Basics of SQL for Data Scientists

Created By	
Stakeholders	
Status	
Type	
Created	@May 5, 2022 3:54 PM
Last Edited Time	@May 20, 2022 11:33 PM
Last Edited By	

What is a Database?

A database is a collection of related information. It could be stored on a computer, on paper, on a mobile phone, in our minds, etc. Computers are the most popular way to store databases. An example of a simple database is a list of friends you are inviting on your birthday or your weekly grocery list. While the example of a complex database is: data of every passenger flown by the airline in a year from Jan 1, 2021, to Dec 31, 2021, or the database of a bank, and many more examples.

Types of database

The databases are mainly of two types: Relational databases and non-relational databases.

> The relational database structured the data into one or more tables. While non-relational databases organize data in a traditional way e.g. flexible tables, graphs, dictionaries (key-value pairs), documents, etc.
>
> The relational database is also called an SQL (structured query language) database and the non-relational database is also called a non-SQL database.

What is a query?

The query is a simple command or a small program used to access the database. The most used queries are called C.R.U.D which stands for creating, reading, updating, and deleting.

If the database is quite complex, the structure of the query becomes complex too as it becomes difficult to access a certain piece of information from the database.

E.g. Considering that you need to get certain information from google, you can easily write the information you want in the search bar and google would provide you with the related information. In the same way, if we need information from a database we need to put a query to get that information, the only difference is rather than using a simple language you need to use a dedicated database language such as SQL.

What is a DBMS?

The database management system (or DBMS) is software that controls the database. We don't access the information from the database directly, rather DBMS takes the query and gets that data from the database, and sends it to the user.

Tables and keys

As we know that the relational database uses a table to store the data. The table has rows and columns, where each column represents an attribute and each row represents an instance. For example, we can have a table of players on the Toronto raptors team. Each row represents a player and each column represents an attribute related to that particular player such as name, height, weight, hometown, number of matches played, and more.

The key or primary key is unique in nature and represents the complete information about an instance. For example in the raptor player's table, the key might be their name. Because this table is not too long, so it is okay to choose the name as a key, but for a longer table or heavy databases, the key might be a serial number, a roll number (for students), a social insurance number (for CRA database), or email id (for a website database) or an employee id (for work).

An example of the simple classroom table is shown below, where the keys are written in blue color. Each ID (or key) will fetch the all data associated with it, the complete row in this case.

Student ID	First Name	last name	Major	GPA
1	Garima	Sharma	Math	4.0
2	Adam	Johnson	English	4.1
3	Ankur	Sharma	Math	4.1
4	Steve	Thompson	Biology	4.0
5	Ashley	Wilson	Chemistry	3.9

Now, we already know about the role of the primary key. Another type of key is called a suggorant key and natural key. This is a key having no information attached to it. For example in the above table student ID is a surrogate key as it is just a unique number, having no information attached to it in the practical real world. But keys such as social insurance number, and email ID are more informative and have some information attached to them. Such keys are called natural keys: keys that have some information attached to them.

Another type of key is called a a foreign key. The foreign key is a column in the database which stores the primary keys for another database. For example in the student database, we have added an additional column showing the rank of students in the math competition. These ranks act like a foreign key and are connected to another database "results of math competition" acting as its primary key. The other table shows the database "results of math competition".

Student ID	First Name	last name	Major	GPA	Rank in math competition
1	garima	Sharma	Math	4.0	1
2	Adam	Johnson	English	4.1	4
3	Ankur	Sharma	Math	4.1	10
4	Steve	thompson	Biology	4.0	12
5	Ashley	Wilson	Chemistry	3.9	33

Rank in math competition	First name	Last Name	Score (500 max)	Teacher name
1	Garima	Sharma	498	Ms. Julie

2	Jack	Frost	495	Mr. Sharma
2	Ankur	Sharma	495	Mr. Norman
3	Jack	Daniel	486	Ms. Julie
4	Adam	Johnson	480	Mr. Norman
5	Robert	Wilson	479...as so on	Mr. Norman

The above example shows that foreign keys build a relationship between two tables. In the second table, we have another foreign key “teacher name”, which will link this database to another database.

A particular table may have more than one foreign key. The table below shows two foreign keys: rank in a math competition and team head. The column team head shows the student ID of the head of their team. For example, Adam Johnson and Ankur Sharma have a team head, Garima Sharma. And Steve Thompson has Ashley Wilson as his team head. So basically foreign keys not only relate with other tables but also establish relationships within the same table.

Student ID	First Name	last name	Major	GPA	Rank in math competition	Team head
1	Garima	Sharma	Math	4.0	1	Null
2	Adam	Johnson	English	4.1	4	1
3	Ankur	Sharma	Math	4.1	10	1
4	Steve	Thompson	Biology	4.0	12	5
5	Ashley	Wilson	Chemistry	3.9	33	Null

Another type of key is called a Composite key: this means we need more than one column to define an entry. For example, look at the table below:

Rank in math competition	First name	Last Name	Score (500 max)	Teacher name
1	Garima	Sharma	498	Ms. Julie
2	Jack	Frost	495	Mr. Sharma
2	Ankur	Sharma	495	Mr. Norman
3	Jack	Daniel	486	Ms. Julie
4	Adam	Johnson	480	Mr. Norman
5	Robert	Wilson	479...as so on	Mr. Norman

In this table, the rank and first name together makes a key and is called a composite key. The rank could not identify all entries uniquely (as we have duplicate entries for rank 2), and the first name also could not uniquely identify all entries (as we have jack as the first name multiple times in the table). But when these two columns are combined together it could define all rows uniquely and hence called a composite key.

The primary keys we have learned till now are:

- Natural keys
- surrogate keys
- Foreign keys
- composite keys

What is SQL?

SQL stands for a structured query language, it is used to interact with relational database management systems (RDBMS). We can use SQL to instruct RDBMS to do things for us such as:

- C.R.U.D (create, read, update and delete)
- Create a database
- Manage a database
- Perform admin tasks (security, insertion, deletion, backup, user management, etc)
- Design and create tables

SQL is actually is a hybrid language that combines 4 types of languages:

- Data control language: we can control who will have access to the database e.g. only the admin can delete content, employees with passwords can access the database, etc.
- Data manipulation language: we can insert, update and delete information in tables/databases.
- Data Query language: We can write a query to get a certain pieces of information from the database.
- Data definition language: we can define the structure of the database like a number of tables it will have, possible column names, and data structure of the content in columns, and more.

