MYSQL DDL COMMANDS

Create a database:

```
CREATE DATABASE <database_name>;
```

Example: CREATE DATABASE university;

Create a table inside a database:

(Click on the database in which you want create a table, then in SQL editor)

```
CREATE TABLE <table_name> (
<attribute1> <datatype>(<size>),
<attribute2> <datatype>(<size>),
......................,
<constraint1> <constraint name> <constraint parameter>,
<constraint2> <constraint name> <constraint parameter>,
...................................................
);
```

Example:

Q: create a table named 'course' with the following column names & datatypes

- *Given question pattern 1:*

Column Name	Datatype	Remarks
course_code	string	primary key
course_name	string	Not null
credit	integer	Can be null

- *Given question pattern 2:*

course(**course_code**, course_name, credit)

***__bold underlined__ columns name means it is a primary key for that table*

```
CREATE TABLE course (
   course_code VARCHAR(20)  PRIMARY KEY,
   course_name VARCHAR(20) NOT NULL,
   credit INT
);
```

Some common datatypes and their behavior

DATATYPES	Behavior	Example
INT	Whole number	172, 1163172, 1, 2
DECIMAL (*totalDigits, digitsAfterDecimaPoint****)***	Fraction number/decimal pointed number	*DECIMAL (3, 2)* → 3.79 *(for example)* *DECIMAL (2,1)* → 1.5 *(for example)*
FLOAT*(totalDigits, digitsAfterDecimaPoint****)***	Fraction number/decimal pointed number	*FLOAT (3, 2)* → 3.79 *(for example)* *FLOAT (2,1)* → 1.5 *(for example)*
VARCHAR(*maximumCharactersAllowed)*	strings	“CSE 3522”, “Gazzali” “+8801234567”
DATE	Date(YYYY-MM-DD)	2021-07-11
DATETIME	A date and time value, in 'CCYY-MM-DD hh:mm:ss' format	2021-07-11 20:01:00

Create a table “student” with foreigner key “course_taken” which references “course” tables primary key (course_code)

```
CONSTRAINT constraint_name FOREIGN
KEY(column_name_in_this_table_which_is_used_as_foreign_key_field)
REFERENCES the_table_we_are_referencing (primary_key_of_that_table)
```

Example:

student(<u>**ID**</u>, Name, Age, ***course_taken***)

*****bold itlaic** columns name means it is a foregner key for that table*

```
CREATE TABLE student (
ID INT PRIMARY KEY,
Name VARCHAR(20)NOT NULL,
Age INT,
course_taken VARCHAR(20),

CONSTRAINT fk_course_taken FOREIGN KEY(course_taken) REFERENCES
course(course_code)
);
```

Points to Remember:

1. **The DATATYPE** of the foreign key filed **MUST MATCH** with the datatype of primary key of the referenced table.
2. The **column name** of the foreign key in the referencing table **may not match** with the column name of the primary key in the referenced table, that’s not a problem.

   In the above example: “**course_taken**” in ***student*** table is the foreign key for “**course_code**” in ***course*** table. Column names are not same, but their datatypes are. (VARCHAR(20)).

3. While in **REFERENCE**, you must need to supply the **exact table name and column name of the primary key table.**

ALTER Commands

(Updated for new MySQL version)

Alter commands are used when we want to **modify or change the filed or column related stuff** of a table after they have been already created.

In other words, **ALTER commands works on the existing table set.**

ADD a **new column** to an existing table

```
ALTER TABLE <table_name> ADD COLUMN <column_name> <data_type> (<size>)
```

Example: add a column “Email” to the student table

```
ALTER TABLE student ADD column Email VARCHAR(20);
```

ADD **multiple columns** to an existing table

```
ALTER TABLE <table_name> ADD COLUMN (
<column_name1> <datatype>(<size>),
<column_name2> <datatype>(<size>),
.......................
);
```

Example: add two columns “CGPA” & “Hobby” in the student table

```
ALTER TABLE student ADD COLUMN (
CGPA FLOAT(3,2),
Hobby Varchar(20)
);
```

CHANGE/MODIFY the **Datatype** of a field (or, column)

```
ALTER TABLE <table_name> MODIFY COLUMN <column_name> <new_dataype>;
```

Example: Change the credit field’s type to FLOAT from INT

```
ALTER TABLE course MODIFY COLUMN credit FLOAT(2,1);
```

DROP (DELETE) A COLUMN

```
ALTER TABLE <table_name> DROP COLUMN <column_name>
```

