**SQL Assignment 2**

1. For an online purchasing database, create entity relationship diagrams. Create a database object from your entity diagram.

# Creating a SQL Entity Relationship Diagram (ERD)

Using PostgreSQL and Pagila Database to practice creating and using ERDs

1. Using pgAdmin 4 and PostgreSQL this article will demonstrate how to Create an Entity Relationship Diagram (ERD) of the Pagila Database to gain a greater understanding of the data you will be working with and to create a reference for which to work off of.
2. There will display two separate methods for ERD creation. One is manually done and is much more time intensive but a better learning experience, the other is automatically done by pgAdmin 4 and is orders of magnitude faster and easier for those that are just looking for an ERD to reference when using the Pagila Database.
3. There will be links to the necessary information, github repositories and completed ERD for your reference.

# PRE-NOTE (If you want to create an ERD fast then read this)

When I initially tried to create an ERD using pgAdmin 4 I clicked on the “New ERD Project” option in the Tools dropdown and it gave me a blank grid to work off of. Because of that I actually had to manually create each table, row, and constraint in the diagram and it took me hours. The next time I attempted to create an ERD I first right-clicked on the database I wanted to work with and instead chose “Generate ERD” and initially it still appeared that I had a blank ERD to start with but when I zoomed out I noticed that an ERD was already created for me and it included all of the accurate table names, row names, constraints and relationships. **In the end what had previously taken me hours to complete now only took me 30 seconds.**

That being said, I found it very useful to create my own ERD manually. It got me thinking about the process, the data types, constraints, and relationships in a way that I felt further solidified my knowledge of SQL.

# PREREQUISITES

This tutorial assumes you already have PostgreSQL, pgAdmin 4, and The Pagila Database installed and set-up. If you need instructions on setting this up please visit my previous article: [Simulating A Real-world SQL Environment.](https://jamesopacich.medium.com/simulating-a-real-world-sql-environment-63784f7f37ae)

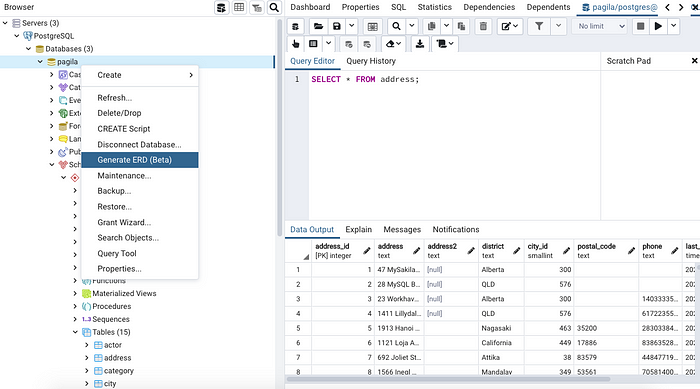
Please have pgAdmin 4 open at this point and your database ready to use.

# STEPS FOR CREATING ERD AUTOMATICALLY

# STEP 1: Generate ERD tool

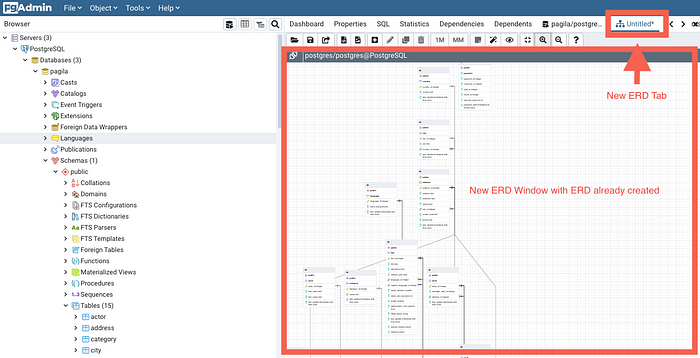
This is a Beta tool in pgAdmin 4 version 5.0.