As we understand that SQL interacts with the database via RDBMS, but as we have so many RDBMS available in the market, the basic instruction will remain the same but we might need to change our query as per the RDBMS requirements.

What is a Query?

A query is a set of intersections fed into RDBMS to get access to certain information from the database. The database as a whole contain tons of information that is not always required. In such scenarios, we need only specific information and this is what query does for us, it fetches the specific information for us.

Softwares needed

To understand further, I need to install a RDBMS. I am installing MySQL. It is a RDBMS to interact with the databases. You may install it at the link:

Another software that I will install is PopSQL, it is a kind of text editor for writing SQL queries. Although, once MySQL is properly installed, we may use terminal to connect with databases, but it is a quite complex process in itself, hence it is recommended to use a text editor for writing SQL queries.

After installing, create a database. we may give any name to the database. I have given the name “Garima’s first Database”

Create a table in a database

Data types in SQL

We have 6 data types available to use with SQL. These are:

1. INT (Integer) : It is basically whole number.
2. DECIMAL(M,N) : It shows the exact number. M is the total number of digits you want in a number, and N is the number of digits after decimal. E.g. if I use DECIMAL(5,2) that means I can have a number having 5 digits in total and that number could have only 2 decimal places e.g. 199.07
3. VARCHAR(L) : It is the string of text of length L. If I use VARCHAR(10), that means the maximum length of the string could be 10. We can change this number as required.
4. BLOB : It stands for binary large objects, it is used to store big binary data e.g. images.
5. DATE: It's format is 'YYYY-MM-DD'
6. TIMESTAMP: It's format is 'YYYY-MM-DD HH:MM:SS'.

There might be more data types, but it depends upon the RDBMS we are using.

Create our first table and its schema

To create a table in SQL I have used the reserved words "CREATE TABLE". The exact syntax is:

CREATE TABLE name ( );

The reverse words "CREATE NAME" is intentionally uppercased so that we can differentiate between SQL commands and normal text. Otherwise also if we use lower case "create table" , it would work.

"name" : it is the name you wish to provide to your database. Let's take the name "student"

Every command in SQL ends with a semicolon - always remember this.

The definition of the table is declared or written within this parenthesis. we can define the columns of the table as discussed above. The column titles are student_id, first name, last name, major, and GPA. The other two things to declare are data type and primary key (if applicable). Let's define our table called "student" in popSQL.

```
CREATE TABLE student (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(20),
    last_name VARCHAR(20),
    major VARCHAR(20),
    gpa DECIMAL(4,2)
);
```

Make sure not to use a comma after the last column title. Now our table has been created.

There is another way of initializing the primary key by using the syntax:

PRIMARY KEY (name_of_column),

The SQL code for this looks like:

```
CREATE TABLE student (
    student_id INT,
    first_name VARCHAR(20),
    last_name VARCHAR(20),
    major VARCHAR(20),
    gpa DECIMAL(4,2),
    PRIMARY KEY(student_id)
);
```

To check the create table, we could use a query called "DESCRIBE name_of_table;". This would give us all the details of the table we had just created. The command is

```
Describe student;
```

Delete the table

To delete the table the popSQL command is:

DROP TABLE name_of_table ;

In our case this command will look like this:

```
DROP TABLE student;
```

To check if the table has been deleted or not, we may use the DESCRIBE command.

Modify the table schema

Add a column

we may modify the table after creating it. Let's say I need to add the email ID of the student. We can use the below syntax to add a column to the existing table:

ALTER TABLE name_of_table ADD column_name data_type

The popSQL query to add a column is copied below:

```
ALTER TABLE student ADD email_id VARCHAR(50);
```

We may use "DESCRIBE student" to check the updated table.

Delete a column

We can also delete a column by changing the "ADD" reserved word to "DROP" in the above command. Let's say we want to drop the column GPA. The syntax to delete a columnis:

ALTER TABLE name_of_table DROP name_of_column;

The command to do that is:

```
ALTER TABLE student DROP gpa;
DESCRIBE student;
```

Insert data into the table

To store the data into the table we use syntax:

INSERT INTO name_of_table VALUES( );

The example is:

```
INSERT INTO student VALUES(1, 'Garima', 'Sharma', 'Math', 4.0);
```

Now we can check the data entered into the table student, by using syntax:

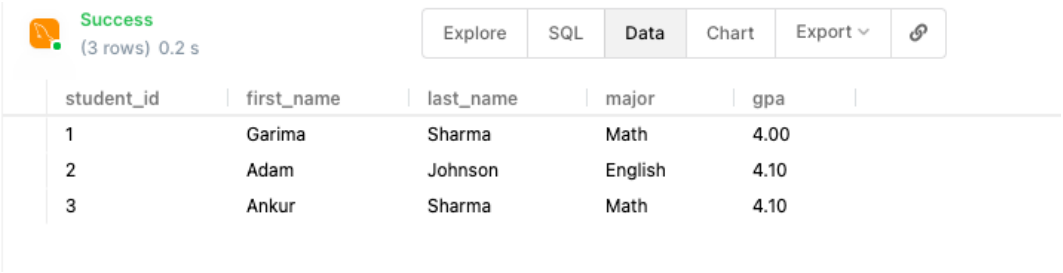
SELECT * FROM name_of_table;

This command will grab all the information present in the table and how it to us. The typical SQL command in our case is:

```
SELECT * FROM student;
```

Let's add another rows to the table and check all the data we have entered til now:

```
INSERT INTO student VALUES(2, 'Adam', 'Johnson', 'English',4.1);
INSERT INTO student VALUES(3, 'Ankur', 'Sharma', 'Math', 4.1);
SELECT * FROM student;
```



student_id	first_name	last_name	major	gpa
1	Garima	Sharma	Math	4.00
2	Adam	Johnson	English	4.10
3	Ankur	Sharma	Math	4.10