Example: Delete “hobby” filed from student table

ALTER TABLE student DROP COLUMN Hobby;

RENAME A TABLE

```
ALTER TABLE <old_tablename> RENAME <new_tablename>
```

Example: Rename “course” table to “University_courses”

ALTER TABLE course RENAME university_courses;

DROP A TABLE

```
DROP TABLE <table_name>
```

Example: DROP / DELETE the student table

DROP TABLE student;

MYSQL DML Commands

Attributes == Columns/Fields

Records == Rows

Data Insertion into a Table

Syntax:

```
INSERT INTO <table name> (attribute1, attribute2, ...)
VALUES (<value for attribute1>, <value for attribute2>, ...)
```

Example:

Insert into department (dept_name, building, budget)
values ('CSE', 'Main Campus', 1000000)

Retrieving/Searching some Data/rows/records/results from database table (SQL query)

Syntax:

```
SELECT <column_name_needs_to_be_shown>

FROM <table_name>

WHERE <some_condition>
```

Example:

- Ques: Show **firstName, lastName, age** from the **students** table whose **student_id is 1163172**

  Ans:

  ```
  SELECT firstName, lastName, age

  FROM students

  WHERE student_id = 1163172
  ```

- Ques: Show **everything** (or every detail) of the **employee named** 'John Doe' from the **employee** table

  Ans: (Hint *** means everything/all the columns**)

  ```
  SELECT *

  FROM employee

  WHERE employee_name = 'John Doe'
  ```
  – (*remember, if it's a string we have to put it under single quotations)*

Data Modification in a Table

Syntax:

```
UPDATE <table name>
SET <attribute name> = <new value>
WHERE <someCondition_on_column_values>;
```

Example:

Update department
set budget = 1500000
where dept_name = 'CSE';

Data Deletion from a Table

Syntax:

```
DELETE FROM <table name>
WHERE <someCondition>;
```

Example:
Delete from department
Where budget<10000;

Conditions in SQL

- **Mathematical operators** such as <; >; <=; >=; =; ! =; <>;+;−; *; ~:%
- **logical operators** such as AND, OR, NOT

example:

- Ques: Find the productNames and their stocks from the products table those have stocks greater than 100 units and MSRP is at least 50 units

  Ans:

  ```
  SELECT productNames, stocks

  FROM products

  WHERE stocks > 100 AND MSRP >= 50
  ```

Patterns in SQL

The `LIKE` operator is a logical operator that tests whether a string contains a specified pattern or not.

MySQL provides two wildcard characters for constructing patterns: percentage `%` and underscore _ .

- The **percentage (** `%` **)** wildcard matches **any string of zero or more** characters.
- The **underscore (_)** wildcard matches **any single character** at its position

The formation of % operator:

% Placed at	**pattern**	**meaning**	**example**
Beginning / Prefix	%x	anything **before** x symbol, 0 or more characters	*%sh* → a**sh**, ca**sh**, ba**sh**, ma**sh**, la**sh**, spla**sh** etc
Ending / Suffix	x%	anything **after** x symbol, 0 or more characters	*a%* → **a**, **a**pple, **a**nt, **a**bracadabra, **a**ntelope etc.
Both ends	%x%	symbol x can be **anywhere as a substring**, 0 or more characters before/after it	%on% → Jeffers**on**, M**on**ir, **On**ie, **On**ion etc.

*** where the % sign is, there can be any number of characters.*

Example:

- find employees' employeeNumber, lastName, firstName whose **last names end with** the literal string 'on'

```
SELECT employeeNumber, lastName, firstName

FROM employees

WHERE lastName LIKE '%on';
```

- find all employees employeeNumber, lastName, firstName whose **last names contain the substring** on:

```
SELECT
    employeeNumber,
    lastName,
    firstName
FROM
    employees
WHERE
    lastname LIKE '%on%';
```

The formation of _ (underscore) operator:

- Must match **both position and length**.
- Must be filled up by a **single symbol** where the underscore is (same as the fill in the gaps)

_ Placed at	**pattern**	**meaning**	**example**
Beginning / Prefix	_xxxx….. (Total n length)	Any pattern of **exact length of n** One **single symbol per underscore at the beginning**	_ am → Cam, Ram, bam, ham etc. But Scam not accepted as it exceeds the length of 3 and has two symbols for a single underscore _ _ sh → ca**sh**, ba**sh**, ma**sh**, la**sh** But Ash/ Splash not accepted as they do not match the length of 4 and has more/less symbols for two underscores
Ending / Suffix	xxx..xx_ (Total n length)	Any pattern of **exact length of n** One **single symbol per underscore at the ending**	Ri _ _ → Rise, Ripe etc.
At different places	X_x_ _X_ _ X (Total of n length)	Any pattern of **exact length of n** One **single symbol per underscore at the places of underscore**	L_m_ → Lime, Lame etc. **But Lemon not accepted** as it exceeds the length of 4 and has two symbols for a single underscore