## Generate ERD



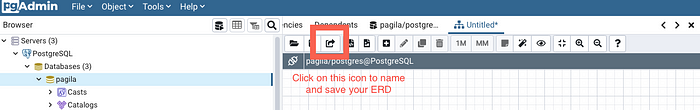
Right-click on your database. I have called my database Pagila in this case. Then find the “Generate ERD” option on the drop down list.

This will open a new window in pgAdmin 4 located where the query window would normally be.

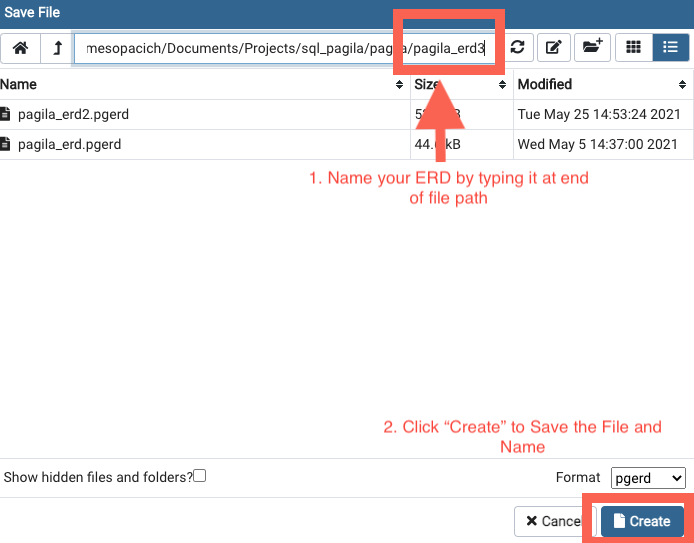


Here a new ERD window and tab are created. This is where the ERD will be created.

# STEP 2: Save and Name Your ERD



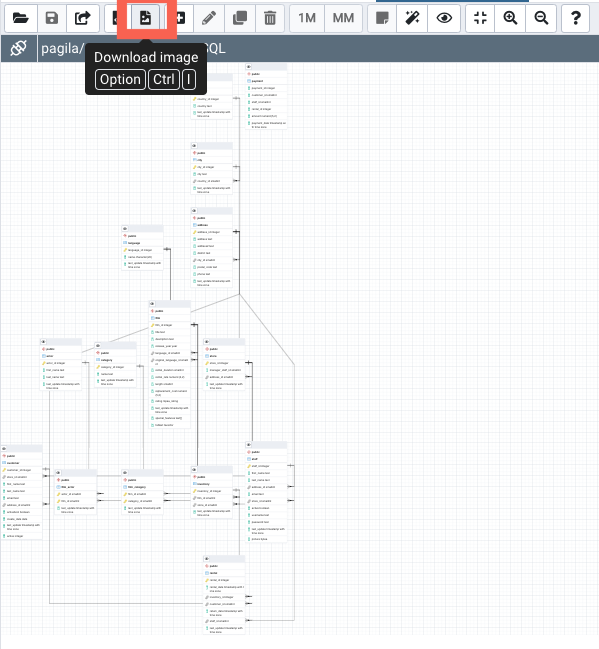
Step 1: Click on this icon to open up the save file browser/box



Step 2: Name your ERD by typing your desired name into the end of the file path. Then click Create to save the named file. I decided to name my ERD pagila\_erd3 for this exercise.

# STEP 3: Export Your ERD

The ERD is a bit difficult to zoom in and move around within pgAdmin 4 so I found it more helpful to export it to an image file for easier referencing.