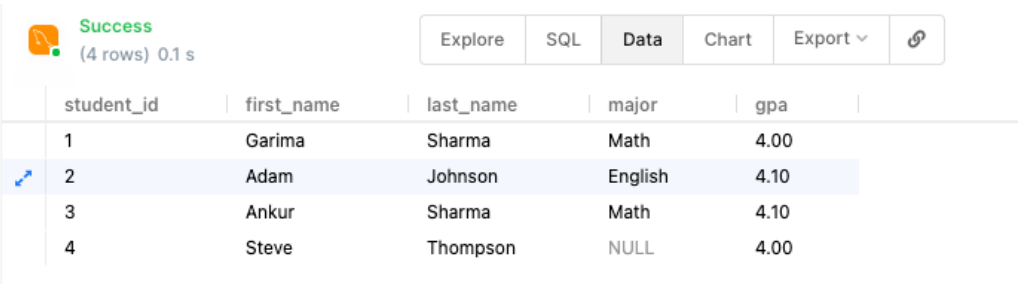
Consider a situation when we don't have all the information for a student say, for a student, we don't know what major subject he/she took. So in that situation, we can explicitly instruct the SQL to add data into the declared columns. The syntax for doing this is:

INSERT INTO name_of_table (column1, column2) VALUES(values_for_column1, value_for_column2)

The SQL code for adding an instance with limited value is:

```
INSERT INTO student(student_id, first_name, last_name, gpa) VALUES (4, 'Steve','Thompson',4.0);
```

Now the table student has another entry in it but its major value is missing.



student_id	first_name	last_name	major	gpa
1	Garima	Sharma	Math	4.00
2	Adam	Johnson	English	4.10
3	Ankur	Sharma	Math	4.10
4	Steve	Thompson	NULL	4.00

Adding conditions to the attributes of a Table

The two most useful conditions are: 'NOT NULL' and 'UNIQUE'. As their name suggests, when we add 'NOT NULL' to the attribute, we can't keep the data blank for that column, similar when we add "UNIQUE' to an attribute, we can't

have duplicate values in that column. Let's DROP the student table, and again CREATE it with having first_name as NOT NULL and major as UNIQUE. The SQL code is:

```
DROP TABLE student
CREATE TABLE student(
    student_id INT PRIMARY KEY,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20),
    major VARCHAR(20) UNIQUE,
    gpa DECIMAL(4,2)
);
```

Now, once the table is created we can start adding the data to the table. When we add NULL to the first_name it gives us an error because we have mentioned that in the table first_name column can't have NULL value.

```
INSERT INTO student VALUES(1, 'Garima', 'Sharma', 'Math', 4.0);
INSERT INTO student VALUES (2, NULL, 'Johson', 'English',4.1);
```

Now. let's check if the UNIQUE attribute is working or not. Insert a duplicate entry for major and see if SQL gives us an error or not.

```
INSERT INTO student VALUES(3, 'Ankur', 'Sharma', 'Math', 4.1);
```

As expected we get an error. To check the data we have entered till now in the table we can use SELECT * FROM table_name command.

```
SELECT * FROM student;
DESCRIBE student;
```

We can also add a default value to an attribute, so if no value is passed then the SQL will take the default value. For example:

```
CREATE TABLE student(
  student_id INT PRIMARY KEY,
  first_name VARCHAR(20) NOT NULL,
  last_name VARCHAR(20),
  major VARCHAR(20) DEFAULT 'undecided',
  gpa DECIMAL(4,2)
);
```

Now if we don't pass any value to the attribute major, the SQL will take the default value. Try inserting the following data

```
INSERT INTO student VALUES(1, 'Garima', 'Sharma', 'Math', 4.0);
INSERT INTO student(student_id, first_name, gpa) VALUES(2, 'ANKUR',4.1);
DESCRIBE student;
SELECT * FROM student;
```

Now the first row has no missing value and has passed all constraints declared in the table schema. For row 2, we don't have a value for column 'last_name' and column 'major'. The SQL will look for the constraints and put 'NULL' in 'last_name' and 'undecided' in the column 'major'. The undeclared values are filled by the keyword 'NULL'.

Another useful keyword is "AUTO_INCREMENT", it is useful where information is incrementing just like in student the student_id is incrementing. If we use AUTO_INCREMENT for our Student_id then we don't have to specify it every time we are inserting a value into the table. Take the example below:

```
CREATE TABLE student(
  student_id INT PRIMARY KEY AUTO_INCREMENT,
  first_name VARCHAR(20) NOT NULL,
  last_name VARCHAR(20),
  major VARCHAR(20) DEFAULT 'undecided',
  gpa DECIMAL(4,2)
);

INSERT INTO student VALUES('Garima','Sharma','Math',4.0);
INSERT INTO stduent(first_name, last_name, gpa) VALUES('Jack','Frost',4.1);
INSERT INTO student VALUES ('Ankur', 'Sharma', 'Mathematics', 4.0);
INSERT INTO student VALUES ('Adam', 'Johnson', 'Computer Science', 3.92);
INSERT INTO student VALUES ('Steve', 'Wilson', 'Bio',4.11);
```

Update and Delete Rows in a Table

It is interesting to see that we can actually update specific values added into the table by using a keyword "UPDATE table_name" and "SET". We can add conditions to update the values by using a keyword "WHERE", it is equivalent to the if statement in python or any other language. Let's say while entering the data into the table, the database admin has used the major 'Math' and 'Mathematics' interchangeably. But to make the uniformity into the table we can update this major to a more formal name 'Mathematics'. So, we can actually update the major column having 'Math' to 'Mathematics'. Note that we have put semicolon only at the end of the third line, that means the set of 3 lines is actually one single command for SQL.

```
UPDATE student
SET major = 'Mathematics'
WHERE major = 'Math' OR major = 'math' OR major = 'Maths';
```

You can become more creative here, say you want to set the gpa of students to 4.3 if the column gpa is greater than or equal to 4.0. We can do that as shown below:

```
UPDATE student
SET gpa = 4.3
WHERE gpa >= 4.0;
```

Or you can update the major of a student by selecting its student_id.

```
UPDATE student
SET major = 'Physics'
WHERE student_id = 2 AND student_id = 3;
SELECT * FROM student;
```

If in case we want to update the whole column, we can do it by just removing the WHERE statement as:

```
UPDATE stduent
SET major = 'Chemistry';
```

Now all the students have ‘chemistry’ in their ‘major’ column.