Example:

To find employees whose first names start with the letter `T_m (eg. Tim. Tom etc)`, end with the letter `m`, and contain any single character between.

```
SELECT employeeNumber, lastName, firstName
FROM employees
WHERE firstname LIKE 'T_m';
```

Important note on % and _

- % → **position must match** but *length doesn't matter*
- _ → **position and length must match**

Join

It is used to retrieve data from multiple tables. It is performed whenever you need to fetch records from two or more tables.

Here we have at least two tables. Left Table (mentioned after FROM clause) & Right Table (mentioned after JOIN clause)

MySQL supports the following types of joins:

- Inner join
- Left join
- Right join
- Cross join

To join tables, we use the cross join, inner join, left join, or right join clause. The join clause is used in the SELECT statement appeared after the FROM clause.

Note that MySQL hasn't supported the FULL OUTER JOIN yet.

Inner Join:

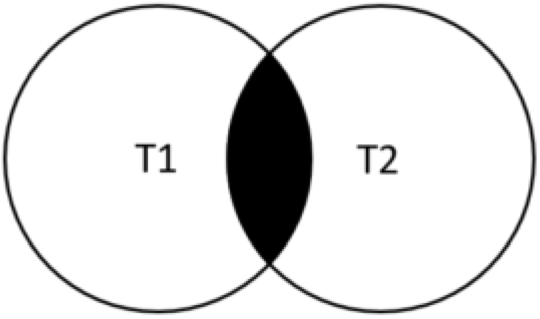
The inner JOIN is used to return rows from both tables that satisfy the given condition.

The `INNER JOIN` matches each row in one table with every row in other tables and allows you to query rows that contain columns from both tables.

```
SELECT column_names

FROM t1

INNER JOIN t2

ON join_condition;
```

Example:

```
SELECT
   productCode,
   productName,
   textDescription
FROM
   products t1
INNER JOIN productlines t2
   ON t1.productline = t2.productline;
```

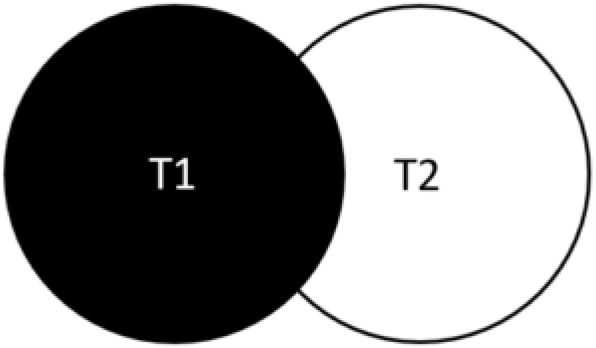


Left Join:

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
    select_list
FROM
    t1
LEFT JOIN t2
ON join_condition;
```

- t1 → Left table
- t2 → Right table



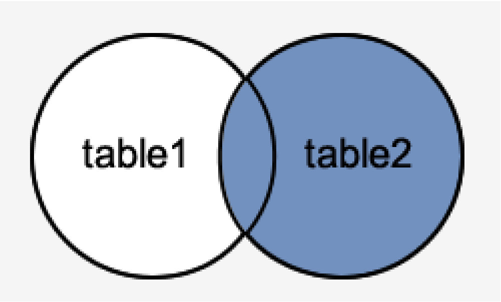
Example:

```
SELECT
   customers.customerNumber,
   customerName,
   orderNumber
FROM
   customers
LEFT JOIN orders ON orders.customerNumber = customers.customerNumber;
```

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.**

```
SELECT select_list

FROM t1

RIGHT JOIN t2

ON join_condition;
```



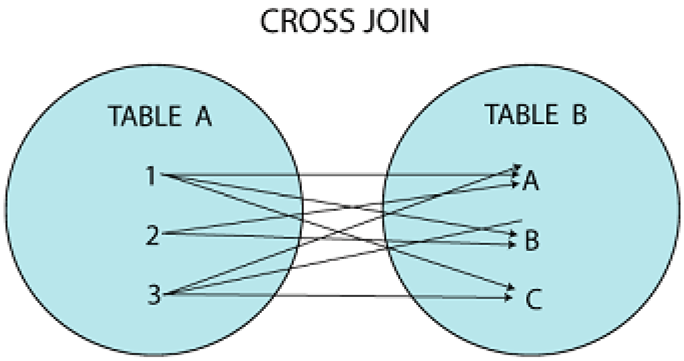
Example:

```
SELECT

   employeeNumber,

   customerNumber

FROM

   customers

RIGHT JOIN employees

   ON salesRepEmployeeNumber = employeeNumber;
```

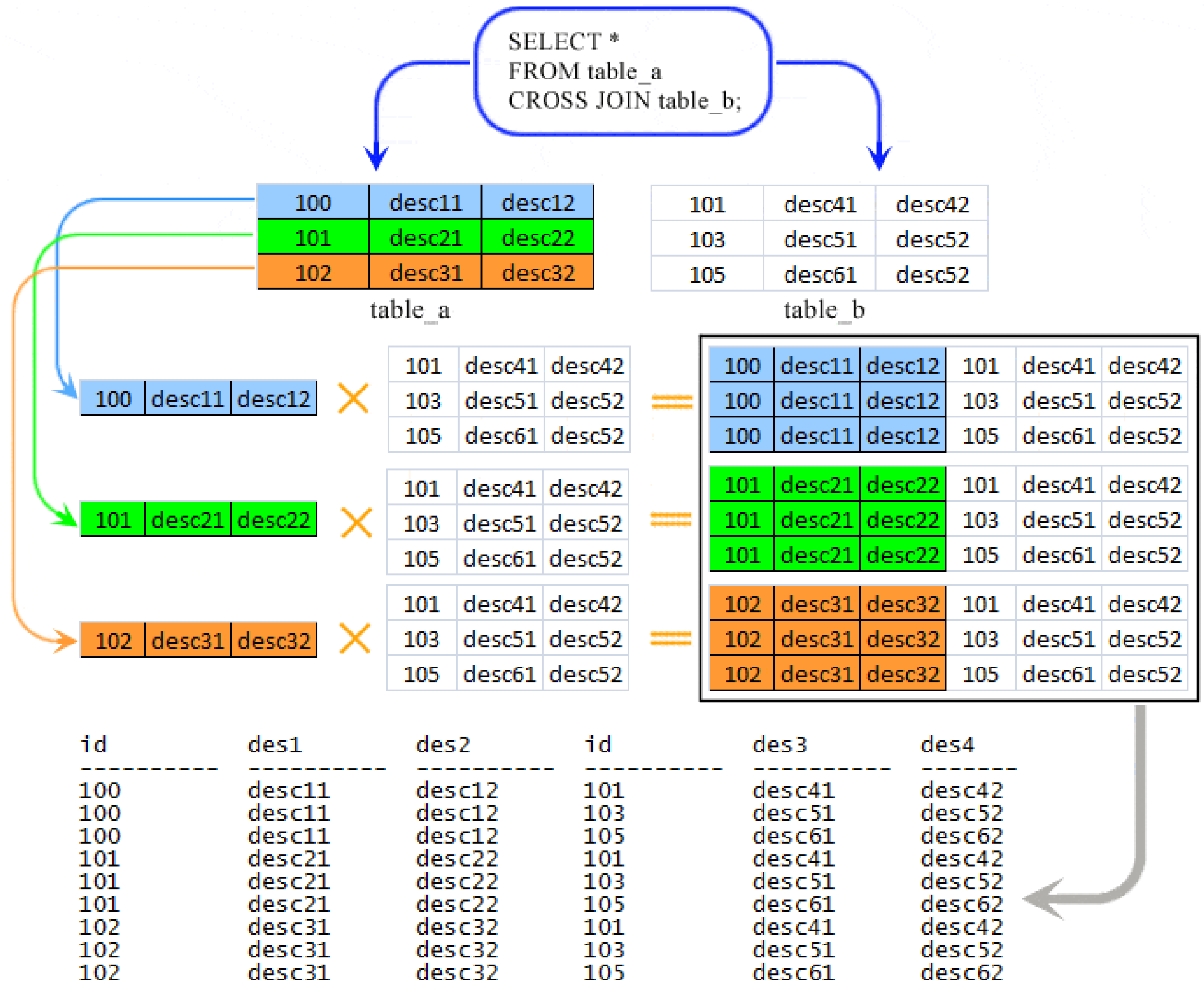
Cross Join:

The result set will include all rows from both tables, where each row is the combination of the row in the first table with the row in the second table. In general, if each table has n and m rows respectively, the result set will have n x m rows.

In other words, the CROSS JOIN clause ***returns a Cartesian product of rows from the joined tables.***