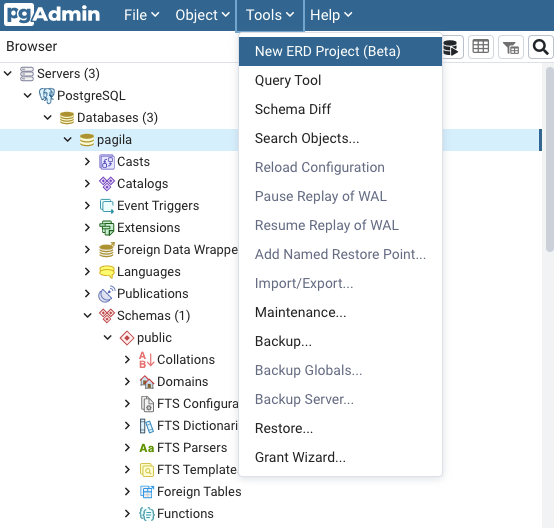


Click on the “Download Image” icon, it will bring up a file browser where you can name your new image file and save it to a desired file on your computer.

# MANUALLY CREATING YOUR OWN ERD

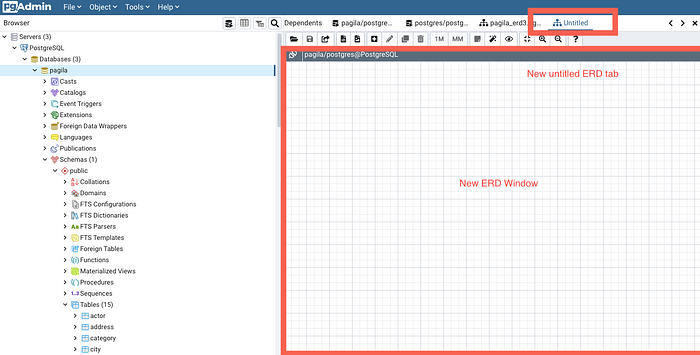
To manually create your own ERD the very first step is different.

# STEP 1: CREATE NEW ERD

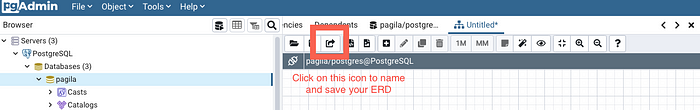


Instead of clicking on the database name and then selecting to generate a new database you will instead click on the “Tools” dropdown at the top of the pgAdmin app window and select “Create New ERD.”

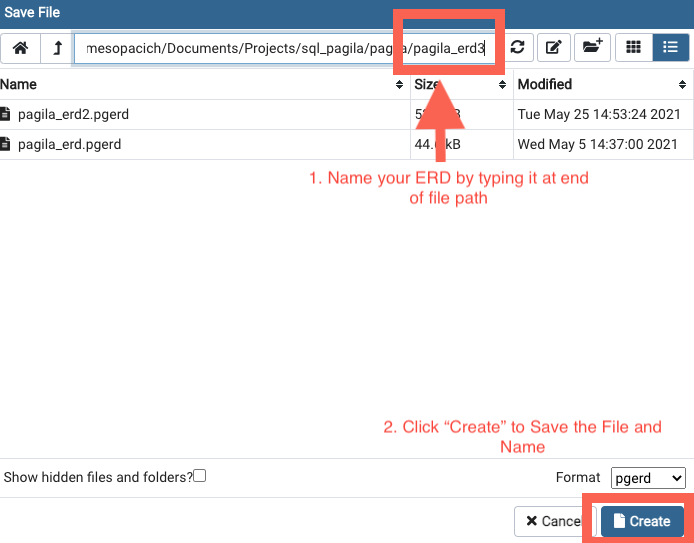
# STEP 2: Save and Name Your ERD



You will see now that you will have a blank ERD Window to work from.



