Hello everyone and welcome to this video. I hope you are all doing good. Today we will start with another MySQL project as discussed in our last session. First we will talk about the database that we will be developing and how the tables of the database are connected with each other. We will also understand the records present in each table and the logic it represents . First let’s have a look at the structure of the database in the below diagram.

The database represents a company , it’s branches and it’s clients as a whole.

For PPT –

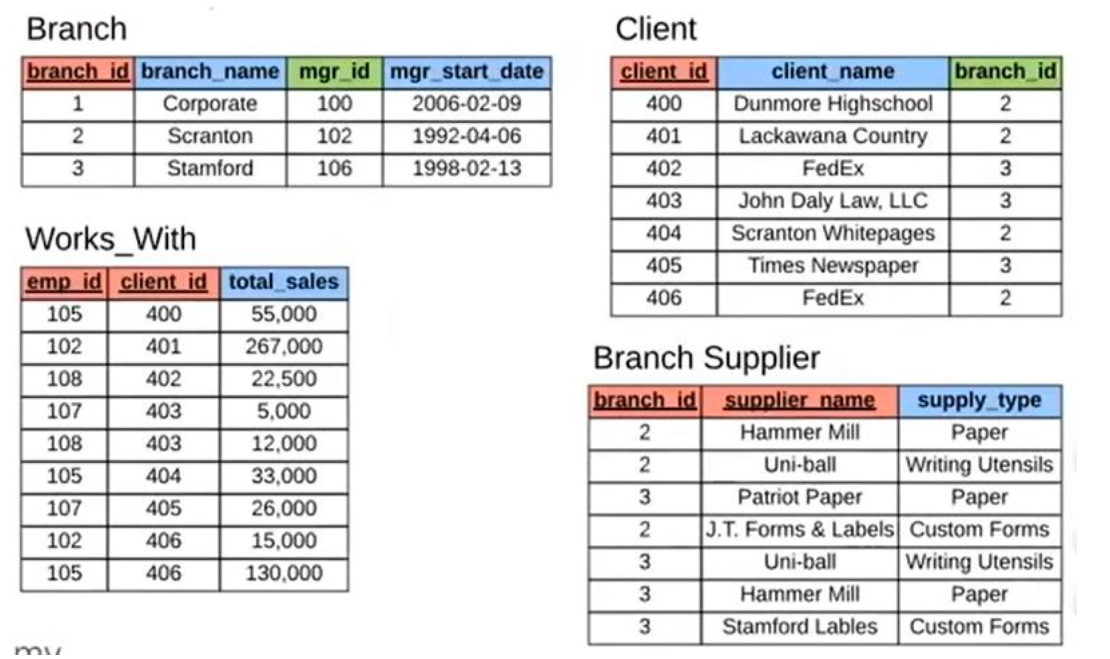
Database Name – Companydatabase

**Blue Columns** – Attributes

**Pink Columns** – Primary Key

**Green Columns** – Foreign Key





Before we jump into the code, let’s understand each table first and what it represents , In the employee table as it can be seen emp\_id is present , which is our primary key for the specific table. Now what is a primary key ? , let’s have a look –

For PPT –

Primary Key –

The PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only ONE primary key

and in the table, this primary key can consist of single or multiple columns (fields).

As from the definition given it can easily be understood that emp\_id fits perfectly into the description of the same. Then let’s talk about the columns marked in blue , these are all called attributes , in MySQL there can be many type of attributes present , let’s try to understand them.

For PPT –

* Attributes –
* Simple attribute :
* Composite attribute :
* Single-valued attribute :
* Multi-valued attribute
* Derived attribute
* Stored attribute
* Key attribute
* Complex attributes

An attribute is a property or characteristic of an entity. An entity may contain any number of attributes. One of the attributes is considered as the primary key. In an Entity-Relation model, attributes are represented in an elliptical shape.

Example: Student has attributes like name, age, roll number, and many more. To uniquely identify the student, we use the primary key as a roll number as it is not repeated. Attributes can also be subdivided into another set of attributes.

There are six such types of attributes: Simple, Composite, Single-valued, Multi-valued, and Derived attribute. One more attribute is their, i.e. Complex Attribute, this is the rarely used attribute.

Simple attribute :

An attribute that cannot be further subdivided into components is a simple attribute.

Example: The roll number of a student, the id number of an employee.