```
SELECT *
FROM t1
CROSS JOIN t2;
```



```
SELECT *
FROM table_a
CROSS JOIN table_b;
```

100	desc11	desc12
101	desc21	desc22
102	desc31	desc32

table_a

101	desc41	desc42
103	desc51	desc52
105	desc61	desc52

table_b

100	desc11	desc12

101	desc41	desc42
103	desc51	desc52
105	desc61	desc52

101	desc21	desc22

101	desc41	desc42
103	desc51	desc52
105	desc61	desc52

102	desc31	desc32

101	desc41	desc42
103	desc51	desc52
105	desc61	desc52

100	desc11	desc12	101	desc41	desc42
100	desc11	desc12	103	desc51	desc52
100	desc11	desc12	105	desc61	desc52
101	desc21	desc22	101	desc41	desc42
101	desc21	desc22	103	desc51	desc52
101	desc21	desc22	105	desc61	desc52
102	desc31	desc32	101	desc41	desc42
102	desc31	desc32	103	desc51	desc52
102	desc31	desc32	105	desc61	desc52

```
id         des1       des2       id         des3       des4
---------- ---------- ---------- ---------- ---------- -------
100        desc11     desc12     101        desc41     desc42
100        desc11     desc12     103        desc51     desc52
100        desc11     desc12     105        desc61     desc62
101        desc21     desc22     101        desc41     desc42
101        desc21     desc22     103        desc51     desc52
101        desc21     desc22     105        desc61     desc62
102        desc31     desc32     101        desc41     desc42
102        desc31     desc32     103        desc51     desc52
102        desc31     desc32     105        desc61     desc62
```

Aggregate Functions

An aggregate function performs a **calculation on multiple values and returns a single value.**

Most commonly used aggregate functions are

- AVG () calculates the average of a set of values.
- COUNT () counts rows in a specified table or view.
- MIN () gets the minimum value in a set of values.
- MAX () gets the maximum value in a set of values.
- SUM () calculates the sum of values.

Example:

SELECT MAX (salary)

FROM employees

GROUP BY:

The `GROUP BY` clause groups a set of rows into a set of summary rows by values of columns or expressions. The `GROUP BY` clause returns one row for each group. In other words, it reduces the number of rows in the result set.

The MySQL GROUP BY clause is used in a SELECT statement to **collect data across multiple records and group the results by one or more columns.**

The `GROUP BY` statement is often used with aggregate functions (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`) to group the result-set by one or more columns.

GROUP BY is by default inherits DISTINCT () [MySQL commands to find only the non-duplicate values] and doesn’t return any NULL values.

Syntax:

```
SELECT c1, c2,..., cn,aggregate_function(ci)

FROM table

WHERE some_conditions

GROUP BY c1, c2,…,cn;
```

Example:

SELECT status, COUNT(*)

FROM orders

GROUP BY status;

HAVING:

The `HAVING` clause is used in the `SELECT` statement to specify filter conditions for a group of rows or aggregates.

The `HAVING` clause is often used with the `GROUP BY` clause to filter groups based on a specified condition.

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot be used with aggregate functions.

```
SELECT select_list

FROM table_name

WHERE search_condition

GROUP BY group_by_expression

HAVING group_condition_or_aggrgate_functions;
```

Example:

Q. Find those employees from **each department** whose salary is **greater than** his/her department's **average salary.**

Ans:

*SELECT first_name, dept_id**, AVG (salary)***

FROM employees

GROUP BY dept_id

*HAVING **AVG (salary)** > 1000;*

Points to remember:

- **HAVING can only take conditions on aggregate functions.**
- WHERE can all other conditions.
- Read more from here: MySQL - WHERE vs HAVING

ORDER BY:

The MySQL ORDER BY clause is used to **sort the records** in the result set.

