value VARCHAR(3);

    -- Compare current date with return date
    IF CURDATE() > return_date THEN
        SET sf_value = 'Yes';
    ELSE
        SET sf_value = 'No';
    END IF;

    -- Return the value
    RETURN sf_value;
END $$

DELIMITER ;
```

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`;
```

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

Aggregate Functions are all about

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

The ISO standard defines five (5) aggregate functions namely;
1) COUNT
2) SUM
3) AVG
4) MIN
5) MAX

COUNT Function

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

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

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

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

DISTINCT Keyword

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

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

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

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

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

MIN function

The MIN function **returns the smallest value in the specified table field**.
As an example, let's suppose we want to know the year in which the oldest movie in our library was released, we can use MySQL's MIN function to get the desired information.
The following query helps us achieve that

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

MAX function

Just as the name suggests, the MAX function is the opposite of the MIN function. It **returns the largest value from the specified table field**.
Let's assume we want to get the year that the latest movie in our database was released. We can easily use the MAX function to achieve that.
The following example returns the latest movie year released.

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

SUM function

Suppose we want a report that gives total amount of payments made so far. We can use the MySQL **SUM** function which **returns the sum of all the values in the specified column**. **SUM works on numeric fields only**. **Null values are excluded from the result returned.** The query shown below gets the all payments made and sums them up to return a single result.

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

AVG function

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

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

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

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

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