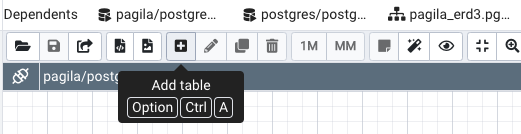
Step 1: Click on this icon to open up the save file browser/box



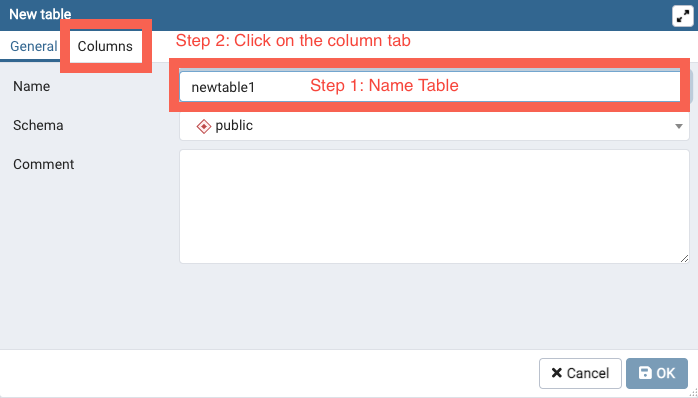
Step 2: Name your ERD buy typing your desired name into the end of the file path. Then click Create to save the named file. I decided to name my ERD pagila\_erd3 for this exercise.

# STEP 2: Create Your First Table

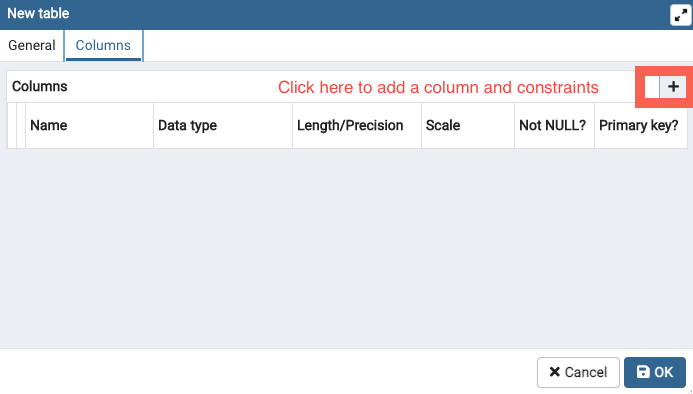
In this step we will create the table diagrams for the ERD. These boxes contain the table, column, constraint and key information for each table in your database.



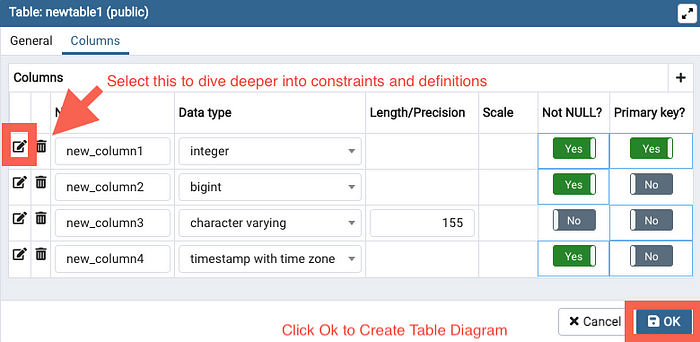
Click the plus-sign icon to add a table.



Name your table and click on the column tab.