Composite attribute :

An attribute that can be split into components is a composite attribute.

Example: The address can be further split into house number, street number, city, state, country, and pin code, the name can also be split into first name middle name, and last name.

Single-valued attribute :

The attribute which takes up only a single value for each entity instance is a single-valued attribute.

Example: The age of a student.

Multi-valued attribute :

The attribute which takes up more than a single value for each entity instance is a multi-valued attribute.

Example: Phone number of a student: Landline and mobile.

Derived attribute :

An attribute that can be derived from other attributes is derived attributes.

Example: Total and average marks of a student.

Complex attribute :

Those attributes, which can be formed by the nesting of composite and multi-valued attributes, are called “Complex Attributes“. These attributes are rarely used in DBMS(DataBase Management System). That’s why they are not so popular.

Stored attribute:

The stored attribute are those attribute which doesn’t require any type of further update since they are stored in the database.

Example: DOB(Date of birth) is the stored attribute.

Key attribute:

Key attributes are those attributes that can uniquely identify the entity in the entity set.

Example: Roll-No is the key attribute because it can uniquely identify the student.

Representation:

Complex attributes are the nesting of two or more composite and multi-valued attributes. Therefore, these multi-valued and composite attributes are called ‘Components’ of complex attributes.

These components are grouped between parentheses ‘( )’ and multi-valued attributes between curly braces ‘{ }’, Components are separated by commas ‘, ‘.

For example: let us consider a person having multiple phone numbers, emails, and an address.

Here, phone number and email are examples of multi-valued attributes and address is an example of the composite attribute, because it can be divided into house number, street, city, and state.

Complex attributes

Components:

Email, Phone number, Address(All are separated by commas and multi-valued components are represented between curly braces).

Complex Attribute: Address\_EmPhone(You can choose any name).

While speaking about attributes , we can see many attributes are present here in the table , some in string format , some in numeric and some dates are are written in. These are called data type in MySQL , let’s see how many data types we have in here.

For PPT -

**String Data Types**

**Data type Description**

CHAR(size) A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1

VARCHAR(size) A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum column length in characters - can be from 0 to 65535

BINARY(size) Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1

VARBINARY(size) Equal to VARCHAR(), but stores binary byte strings. The size parameter specifies the maximum column length in bytes.

TINYBLOB For BLOBs (Binary Large OBjects). Max length: 255 bytes

TINYTEXT Holds a string with a maximum length of 255 characters

TEXT(size) Holds a string with a maximum length of 65,535 bytes

BLOB(size) For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data

MEDIUMTEXT Holds a string with a maximum length of 16,777,215 characters

MEDIUMBLOB For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data

LONGTEXT Holds a string with a maximum length of 4,294,967,295 characters

LONGBLOB For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data

ENUM(val1, val2, val3, ...) A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them

SET(val1, val2, val3, ...) A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

**Numeric Data Types**

**Data type Description**

BIT(size) A bit-value type. The number of bits per value is specified in size. The size parameter can hold a value from 1 to 64. The default value for size is 1.

TINYINT(size) A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies the maximum display width (which is 255)

BOOL Zero is considered as false, nonzero values are considered as true.

BOOLEAN Equal to BOOL

SMALLINT(size) A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255)

MEDIUMINT(size) A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter specifies the maximum display width (which is 255)

INT(size) A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)

INTEGER(size) Equal to INT(size)

BIGINT(size) A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)

FLOAT(size, d) A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions

FLOAT(p) A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()

DOUBLE(size, d) A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter

DOUBLE PRECISION(size, d)

DECIMAL(size, d) An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0.

DEC(size, d) Equal to DECIMAL(size,d)

**Date and Time Data Types**

**Data type Description**

DATE A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'

DATETIME(fsp) A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time

TIMESTAMP(fsp) A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP in the column definition

TIME(fsp) A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'

YEAR A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

MySQL 8.0 does not support year in two-digit format.

Now that we have a clarity on data type and attributes in general let’s move forward to the green coloured columns . As mentioned above , these are foreign keys , like primary keys , foreign keys also helps us to interconnect the present tables , it takes reference from a primary key in another table and connects the same

For PPT –

Foreign Key –

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables. A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

Whatever we have discussed above will be applicable for rest of the tables as well , Null as you can see in the records , it represents the missing/null values in the data. Few more observations that we can understand by carefully observing the data is as this –

1. David Wallace here is the manage at the top level and he doesn’t have a superior present in the data.
2. David and Jan both work in corporate
3. Despite being a branch manager , Michael has sales record present in his name , which josh doesn’t although he is also a branch manager at Stamford.
4. Same client can purchase from two different branches , in here clients were also making purchases based on their location . Eg : Fedex.
5. Same vendors can also provide supplies to different branches.