We can delete the rows in the same manner by using the syntax “DELETE FROM name_of_table”. Let’s say we want to delete the rows of the students having gpa less than 4. The SQL query for this looks like:

```
DELETE FROM student
WHERE gpa<=4;
```

We can also add more conditions to the WHERE statement using OR/ AND logical keywords. For example we want to delete a row if it has ‘first_name’ = ‘Garima’ and major = ‘Mathematics’. The SQL query is:

```
DELETE FROM student
WHERE first_name = 'Garima' AND major = 'Mathematics';
SELECT * FROM student;
```

Basic Queries to Access database

The queries to select some pieces for information from the RDBMS is done by using the syntax “SELECT”. For example, if have previously used the following code to access all the rows present in the database.

```
SELECT *
FROM student;
```

To select a particular column from the student table, we will replace ‘*’ with the column name. For e.g. if we need all the first_names from the table, then the SQL query looks like:

```
SELECT first_name
FROM student;
```

We can add more column names to the query:

```
SELECT first_name, gpa
FROM student;
```

There is another way to tell RDBMS what information we want by specifying column names as “ name_of_table.column_name”. For. e.g. the above query could be also written as:

```
SELECT student.first_name, student_gpa
FROM student;
```

This syntax will help us to access data from complex databases with more than one table.

We could also add order to the information we are getting, we can do this by using syntax “ORDER BY” after the “FROM name_of_table” command. For e.g. if can ORDER the rows in alphabetical order of names or in ascending order of gpa. we can do this by command:

```
SELECT name, gpa
FROM student
ORDER BY name;
```

By default, it is going to be in ascending order, but we can change it to defending order by adding “DESC” to the end of the command. For e.g. we want to have gpa ordered in descending order. The SQL query to do this is:

```
SELECT name, gpa
FROM student
ORDER by gpa DESC;
```

Another fun thing to try is, we can actually “ORDER BY” any othe column, even if we are not returning any values of that column in this query. For example, we can”ORDER BY” last_name or student_id. The SQL query to do this is:

```
SELECT name, gpa
FROM student
ORDER by student_id DESC;
```

We can add more columns to the ORDER BY to add more order to the data. For example we can order the data by gpa first, then if there are multiple rows with same gpa, these could be further ordered by say last_name. we can do this by writing a SQL query:

```
SELECT *
FROM student
ORDER by gpa, last_name ASC;
```

The above query will return all the data in the database student which is ordered first by GPA then by last_name in ascending order.

There is also another useful keyword “LIMIT number”. It would give us the number of rows we want. For example in the student database, we want to see the top 3 students in the class. The SQL query to do that is:

```
SELECT *
FROM student
ORDER by gpa ASC
LIMIT 2;
```

We have used WHERE statements in updating and deleting the information from the table. We can use WHERE statement to get filtered data too. For e.g. we want to have first_name of students having major = ‘Mathematics’. The SQL query is:

```
SELECT *
FROM student
ORDER BY student_id
WHERE major = 'Mathematics";
```

We can add more complexity to this query by using OR AND logical operators in the where statement. For. e.g to get students with, major mathematics and GPA greater than 4.0, the SQL query is:

```
SELECT *  -- select evertything
FROM student
ORDER BY studnet_id DESC
WHERE major = 'Mathematics' AND gpa > 4.0; -- <> is a not operator, we can use
-- OR AND < > <= >= or <>
```

Another useful keyword is “IN”. This keyword is associated with a list (’data1’, ‘data2’....) where the select command gives rows if column data belongs to the list. For. e.g we need to get the data if majors are from the list (‘Mathematics’, ‘Biology’). The SQL query to do this is:

```
SELECT *  -- select evertything
FROM student
ORDER BY studnet_id DESC
WHERE major IN ('Mathematics', 'Biology');
```

Till now we have learned about the SQL basics including tables, keys, schema, data types, updating and deleting from tables, and selecting specific information from tables. The SQL queries we have done till now are very simple because our database was simple, but as more and more complexity is added to the database, we need more complex SQL queries to access this data. For example, we can have multiple tables in the database, tables that can have foreign keys, and more.

More complex database and SQL queries

Let’s create a more complex database name: ‘Company database” having 5 tables. The schema for this database is copied below. It has 5 tables : employee, branch, client, works with, and branch supplier.

The first table to be created is “Employee” that has Emp_id as primary key, and super_id (supervisor id) and branch_id as foreign keys. The SQL to create this table is below. Keep in mind that at this point of time we can’t make branch_id and super_id as foreign keys because their respective tables have not been created yet. So we will start with the creation of table “Employee” with primary key “emp_id”.

Employee

emp_id	first_name	last_name	birth_date	sex	salary	super_id	branch_id
100	David	Wallace	1967-11-17	M	250,000	NULL	1
101	Jan	Levinson	1961-05-11	F	110,000	100	1
102	Michael	Scott	1964-03-15	M	75,000	100	2
103	Angela	Martin	1971-06-25	F	63,000	102	2
104	Kelly	Kapoor	1980-02-05	F	55,000	102	2
105	Stanley	Hudson	1958-02-19	M	69,000	102	2
106	Josh	Porter	1969-09-05	M	78,000	100	3
107	Andy	Bernard	1973-07-22	M	65,000	106	3
108	Jim	Halpert	1978-10-01	M	71,000	106	3

Branch

branch_id	branch_name	mgr_id	mgr_start_date
1	Corporate	100	2006-02-09
2	Scranton	102	1992-04-06
3	Stamford	106	1998-02-13

Client

client_id	client_name	branch_id
400	Dunmore Highschool	2
401	Lackawana Country	2
402	FedEx	3
403	John Daly Law, LLC	3
404	Scranton Whitepages	2
405	Times Newspaper	3
406	FedEx	2

Works_With

emp_id	client_id	total_sales
105	400	55,000
102	401	267,000
108	402	22,500
107	403	5,000
108	403	12,000
105	404	33,000
107	405	26,000
102	406	15,000
105	406	130,000

Branch Supplier

branch_id	supplier_name	supply_type
2	Hammer Mill	Paper
2	Uni-ball	Writing Utensils
3	Patriot Paper	Paper
2	J.T. Forms & Labels	Custom Forms
3	Uni-ball	Writing Utensils
3	Hammer Mill	Paper
3	Stamford Lables	Custom Forms

```
CREATE TABLE employee (
  emp_id INT PRIMARY KEY,
  first_name VARCHAR(20) NOT NULL,
  last_name VARCHAR(20),
  birth_date DATE,
  sex VARCHAR(1),
  salary INT,
  super_id INT,
  branch_id INT
);
```