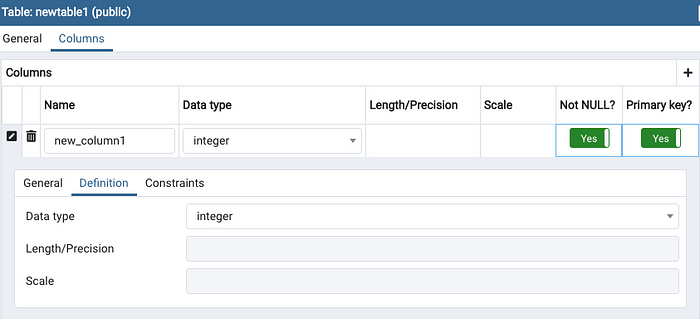


Click on plus sign to add column and constraints

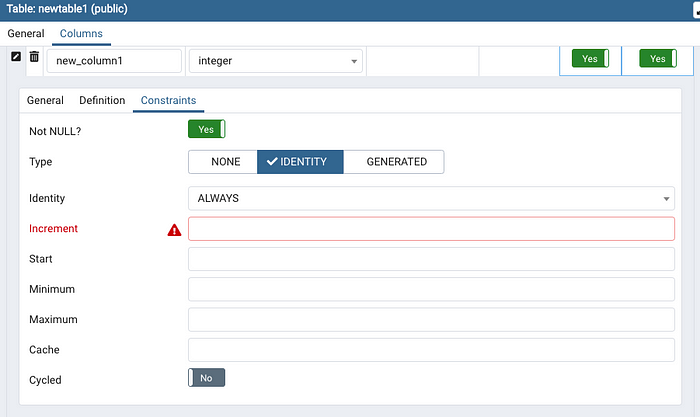


Enter in your column names, data-types and basic constraints here and hit ok to create the table diagram. If you want to dive deeper into constraints then click on the pencil icon next to each column.

## IN CASE YOU WANT TO GET DEEPER WITH CONSTRAINTS



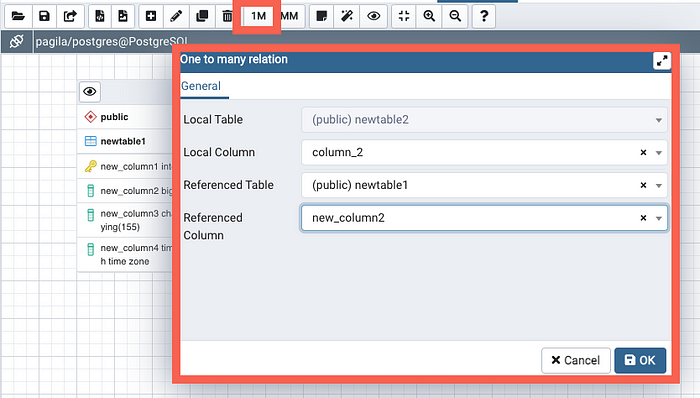
There is a definitions tab although this doesn’t seem to add any additional functionality.



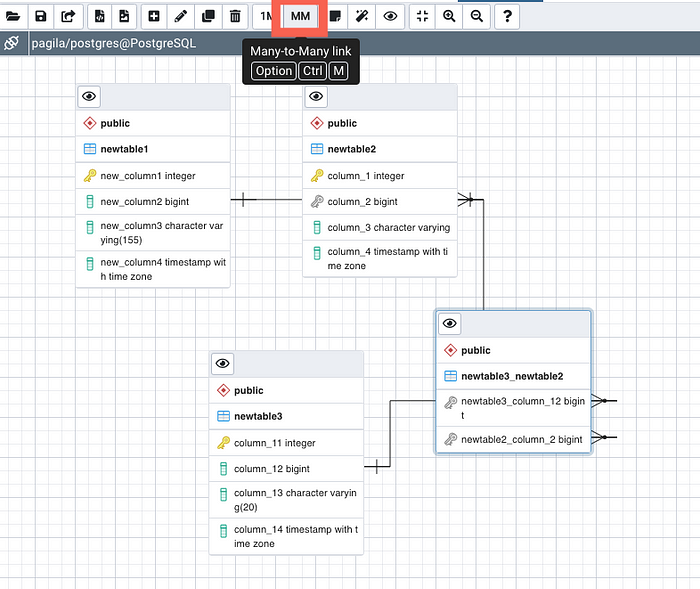
The constraints tab does seem to add a few more parameters to add to your diagram.

# STEP 3: Creating Relationships

To complete your ERD you need to create relationships between tables.



To create a 1 to many relationship highlighting foreign key relationships click on the 1M icon. This will open a form for you to select columns corresponding to the tables and relationships you want to create.



1. Create a SQL store process to register the use of the database, complete it with proper validation and transaction rollback and commit.

# SQL TRANSACTIONS

**Last Updated :**06 Sep, 2023

Transactions group a set of tasks into a single execution unit. Each transaction begins with a specific job and ends when all the tasks in the group successfully completed. If any of the tasks fail, the transaction fails. Therefore, a transaction has only two results: **success** or **failure**.