Now let’s go ahead and look into the code of the database development –

In here we are creating a table called employees , where emp\_id stands to be our primary key. Which means it should be unique and should not contain any null value in it. First and last both names and sex as explained are character in terms of data type and we have limited it’s length to 40 character. Birth date data type as can be seen is given in Date format. Now the salary , super\_id and branh\_id all are Integer in terms of data type , but super\_id and branch id is acting as a foreign key here. The super id is taking reference from the emp\_id from this very same table and the branch\_id will take reference from the branch table where branch id will work as primary key.

CREATE TABLE employee ( emp\_id INT PRIMARY KEY, first\_name VARCHAR(40), last\_name VARCHAR(40), birth\_day DATE, sex VARCHAR(1), salary INT, super\_id INT, branch\_id INT );

 As stated in above while creating the branch table , branch\_id is taken as the primary key. While most of the code is self-explanatory , the assigning of foreign key is to be detailed here. Manager id (mgr\_id) is being treated here as the foreign key which is taking reference from the emp\_id from the employee table. On delete set null in here states that if value from the table and column from where the reference is taken , if deleted , the corresponding value in the foreign key table will be set as null.

CREATE TABLE branch ( branch\_id INT PRIMARY KEY, branch\_name VARCHAR(40), mgr\_id INT, mgr\_start\_date DATE, FOREIGN KEY(mgr\_id) REFERENCES employee(emp\_id) ON DELETE SET NULL );

 As previously branch table was not created , hence now after creating of branch table , we are modifying our employee table where we are adding the branch\_id as a foreign key and stating that it will take references from branch id of branch table. And also setting super\_id as a foreign key here stating that it’ll take reference from emp\_id column of employee table itself. On delete set null in here states that if value from the table and column from where the reference is taken if deleted , the corresponding value in the foreign key table will be set as null.

ALTER TABLE employee ADD FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE SET NULL;

ALTER TABLE employee ADD FOREIGN KEY(super\_id) REFERENCES employee(emp\_id) ON DELETE SET NULL;

 The tables below mostly follow the rules stated above. The difference lies in two phases. In the table ‘works\_with’ has two primary keys or composite primary keys ,emp\_id and client\_id , because to determine the value of sales and to which individual it’s aligned to , we need information from both client and employee , as an example in the table if we get a sale value , with only emp\_id we can determine which employee sold it and with client\_id we can only determine which client purchased it. But with both of them working as a composite primary key , we will be able to determine , who sold to which client and for how much value in sales. Same concept works for the branch supplier table where both branch\_id and supplier name is working as composite primary key which is determining and explaining the supply type. Assigning foreign key and it’s references are same as above. There is another difference while setting the on delete section. On delete cascade means , if from the base table from where the foreign keys are taking reference from ,values are deleted then the values in the foreign key section will also be deleted instead of setting it’s value as null.

CREATE TABLE client ( client\_id INT PRIMARY KEY, client\_name VARCHAR(40), branch\_id INT, FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE SET NULL );

CREATE TABLE works\_with ( emp\_id INT, client\_id INT, total\_sales INT, PRIMARY KEY(emp\_id, client\_id), FOREIGN KEY(emp\_id) REFERENCES employee(emp\_id) ON DELETE CASCADE, FOREIGN KEY(client\_id) REFERENCES client(client\_id) ON DELETE CASCADE );

CREATE TABLE branch\_supplier ( branch\_id INT, supplier\_name VARCHAR(40), supply\_type VARCHAR(40), PRIMARY KEY(branch\_id, supplier\_name), FOREIGN KEY(branch\_id) REFERENCES branch(branch\_id) ON DELETE CASCADE );

Adding Value in the Tables –

We have three branch specified in our data , corporate , Stamford and Scranton. As the managers of the branches are connected with branch table , hence while adding the value for these managers we will set the respective foreign key values as null , and later on will update the table based on given connections. Super id in the employee table refers to the immediate supervisor for that particular employee , and as David Wallace is the top boss here hence he doesn’t have any supervisor , so super\_id row for David Wallace will be set as null.