Now let's create another table branch. This table has a foreign key 'mgr_id' (manager id). The SQL query to create branch table is:

```
CREATE TABLE branch (
  branch_id INT PRIMARY KEY,
  branch_name VARCHAR(30),
  mgr_id INT,
  mgr_start_date DATE,
  FOREIGN KEY (mgr_id) REFERENCES employee(emp_id) ON DELETE SET NULL
);
```

In the table Branch, we have create a foreign id that took references from the emp_id. Now we can alter the columns of employee table and set branch_id and supe_id as foreign keys. The SQL query is:

```
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;
```

Now create another table ‘client’. By following the above database schema, the SQL query for creating client table is:

```
CREATE TABLE client (
  client_id INT PRIMARY KEY,
  client_name VARCHAR(30),
  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(20),
    supply_type VARCHAR(30),
    PRIMARY KEY(branch_id, supplier_name),
    FOREIGN KEY (branch_id) REFERENCES branch(branch_id) ON DELETE CASCADE
);
```

Now, we have created all the tables in company_database. The next step is to put information in these tables. Because, here in tables are linked to each other via foreign keys, we will be adding information in a certain way.

Adding data to the tables

Let’s add the first row in table employee. The SQL query is:

```
INSERT INTO employee VALUES(100, 'David','Cooper', '1986-03-08', 'M', 150000, NULL, NULL);
```

While inserting our first row in employee, we have added NULL values to the foreign keys because the tables connected to these keys haven’t yet made or have values. So, the trick here is to update the NULL values in branch_id foreign key later i.e. after adding the data to branch table. Okay then, let’s add data (first_row) to branch table:

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

Now once we have first row ready in branch table, we can update the foreign key branch_id in employee table as:

```
UPDATE employee
SET branch_id =1
WHERE emp_id=100;
```

Because branch_id = 1 is for corporate branch, we have one more entry in employee table associated with corporate branch. Let's add that Information too in the employee table by using the following query:

```
INSERT INTO employee VALUES(101, 'Jan', 'Barter', '1961-05-11', 'F',120000, 100, 1);
```

Now, we will do the same thing for Scranton branch. The SQL query for this is:

```
INSERT INTO employee VALUES(102, 'Michael','Scott','1964-05-04', 'M', 110000, 100, NULL);
INSERT INTO branch VALUES(2, 'Scranton', 102, '1992-04-10');

UPDATE employee
SET branch_id =2
WHERE emp_id = 102;

INSERT INTO employee VALUES (103, 'Angela', 'martin', '1991-07-19', 'F', 115000, 102, 2);
INSERT INTO employee VALUES (104, 'kelly', 'kapoor', '1998-03-10', 'M', 112000, 102, 2);
INSERT INTO employee VALUES (105, 'Stanley', 'Hudson', '1991-07-19', 'M', 95000, 102, 2);
```

Now, we have successfully entered the 5 rows in table employee and 2 rows in table branch. Repeating the steps, we can add data for Stanford branch too in both employee and branch tables. The SQL query to do so is:

```
INSERT INTO employee VALUES(106, 'Josh','Porter', '1998-12-28','M', 98000, 100, NULL);
INSERT INTO branch VALUES(3, 'Stanford', 106, '1998-10-18');

UPDATE employee
SET branch_id=3
WHERE emp_id = 106;

INSERT INTO employee VALUES(107, 'Andy', 'Butler', '1999-10-11', 'F', 78000, 106, 3);
INSERT INTO employee VALUES(108, 'Garima', 'Sharma', '1999-08-03', 'F', 80000, 106, 3);
```

After this query, we have entered all the data in table: employee and branch.

Lets' move further and add values to the rest of the tables: works_with, clients, and branch_supplier. This is pretty straightforward just like normal entries. The SQL queries for each table is written below:

```
INSERT INTO client VALUES(400, 'WC highschool', 2);
INSERT INTO client VALUES(401, 'Simcoe County', 2);
INSERT INTO client VALUES(402, 'UPS', 3);
INSERT INTO client VALUES(403, 'Innisfil', 3);
INSERT INTO client VALUES(404, 'bradford', 2);
INSERT INTO client VALUES(405, 'CA highschool', 3);
INSERT INTO client VALUES(406, 'UPS', 2);
```

```
INSERT INTO works_with VALUES(105, 400, 55000);
INSERT INTO works_with VALUES(102, 401, 155000);
INSERT INTO works_with VALUES(108, 402, 65000);
INSERT INTO works_with VALUES(107, 403, 5000);
INSERT INTO works_with VALUES(108, 403, 10000);
INSERT INTO works_with VALUES(105, 404, 33000);
INSERT INTO works_with VALUES(107, 405, 255000);
INSERT INTO works_with VALUES(102, 406, 100000);
INSERT INTO works_with VALUES(105, 406, 355000);
```

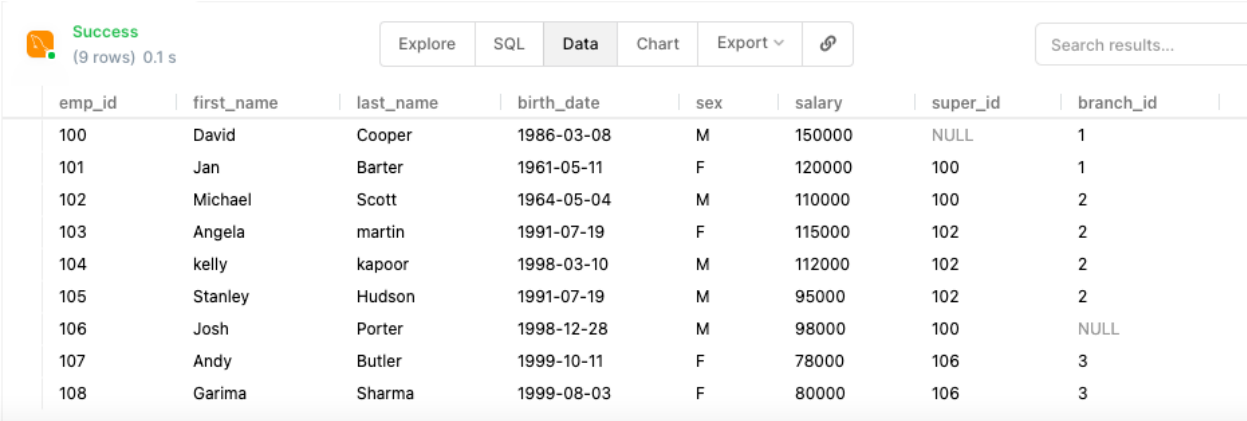
```
INSERT INTO branch_supplier VALUES(2, 'Hammermil', 'paper');
INSERT INTO branch_supplier VALUES(2, 'Uni-ball', 'utensils');
INSERT INTO branch_supplier VALUES(3, 'ABC Inc', 'paper');
INSERT INTO branch_supplier VALUES(2, 'G.S.supplies', 'stationary');
INSERT INTO branch_supplier VALUES(3, 'Uni-ball', 'pen');
INSERT INTO branch_supplier VALUES(3, 'Hammermil', 'paper');
INSERT INTO branch_supplier VALUES(3, 'Staples', 'toner');
```