Example of a transaction to transfer $150 from account A to account B:

1. read(A)  
2. A:= A – 150  
3. write(A)  
4. read(B)  
5. B:= B + 150  
6. write(B)

Incomplete steps result in the failure of the transaction. A database transaction, by definition, must be atomic, consistent, isolated, and durable.

These are popularly known as [ACID](https://en.wikipedia.org/wiki/ACID) properties.  These properties can ensure the concurrent execution of multiple transactions without conflict.

## Properties of Transaction

* **Atomicity:**The outcome of a transaction can either be completely successful or completely unsuccessful. The whole transaction must be rolled back if one part of it fails.
* **Consistency:** Transactions maintain integrity restrictions by moving the database from one valid state to another.
* **Isolation:** Concurrent transactions are isolated from one another, assuring the accuracy of the data.
* **Durability:** Once a transaction is committed, its modifications remain in effect even in the event of a system failure.

## **How to Implement Transactions Control Command using SQL?**

The following commands are used to control transactions. It is important to note that these statements cannot be used while creating tables and are only used with the DML Commands such as- [INSERT](https://www.geeksforgeeks.org/sql-insert-statement/), [UPDATE](https://www.geeksforgeeks.org/sql-update-statement/), and [DELETE](https://www.geeksforgeeks.org/sql-delete-statement/).

### **BEGIN TRANSACTION Command**

It indicates the start point of an explicit or local transaction.

**Syntax:**

*BEGIN TRANSACTION transaction\_name ;*

### **SET TRANSACTION Command**

The values for the properties of the current transaction, such as the transaction isolation level and access mode, are set using the SET TRANSACTION Statement in MySQL.

**Syntax:**

*SET TRANSACTION [ READ WRITE | READ ONLY ];*

### **COMMIT Command**

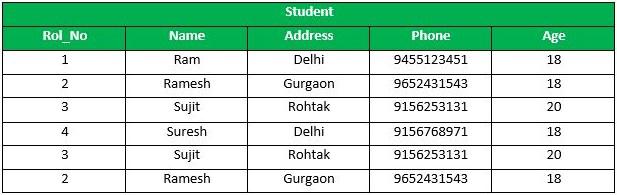
If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called **committed**. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

**Syntax:**

*COMMIT;*

**Example:**

**Sample Table1**



*output*

Following is an example which would delete those records from the table which have age = 20 and then COMMIT the changes in the database.

### **Query**

DELETE FROM Student WHERE AGE = 20;  
COMMIT;

### **Output**

Thus, two rows from the table would be deleted and the SELECT statement would look like,



*output*

### **ROLLBACK Command**

If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback**. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

**Syntax for ROLLBACK command:**

*ROLLBACK;*

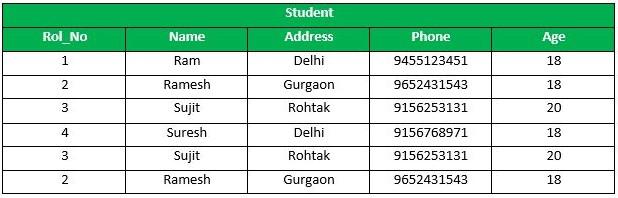
**Example:**

From the above example **Sample table1**,   
Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database.

### **Query**

DELETE FROM Student WHERE AGE = 20;  
ROLLBACK;

### **Output**



*output*

### **SAVEPOINT Command**

**SAVEPOINT** creates points within the groups of transactions in which to ROLLBACK.   
A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction.

**Syntax for Savepoint command:**

*SAVEPOINT SAVEPOINT\_NAME;*

This command is used only in the creation of SAVEPOINT among all the transactions.   
In general ROLLBACK is used to undo a group of transactions.

**Syntax for rolling back to Savepoint Command:**

*ROLLBACK TO SAVEPOINT\_NAME;*

you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.

**Example:**

From the above example **Sample table1**, Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database by keeping Savepoints.