-- -----------------------------------------------------------------------------

-- Corporate INSERT INTO employee VALUES(100, 'David', 'Wallace', '1967-11-17', 'M', 250000, NULL, NULL);

INSERT INTO branch VALUES(1, 'Corporate', 100, '2006-02-09');

UPDATE employee SET branch\_id = 1 WHERE emp\_id = 100;

INSERT INTO employee VALUES(101, 'Jan', 'Levinson', '1961-05-11', 'F', 110000, 100, 1);

-- Scranton INSERT INTO employee VALUES(102, 'Michael', 'Scott', '1964-03-15', 'M', 75000, 100, NULL);

INSERT INTO branch VALUES(2, 'Scranton', 102, '1992-04-06');

UPDATE employee SET branch\_id = 2 WHERE emp\_id = 102;

INSERT INTO employee VALUES(103, 'Angela', 'Martin', '1971-06-25', 'F', 63000, 102, 2); INSERT INTO employee VALUES(104, 'Kelly', 'Kapoor', '1980-02-05', 'F', 55000, 102, 2); INSERT INTO employee VALUES(105, 'Stanley', 'Hudson', '1958-02-19', 'M', 69000, 102, 2);

-- Stamford INSERT INTO employee VALUES(106, 'Josh', 'Porter', '1969-09-05', 'M', 78000, 100, NULL);

INSERT INTO branch VALUES(3, 'Stamford', 106, '1998-02-13');

UPDATE employee SET branch\_id = 3 WHERE emp\_id = 106;

INSERT INTO employee VALUES(107, 'Andy', 'Bernard', '1973-07-22', 'M', 65000, 106, 3); INSERT INTO employee VALUES(108, 'Jim', 'Halpert', '1978-10-01', 'M', 71000, 106, 3);

-- BRANCH SUPPLIER INSERT INTO branch\_supplier VALUES(2, 'Hammer Mill', 'Paper'); INSERT INTO branch\_supplier VALUES(2, 'Uni-ball', 'Writing Utensils'); INSERT INTO branch\_supplier VALUES(3, 'Patriot Paper', 'Paper'); INSERT INTO branch\_supplier VALUES(2, 'J.T. Forms & Labels', 'Custom Forms'); INSERT INTO branch\_supplier VALUES(3, 'Uni-ball', 'Writing Utensils'); INSERT INTO branch\_supplier VALUES(3, 'Hammer Mill', 'Paper'); INSERT INTO branch\_supplier VALUES(3, 'Stamford Labels', 'Custom Forms');

-- CLIENT INSERT INTO client VALUES(400, 'Dunmore Highschool', 2); INSERT INTO client VALUES(401, 'Lackawana Country', 2); INSERT INTO client VALUES(402, 'FedEx', 3); INSERT INTO client VALUES(403, 'John Daly Law, LLC', 3); INSERT INTO client VALUES(404, 'Scranton Whitepages', 2); INSERT INTO client VALUES(405, 'Times Newspaper', 3); INSERT INTO client VALUES(406, 'FedEx', 2);

-- WORKS\_WITH INSERT INTO works\_with VALUES(105, 400, 55000); INSERT INTO works\_with VALUES(102, 401, 267000); INSERT INTO works\_with VALUES(108, 402, 22500); INSERT INTO works\_with VALUES(107, 403, 5000); INSERT INTO works\_with VALUES(108, 403, 12000); INSERT INTO works\_with VALUES(105, 404, 33000); INSERT INTO works\_with VALUES(107, 405, 26000); INSERT INTO works\_with VALUES(102, 406, 15000); INSERT INTO works\_with VALUES(105, 406, 130000);

Now let’s go ahead and run some queries on the database itself to extract different category of information

With the help of distinct syntax we can find unique values present in the data based on a specific column. Suppose we want to find the unique branch code or branch id present in the employee table. The code will look like this –

SELECT DISTINCT branch\_id FROM employee;

To find the number of employees , we can use the count syntax. Now we all know that the number of employee id present in the table will reflect the number of employees. So if we count the distinct number count of employee id , we will get our result.