These three sets of SQL queries helped us to add all the data to the database tables. Let's check all the data in all tables by using the command SELECT as shown below:

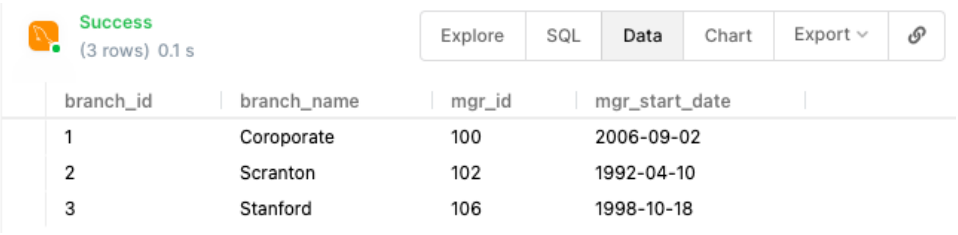
```
SELECT *
FROM employee;

SELECT * FROM branch;
SELECT * FROM works_with;
SELECT * FROM client;
SELECT * FROM branch_supplier;
```

The tables created are shown below:



emp_id	first_name	last_name	birth_date	sex	salary	super_id	branch_id
100	David	Cooper	1986-03-08	M	150000	NULL	1
101	Jan	Barter	1961-05-11	F	120000	100	1
102	Michael	Scott	1964-05-04	M	110000	100	2
103	Angela	martin	1991-07-19	F	115000	102	2
104	kelly	kapoor	1998-03-10	M	112000	102	2
105	Stanley	Hudson	1991-07-19	M	95000	102	2
106	Josh	Porter	1998-12-28	M	98000	100	NULL
107	Andy	Butler	1999-10-11	F	78000	106	3
108	Garima	Sharma	1999-08-03	F	80000	106	3



branch_id	branch_name	mgr_id	mgr_start_date
1	Coroporate	100	2006-09-02
2	Scranton	102	1992-04-10
3	Stanford	106	1998-10-18

Success
(9 rows) 0.1 s

Explore

emp_id	client_id	total_sales
102	401	155000
102	406	100000
105	400	55000
105	404	33000
105	406	355000
107	403	5000
107	405	255000
108	402	65000
108	403	10000

client_id	client_name	branch_id
400	WC highschool	2
401	Simcoe County	2
402	UPS	3
403	Innisfil	3
404	bradford	2
405	CA highschool	3
406	UPS	2

Success
(7 rows) 0.1 s

Explore SQL

branch_id	supplier_name	supply_type
2	G.S.supplies	stationary
2	Hammermil	paper
2	Uni-ball	utensils
3	ABC Inc	paper
3	Hammermil	paper
3	Staples	toner
3	Uni-ball	pen

Awesome Queries and keywords

Let's get all the employees from the employee table. We will be using the SELECT command.

```
SELECT first_name, last_name
FROM employee
```

Similarly, we can use this query to get data from other tables. Say we want to see the client_names from the client table, the SQL query would be:

```
SELECT client_name FROM client;
```

Success
(7 rows) 0.1 s

client_name
WC highschool
Simcoe County
UPS
Innisfil
bradford
CA highschool
UPS

Let's find all the employees ordered by salary:

```
SELECT * FROM employee ORDER BY salary; -- ascending order by default
```

Success
(9 rows) 0.1 s

Explore | SQL | Data | Chart | Export | Search results...

emp_id	first_name	last_name	birth_date	sex	salary	super_id	branch_id
107	Andy	Butler	1999-10-11	F	78000	106	3
108	Garima	Sharma	1999-08-03	F	80000	106	3
105	Stanley	Hudson	1991-07-19	M	95000	102	2
106	Josh	Porter	1998-12-28	M	98000	100	NULL
102	Michael	Scott	1964-05-04	M	110000	100	2
104	kelly	kapoor	1998-03-10	M	112000	102	2
103	Angela	martin	1991-07-19	F	115000	102	2
101	Jan	Barter	1961-05-11	F	120000	100	1
100	David	Cooper	1986-03-08	M	150000	NULL	1

Let's write a query with more complexity, find all employees order by sex then first_name.

```
SELECT * FROM employee ORDER BY sex, first_name;
```

Success
(9 rows) 0.1 s

Explore | SQL | Data | Chart | Export | Search results...

emp_id	first_name	last_name	birth_date	sex	salary	super_id	branch_id
107	Andy	Butler	1999-10-11	F	78000	106	3
103	Angela	martin	1991-07-19	F	115000	102	2
108	Garima	Sharma	1999-08-03	F	80000	106	3
101	Jan	Barter	1961-05-11	F	120000	100	1
100	David	Cooper	1986-03-08	M	150000	NULL	1
106	Josh	Porter	1998-12-28	M	98000	100	NULL
104	kelly	kapoor	1998-03-10	M	112000	102	2
102	Michael	Scott	1964-05-04	M	110000	100	2
105	Stanley	Hudson	1991-07-19	M	95000	102	2

Get the first 4 employees from the employee table.

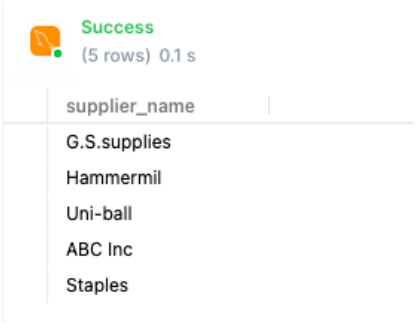
```
SELECT * FROM employee LIMIT 4;
```

In SQL, the first_name has a keyword forename and the last_name has a keyword surname. While in our table we have used first_name and last_name, so we can get the forename and surname by buying AS keyword.

```
SELECT first_name AS forename, last_name AS surname FROM employee LIMIT 4;
```

Another very useful SQL keyword is “DISTINCT”. It gives all the unique entries in the column. For eg. if we want to see our branch_suppliers, we can write query as:

```
SELECT DISTINCT supplier_name FROM branch_supplier;
```



SQL functions

There are some pretty useful in-build functions in SQL. These functions have some dedicated roles and could be called as needed without any need for explanation in code.