### **Query**

SAVEPOINT SP1;  
//Savepoint created.  
DELETE FROM Student WHERE AGE = 20;  
//deleted  
SAVEPOINT SP2;  
//Savepoint created.

Here SP1 is first SAVEPOINT created before deletion.In this example one deletion have taken place.   
After deletion again SAVEPOINT SP2 is created.

### **Output**



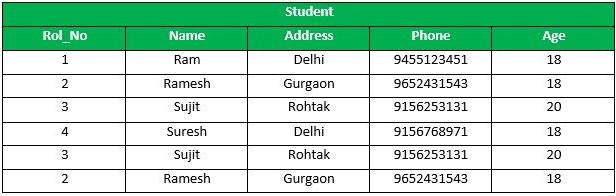
*output*

Deletion have been taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP1 which is before deletion.   
deletion is undone by this statement.

### Query

ROLLBACK TO SP1;  
//Rollback completed

### Output



*output*

### **RELEASE SAVEPOINT Command**

This command is used to remove a SAVEPOINT that you have created.

**Syntax:**

*RELEASE SAVEPOINT SAVEPOINT\_NAME*

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

It is used to initiate a database transaction and used to specify characteristics of the transaction that follows.

## Conclusion

* SQL statements are grouped together using transactions.
* They promise that either all changes are made or none at all.
* Reliability is ensured by the ACID qualities of atomicity, consistency, isolation, and durability.
* Start a transaction with “BEGIN TRANSACTION,” and finish it with “COMMIT” or “ROLLBACK” to complete or undo changes.

1. List the SQL aggregate function and demonstrate how to utilize it.

# SQL Aggregate functions

**Last Updated :**30 Apr, 2024

**SQL Aggregate functions** are functions where the values of multiple rows are grouped as input on certain criteria to form a single value result of more significant meaning.

It is used to summarize data, by combining multiple values to form a single result.

SQL Aggregate functions are mostly used with the GROUP BY clause of the SELECT statement.

**Various Aggregate Functions**

1. Count()
2. Sum()
3. Avg()
4. Min()
5. Max()

## Aggregate Functions in SQL

Below is the list of SQL aggregate functions, with examples

### **Count():**

* ***Count(\*):*** Returns the total number of records .i.e 6.
* ***Count(salary):*** Return the number of Non-Null values over the column salary. i.e 5.
* ***Count(Distinct Salary):*** Return the number of distinct Non-Null values over the column salary .i.e 5.

### **Sum():**

* ***sum(salary):*** Sum all Non-Null values of Column salary i.e., 310
* ***sum(Distinct salary):***Sum of all distinct Non-Null values i.e., 250.

### **Avg():**

* ***Avg(salary)*** = Sum(salary) / count(salary) = 310/5
* ***Avg(Distinct salary)*** = sum(Distinct salary) / Count(Distinct Salary) = 250/4

### **Min():**

* ***Min(salary):***Minimum value in the salary column except NULL i.e., 40.

### Max():

* ***Max(salary):***Maximum value in the salary i.e., 80.

## Demo SQL Database

In this tutorial on aggregate functions, we will use the following table for examples:

| **Id** | **Name** | **Salary** |
| --- | --- | --- |
| 1 | A | 802 |
| 2 | B | 403 |
| 3 | C | 604 |
| 4 | D | 705 |
| 5 | E | 606 |
| 6 | F | NULL |

You can also create this table on your system, by writing the following queries:

MySQL

CREATE TABLE Employee (

Id INT PRIMARY KEY,

Name CHAR(1), -- Adjust data type and length if names can be longer than a single character

Salary DECIMAL(10,2) -- Adjust precision and scale if needed for salaries

);

INSERT INTO Employee (Id, Name, Salary)

VALUES (1, 'A', 802),

(2, 'B', 403),

(3, 'C', 604),

(4, 'D', 705),

(5, 'E', 606),

(6, 'F', NULL);

## Aggregate Function Example

In this example, we will use multiple aggregate functions on the data.