SELECT DISTINCT COUNT(emp\_id) FROM employee;

Suppose we want to find that how many of the employees are having supervisors , as we know super\_id column is for the supervisor emp\_id , so if we check in accordance with where the super id is present or the value of it is not null , then those employees must have a supervisor. The code will look like below.

Approach 1 – SELECT DISTINCT COUNT(emp\_id) FROM employee WHERE super\_id IS NOT NULL;

Approach 2 – SELECT DISTINCT COUNT(super\_id) FROM employee;

Now we are trying to find out the number of female employees , born after 1970. The code is self explanatory and looks like as below –

SELECT COUNT(emp\_id) FROM employee WHERE sex = 'F' AND birth\_day > '1970-01-01' ;

 Now let’s try to find out the average salary for male and female employees in the employee table.

SELECT AVG(salary) FROM employee WHERE sex = 'F';

SELECT AVG(salary) FROM employee WHERE sex = 'M';

 To find the sum of salaries for all employee –

SELECT SUM(salary) FROM employee ;

 Aggregate functions – We can use the aggregate functions to find out the number of male and female employees present in the database. The code will first display the sex nature(M or F) , then the count of each sex category grouped by sex.

SELECT sex , COUNT(emp\_id) FROM employee GROUP BY sex;

 Another approach - We can use aggregate function to find out how many male and female employees are there in our table. We need to count the total number of rows present in sex column using the count function , then we will select the sex column as well to display if the sex is m or f . At the very end , we will use group by in our query to display the desired result. The query would look like as below ,

SELECT COUNT(sex) , sex FROM employee GROUP BY sex;

 We can find out how much one individual employee has sold with aggregate function as well. The code will look like below , where we are trying to find out how much the employee with emp\_id 105 has sold.

SELECT emp\_id , SUM(total\_sales) FROM works\_with WHERE emp\_id=105

 Same we can find out this for all the employees that how much they have sold , the query would look like as below ,

SELECT emp\_id , SUM(total\_sales) FROM works\_with GROUP BY emp\_id;

 We can join the works with table with the employee table to find out the total sales made by each employee. The reason why we are joining these tables are because we want to display the employee names as well , the employee names are present in the employee table , The emp\_id will work as the primary key here based on which we will be joining both the tables. The code will look as below. Total sales column is renames as TS with the help of As syntax , the works with and the employee tables are taken with the help of aliases as such w and e respectively.

SELECT first\_name , last\_name , SUM(total\_sales) AS TS FROM works\_with w JOIN employee e ON w.emp\_id = e.emp\_id GROUP BY first\_name ORDER BY TS DESC ;

Now lets try to find out how much the clients paid to the company , we will use the same syntax what we used while trying to find out the total sales made by a salesman here. The code will look like as below.

SELECT client\_id , SUM(total\_sales) FROM works\_with GROUP BY client\_id;

 Now just like before we will use the join function to display the client names alongside the sales volume from each of them. The Query code will be like below .

SELECT client\_name , SUM(total\_sales) AS TS FROM works\_with w JOIN client c ON w.client\_id = c.client\_id GROUP BY client\_name ORDER BY TS DESC;

Wildcard Functions –

 Wildcard characters are used to find a pattern in our databases. ‘LIKE’ operator is used to denote wildcard characters. ‘%’ is a wildcard character , as an example ‘%pvt’ means we are trying find names that ends with ‘pvt’ , ‘%’ this denotes that any number of character can come before ‘pvt’ in the name to be displayed by ‘pvt’ should be at the end of it. As an example let’s trying to find out the client names which ends with ‘ LLC ‘.

SELECT \* FROM client WHERE client\_name LIKE '%LLC';

 Now we are trying to find all the branch suppliers who are in label business. The query code will look like below ,

SELECT \* FROM branch\_supplier WHERE supplier\_name LIKE '%Labels';

 Now please note , by using ‘%’ symbol twice , we can provide arguments inside these wildcard characters which denotes that the word present in between two % , can be present anywhere in the name , not necessarily in the beginning or at the end. Let’s try to find out every employee who’s first or last name contains the letter ‘N’.

SELECT \* FROM employee WHERE first\_name LIKE '%N%' OR last\_name LIKE '%N%' ;

 Now lets talk about another wildcard character ‘\_’ . This wildcard represents one character , that meane one \_ means one character , \_\_ means two , \_\_\_ means three etc. Now our goal here is to find out employees born in October. In the employee table we have seen that the data is placed like ‘ YYYY-MM-DD’. Now in the year section there are four characters , then we have a hifen and then the respective month. As we know the year can be anything , we will use ‘\_\_\_\_’ here representing any four character , then ‘-‘ and then the number 10 as it represents the month October. And at the end we will use ‘%’ wildcard as it signifies that after these characters and the desired month , anything can be present on the row value. The code will look like as below .