```
SELECT select_list

FROM table_name

ORDER BY

   column1 [ASC|DESC],

   column2 [ASC|DESC],

   ...;
```

ASC → sort the resultant table using **Ascending order**

DESC → sort the resultant table using **Descending order**

<u>Example:</u>

```
SELECT

  contactLastname,

  contactFirstname

FROM

  customers

ORDER BY

        contactLastname DESC,

        contactFirstname ASC;
```

LIMIT

The `LIMIT` clause is used in the `SELECT` statement to constrain the number of rows to return. The `LIMIT` clause accepts one or two arguments. The values of both arguments must be zero or positive integers.

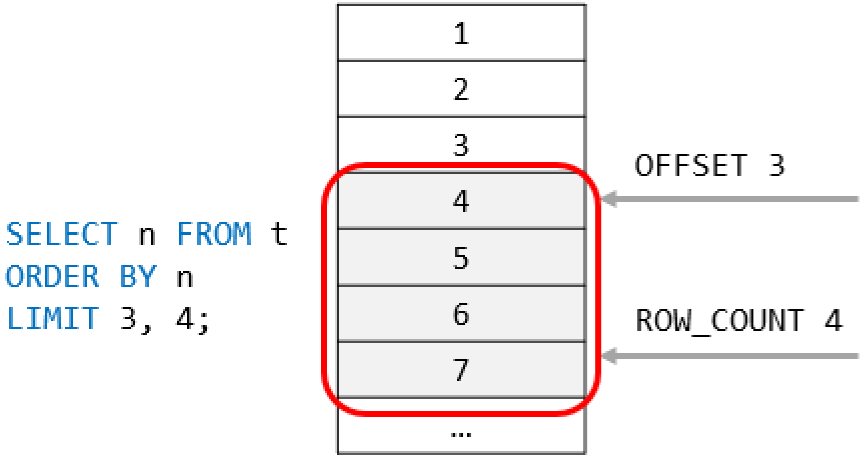
The following illustrates the `LIMIT` clause syntax with two arguments:

```
SELECT select_list

FROM table_name

LIMIT [offset,] row_count;
```

In this syntax:

- The `offset` specifies the offset of the first row to return. The `offset` of the first row is 0, not 1.
- The `row_count` specifies the maximum number of rows to return.

The following picture illustrates the `LIMIT` clause:



```
SELECT n FROM t
ORDER BY n
LIMIT 3, 4;
```

When you use the **`LIMIT` clause with one argument (e.g.: `LIMIT` 5)**, MySQL will use this argument to **determine the maximum number of rows (here first five rows) to return from the first row of the result set.**

Therefore, these two clauses are equivalent:

```
LIMIT row_count;
```

And

```
LIMIT 0, row_count;
```

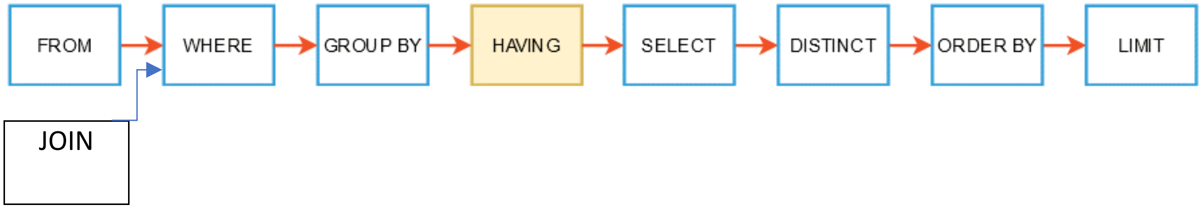
Example:

```
SELECT
  customerNumber,
  customerName,
  creditLimit
FROMcustomers
ORDER BY creditLimit DESC
LIMIT 5;
```

MYSQL query **Writing** orders:

```
SELECT columns_names

FROM table_name

WHERE search_condition

GROUP BY group_by_expression

HAVING group_condition_or_aggrgate_functions

ORDER BY column_name ASC|DESC

LIMIT row_count
```

MYSQL query **Execution Flow** Order:

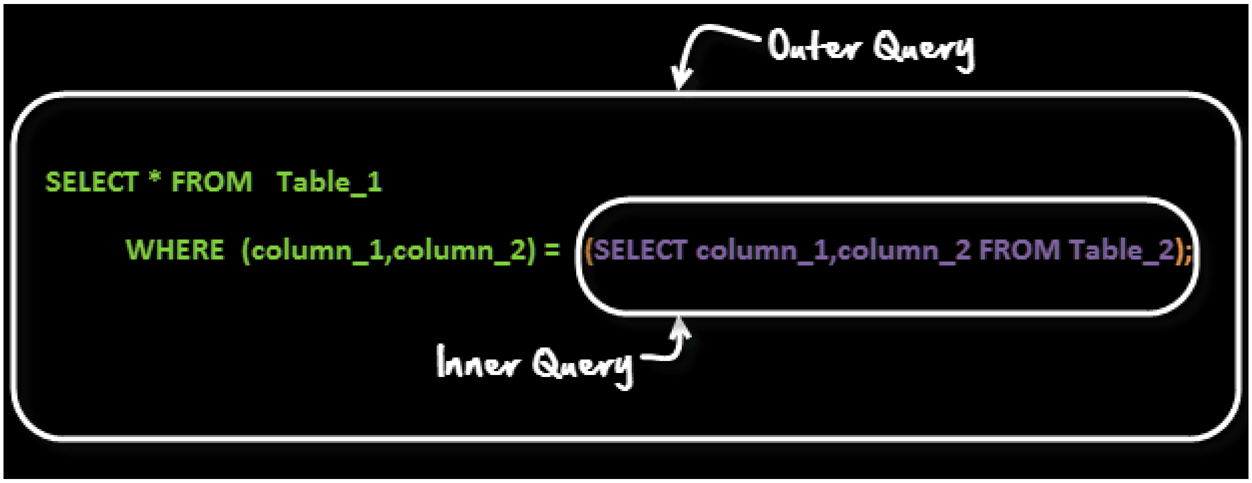


MYSQL Subqueries:

A MySQL subquery is called an inner query while the query that contains the subquery is called an outer query. A subquery can be used anywhere that expression is used and must be closed in parentheses.

- In MySQL, a subquery is also called an INNER QUERY or INNER SELECT.
- In MySQL, the main query that contains the subquery is also called the OUTER QUERY or OUTER SELECT.

The inner select query is usually used to determine the results of the outer select query.



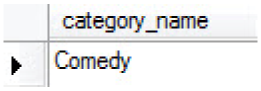
```
SELECT * FROM   Table_1
     WHERE  (column_1,column_2) = (SELECT column_1,column_2 FROM Table_2);
```

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

You can use a query like

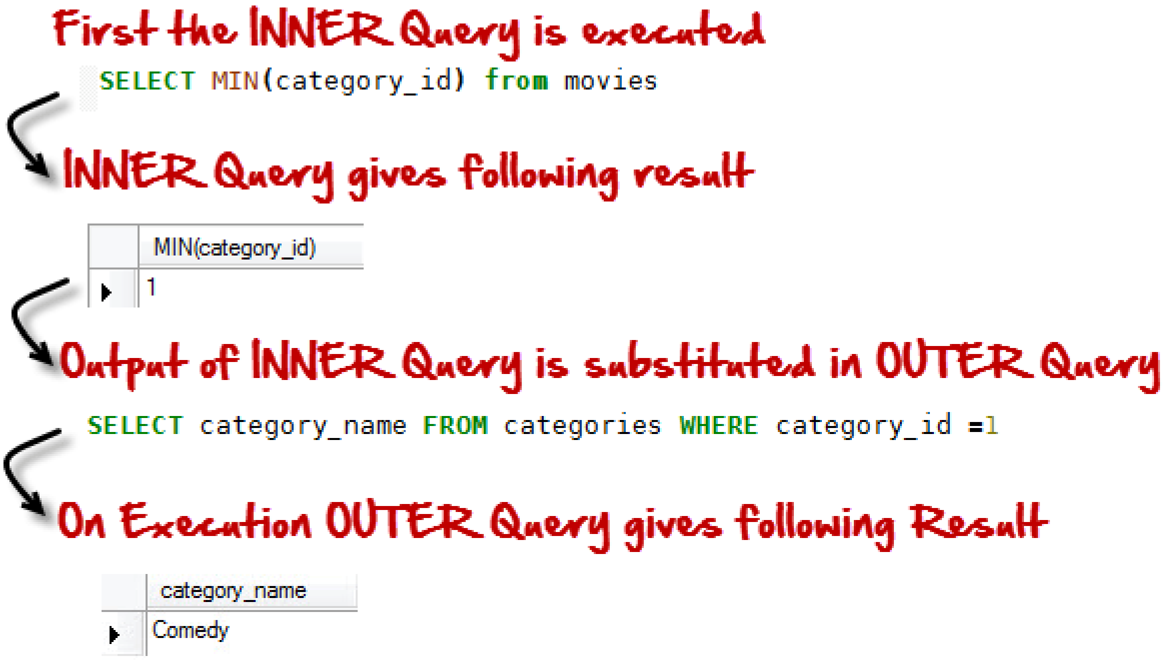
SELECT category_name FROM categories WHERE category_id =(SELECT MIN(category_id) from movies);

It gives a result



	category_name
▶	Comedy

Let's see how this query works



First the INNER Query is executed

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

INNER Query gives following result

	MIN(category_id)
▶	1

Output of INNER Query is substituted in OUTER Query