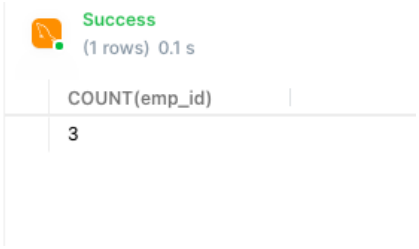
COUNT( )

This function is very useful in counting entries in a column. For e.g. the SQL query to count athenumber of employees in the employee table is copied below. Another

```
SELECT COUNT(emp_id) FROM employee;
SELECT COUNT(super_id) FROM employee; -- counts not null values
```

Let's have a more complex query. Let's count the number of female employees born after 1970. The SQL query is:

```
SELECT COUNT(emp_id)
FROM employee
WHERE sex = 'F' AND birth_date > '1970-12-31';
```



AVG( )

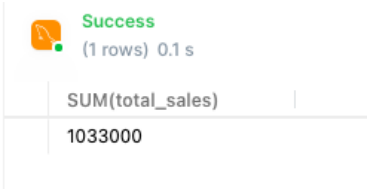
This keyword stands for average. Let's find the average of all male employee salaries.

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

SUM ( )

As the name suggests, it gives the sum of the column data. For e.g. let's find out the total sales from works_with table. The SQL query would be:

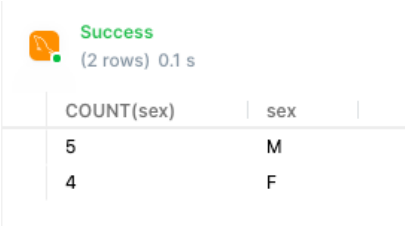
```
SELECT SUM(total_sales) FROM works_with;
```



GROUP BY

This is an awesome keyword that sorts the data into the groups defined by the user in the query. For example if we want to see how many male and female employees the company has, we can use the keyword GROUP BY, so that we can group the data into male and female clusters. The SQL query is:

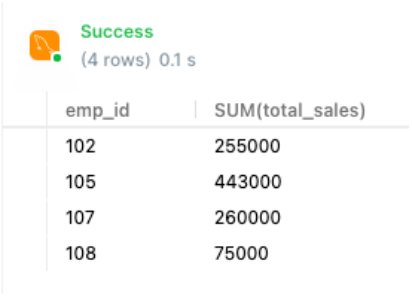
```
SELECT COUNT(sex), sex
FROM employee
GROUP BY sex;
```



This query will return 2 column data : 1st the count of Sex, and sex itself. The count is grouped by sex. That means it will give the number of females (sex) in column sex and the number of males in column sex.

Let's design another query where we want to see the sales done by each salesman in table works_with. You can see multiple entries here in emp_id. That means multiple sales have been done by one salesperson, so we can use GROUP BY the emp_id. The SQL query is:

```
SELECT emp_id, SUM(total_sales)
FROM works_with
GROUP BY emp_id;
```



The way of getting more than one data from the table is called as aggregation, it is quite useful. In this, we use multiple keywords together to get some meaningful information.

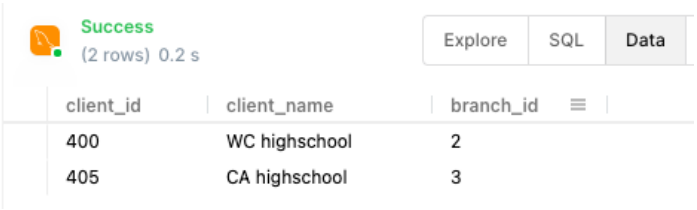
WILDCARDS (%, _) and LIKE keyword

The wildcard is used to match the data with a specific pattern. The two wildcards we have are % and _ (underscore). The % stands for any number of characters, and _ (underscore) stands for one character.

LIKE keyword is used to match the data with the pattern defined after like.

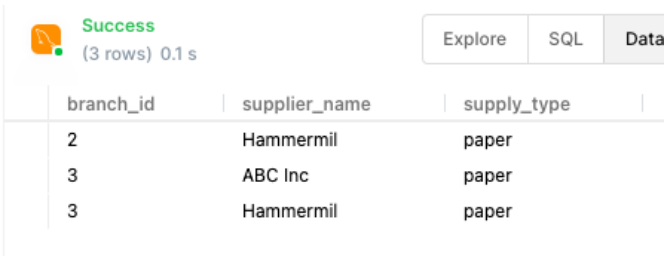
1. Let's write a SQL query to find the client's name ending with the word 'school'.

```
SELECT *
FROM client
WHERE client_name LIKE '%school';
```



2. Write another Query to find all the clients that supply paper. The SQL query is:

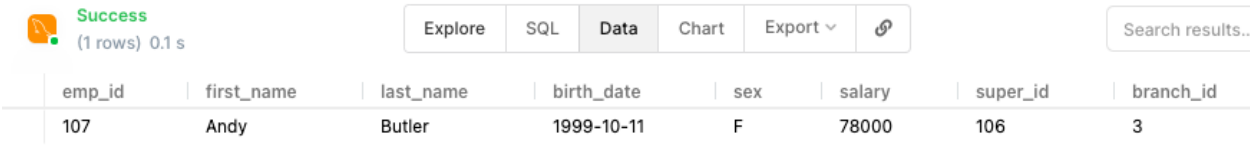
```
SELECT *
FROM branch_supplier
WHERE supply_type LIKE '%paper%';
```



The % wildcard before and after the word paper will find if word paper appears anywhere in the column supply_type, it could have any number of characters before and after the word ‘paper’.