SELECT \* FROM employee WHERE birth\_day LIKE '\_\_\_\_-10%';

UNION Statements –

 With the help of UNION operator we can combine multiple select statements together. There are few conditions to use UNION statements. Such as –

1. All the select statements should have same number of columns selected for display.

2. All columns selected should be of same data type.

3. Generally the result that we get will display a column name based on the first select statement we used while combining the statements. We can change it with AS syntaxes.

A demo query code that will give us employee first name and branch names and client names in a single column is given as below.

SELECt first\_name AS combined\_details FROM employee UNION SELECT branch\_name FROM branch UNION SELECT client\_name FROM client ;

 Now let’s try a different one and try to find out all the money spent or earned by the company. Now as per our given tables , salary can be counted as a column that stores all the money spent by the company and sales can be a column that stores all the money earned by the company. The query code will look like as below

SELECT SUM(total\_sales) AS money\_details FROM works\_with UNION SELECT SUM(salary) AS money\_spent FROM employee

 Now let’s look at a bit complex query. By employee justification we mean , the net profit earned by that individual , can be calculated as (sales made – salary received ) by the employee. Let’s join employee and works with table to find out the required details. The code would be like below ,

SELECT first\_name , last\_name , SUM(total\_sales)-SUM(salary) AS emp\_justification FROM employee e JOIN works\_with w ON e.emp\_id = w.emp\_id GROUP BY first\_name ORDER BY emp\_justification DESC;

 Now lets have a look how we can add details into our table with INSERT operator. We will add another branch ‘Buffalo’ here with a branch id. But this branch will not have a manager / manager start date so we will set that value as null. The code will look like as below –

INSERT INTO branch VALUES('4','Buffalo',NULL,NULL) ;

 Now let’s write a query to find all branches and their managers. We will join the employee and branch table based on the emp\_id column present.

SELECT e.emp\_id , e.first\_name , b.branch\_name FROM employee e JOIN branch b ON e.emp\_id = b.mgr\_id ;

 Now with the query above , let’s understand few other type of joins as well. For example left join , if we execute the query with a left join , then all of the rows present in the left table(employee table) will be displayed in the output. Which means , employee names who are not managers and thus does not have their emp id on the mgr id column will also be displayed. The code will be as below ,

SELECT e.emp\_id , e.first\_name , b.branch\_name FROM employee e LEFT JOIN branch b ON e.emp\_id = b.mgr\_id ;

 Now if we use right join here , all rows from right table (branch table) will be displayed , i:e – the buffalo branch which doesn’t contain a manager will also be displayed.

SELECT e.emp\_id , e.first\_name , b.branch\_name FROM employee e RIGHT JOIN branch b ON e.emp\_id = b.mgr\_id ;

NESTED Queries –

Now things are getting interesting. We will try to find out all employees who have done sales over 30000 to a single client. As you can see , the trick here is put one query within the bracket , that’s our nested query , That select statement within the bracket is extracting the emp\_id for the employees who sold over 30000 to a single client. Then this particular select statement is getting inside of a where statement to ensure that while running the outside select statement , we get output based on the nested query submitted here on.

SELECT e.first\_name , e.last\_name , w.total\_sales FROM employee e JOIN works\_with w ON e.emp\_id = w.emp\_id WHERE e.emp\_id IN ( SELECT DISTINCT w.emp\_id FROM works\_with w WHERE w.total\_sales>30000 ) ;

Now let’s work on a query that will determine all the clients handles by the branch which Michael scott manage. We know Michael’s ID . here as well , first in the nested segment , within the bracket , we are finding out , what branch id Michael scott manages , as we have his ID with us. Then we are using this nested query on our primary select statement to find out the client names attached to that specific branch id.

SELECT c.client\_name FROM client c WHERE c.branch\_id IN( SELECT b.branch\_id FROM branch b WHERE b.mgr\_id = 102 ) ;

I hope this provides a larger perspective on how Mysql and different query type works in it. We will end the session here and will connect for our next project.