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

On Execution OUTER Query gives following Result

	category_name
▶	Comedy

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

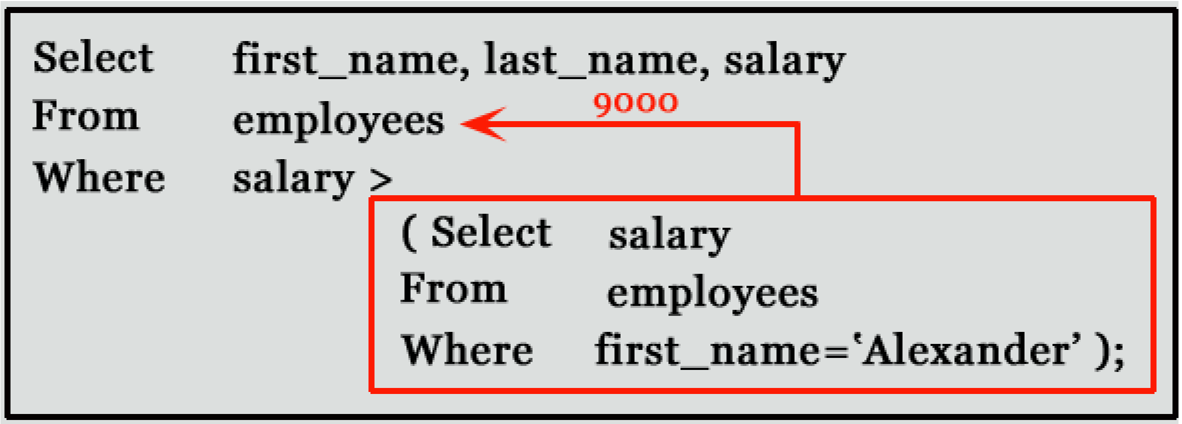
[Sources: https://www.guru99.com/sub-queries.html]

Subqueries: Guidelines

There are some guidelines to consider when using subqueries:
- A subquery must be enclosed in parentheses.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.
- If a subquery (inner query) returns a null value to the outer query, the outer query will not return any rows when using certain comparison operators in a WHERE clause.

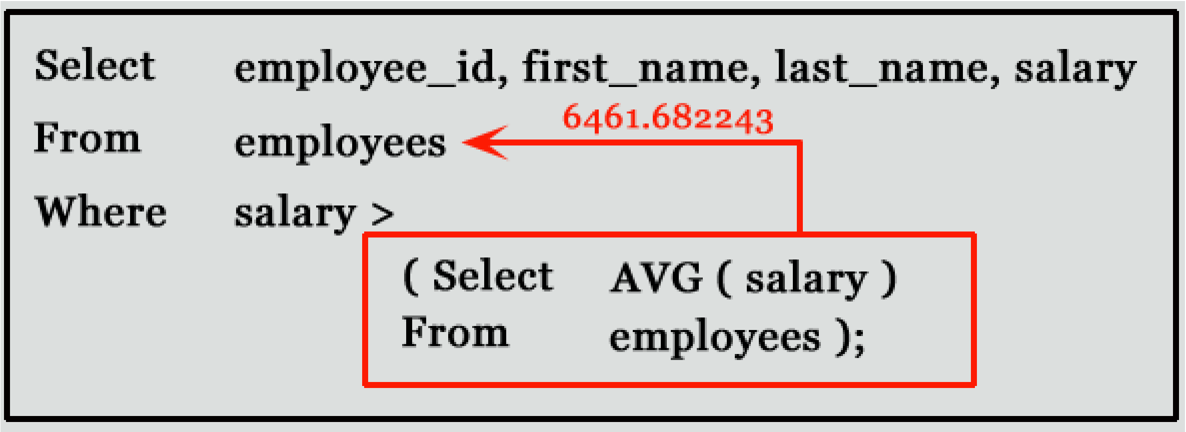
MySQL Subquery Example:

- Using a subquery, list the name of the employees, paid more than 'Alexander' from emp_details of the hr_schema database.



```
Select     first_name, last_name, salary
From       employees
Where      salary >
                ( Select    salary
                  From      employees
                  Where     first_name='Alexander' );
```

- Suppose you want to find the employee id, first_name, last_name, and salaries for employees whose average salary is higher than the average salary throughout the company.



```
Select     employee_id, first_name, last_name, salary
From       employees
Where      salary >
                ( Select    AVG ( salary )
                  From      employees );
```

For a more detailed explanation: https://youtu.be/hBF5PO1fD0Q