3. Find any employee born in October. This is interesting, we will be using the underscore wildcard that matches only one character. The October month is the 10th and dates are written as YYYY-MM-DD. So we need to find ‘10’ in the MM pattern. We can have anything in YYYY or anything in DD. So, we can use 4 underscores for YYYY and % for anything after MM. The SQL query is:

```
SELECT *
FROM employee
WHERE birth_date LIKE '____-10%';
```



UNION operator

The union operator combines results from more than one select statement. The foremost important rule to using UNION is that both the SELECT statements must have the same # of columns. Another rule is that both SELECT statements must return the same data type. We will learn how to use the UNION keyword by writing some SQL queries:

1. Find the list of employee, client names, and branch names

```
SELECT first_name AS company_data
FROM  employees
UNION
SELECT client_names
FROM client
UNION
SELECT branch_name
FROM branch;
```

Joins in SQL

Join operator in SQL is used for joining rows of two or more tables having at least one variable in common. The JOIN operator is used with another operator ‘ON’ which specifies the common column between two tables.

Let’s take an example and find out all the branches and their branch managers. If we see in the database schema, we can find that the common column between the branch table and employee table is employee_id/ mgr_id. That means we will be joining rows from these two tables via column emp_id or mgr_id.

Before starting to write SQL query let’s add another branch into the branch table having null in mgr_id, and mgr_start_date columns. The SQL query for this is a simple INSERT operator as shown below:

```
INSERT branch VALUES (4, 'Bradford', NULL, NULL);
```

Now, let’s find all the branches and the names of their branch managers.

```
SELECT employee.emp_id, employee.first_name, branch.branch_name
FROM employee
JOIN branch
ON employee.emp_id = branch.branch_id;
```

Unlike the select statement, we have written till now, In the above SQL code we have selected columns from two tables, we can do it because we are joining these two tables. The join operator works till the entries in emp_id is equal to entries in mgr_id. The output of this Query is copied below, we can see that there are 3 rows, one for each branch.

This was the basic JOIN operator. In SQL, we have 4 types of join: Inner Join (the one explained above), left join,

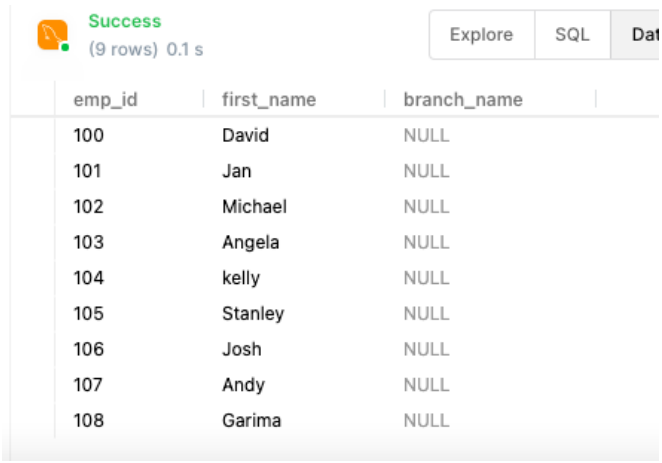
LEFT JOIN

In left join, all of the columns selected from the left table are included in the output. The left table is the table written in front of FROM operator. In our case the left table is employee table, hence all the emp_id and first_name will be selected and returned in the output (not just branch managers), the data under the column branch_name will be NULL because these employees are not branch managers.

The SQL query is:

```
SELECT employee.emp_id, employee.first_name, branch.branch_name
FROM employee
LEFT JOIN branch
ON employee.emp_id = branch.branch_id;
```

The output of this query is:



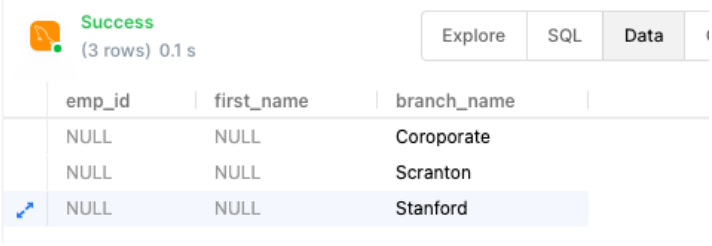
emp_id	first_name	branch_name
100	David	NULL
101	Jan	NULL
102	Michael	NULL
103	Angela	NULL
104	kelly	NULL
105	Stanley	NULL
106	Josh	NULL
107	Andy	NULL
108	Garima	NULL

RIGHT JOIN

As you might have guessed, this right join will do just the opposite and will show all the rows from the right table. In this example, the right table is branch table. The SQL query and the output is:

```
SELECT employee.emp_id, employee.first_name, branch.branch_name
FROM employee
RIGHT JOIN branch
ON employee.emp_id = branch.branch_id;
```

The output is copied below, now you can see the NULL values in emp_id and first_name for Bradford branch because for this branch there are no managers.



emp_id	first_name	branch_name
NULL	NULL	Coroporate
NULL	NULL	Scranton
NULL	NULL	Stanford

FULL JOIN

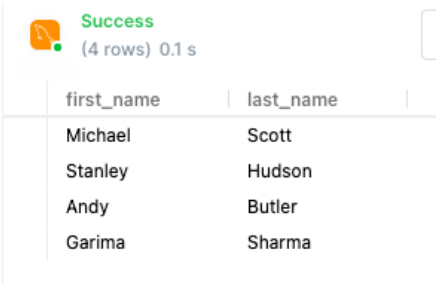
In this type of join, all the rows from left and right tables are joined together.

NESTED QUERIES

In nested queries, we use multiple select statements in order to get a certain information.

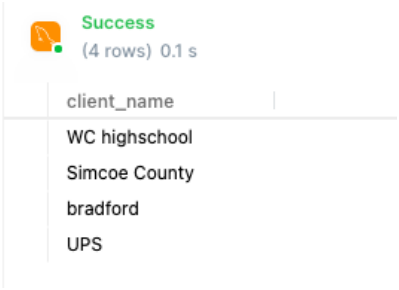
1. Find name of all employees who have sold over 50,000 to a single client.

```
SELECT employee.first_name, employee.last_name
FROM employee
WHERE employee.emp_id IN (
  SELECT works_with.emp_id
  FROM works_with
  WHERE works_with.total_sales > 50000
);
```



2. Find all clients who are handled by the brach that Michael Scott manages, Assume you know Michale's id.

```
SELECT client.client_name
FROM client
WHERE client.branch_id =(
  SELECT branch.branch_id
  FROM branch
  WHERE branch.mgr_id = 102
);
```



Note rather than using the operator 'IN', we are actually using the equality here because we want to select items associated with mgr_id = 102.