#### Queries

--Count the number of employees  
**SELECT COUNT**(\*) **AS** TotalEmployees **FROM** Employee;  
  
-- Calculate the total salary  
**SELECT** **SUM**(Salary) **AS** TotalSalary **FROM** Employee;  
  
-- Find the average salary  
**SELECT AVG**(Salary) **AS** AverageSalary **FROM** Employee;  
  
-- Get the highest salary  
**SELECT MAX(**Salary) **AS** HighestSalary **FROM** Employee;  
  
-- Determine the lowest salary  
**SELECT MIN**(Salary) **AS** LowestSalary **FROM** Employee;

#### Output

**TotalEmployees**  
6  
**TotalSalary**  
3120  
**AverageSalary**  
624  
**HighestSalary**  
802  
**LowestSalary**  
403

## Key Takeaways about SQL Aggregate Functions

* *Aggregate functions in SQL operate on a group of values and return a single result.*
* *They are often used with the GROUP BY clause to summarize the grouped data.*
* *Aggregate function operates on non-NULL values only (except COUNT).*
* *Commonly used aggregate functions are –*[*MIN()*](https://www.geeksforgeeks.org/sql-min-and-max/#:~:text=and%20max%20functions.-,SQL%20MIN()%20Function,-SQL%20min%20function)*,*[*MAX()*](https://www.geeksforgeeks.org/sql-min-and-max/#:~:text=from%20all%20records.-,SQL%20MAX()%20Functions,-SQL%20max%20function)*,*[*COUNT()*](https://www.geeksforgeeks.org/sql-count-avg-and-sum/#:~:text=SQL%20COUNT()%20Function)*,*[*AVG()*](https://www.geeksforgeeks.org/sql-count-avg-and-sum/#:~:text=table_name%0AWHERE%20condition%3B-,SQL%20AVG()%20Function,-%3A%0AThe%20AVG)*, and*[*SUM()*](https://www.geeksforgeeks.org/sql-count-avg-and-sum/#:~:text=table_name%0AWHERE%20condition%3B-,SQL%20SUM()%20Function,-%3A%0AThe%20SUM)*.*

1. In SQL, create a pivot query.

In SQL, Pivot and Unpivot are relational operators that are used to transform one table into another in order to achieve more simpler view of table. Conventionally we can say that **Pivot** operator converts the rows data of the table into the column data.

The **Unpivot** operator does the opposite that is it transform the column based data into rows.

**Syntax:**

**1. Pivot:**

SELECT (ColumnNames)

FROM (TableName)

PIVOT

(

AggregateFunction(ColumnToBeAggregated)

FOR PivotColumn IN (PivotColumnValues)

) AS (Alias) //Alias is a temporary name for a table

1. With an example, describe how to join in SQL.

# SQL Joins (Inner, Left, Right and Full Join)

**SQL Join** operation combines data or rows from two or more tables based on a common field between them.

In this article, we will learn about **Joins in SQL,**covering JOIN types, syntax, and examples.

## SQL JOIN

SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables.

We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with [**WHERE clause**](https://www.geeksforgeeks.org/sql-where-clause)to filter data retrieval.

## SQL JOIN Example

Consider the two tables below as follows:

**Student:**

**StudentCourse** :



Both these tables are connected by one common key (column) i.e ROLL\_NO.

We can perform a JOIN operation using the given SQL query:

**SELECT** s.roll\_no, s.name, s.address, s.phone, s.age, sc.course\_id  
**FROM** Student s  
**JOIN** StudentCourse sc **ON** s.roll\_no = sc.roll\_no;

**Output:**

| ROLL\_NO | NAME | ADDRESS | PHONE | AGE | COURSE\_ID |
| --- | --- | --- | --- | --- | --- |
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 | 1 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 | 2 |
| 3 | RIYANKA | SILGURI | XXXXXXXXXX | 20 | 2 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 | 3 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 | 1 |

## Types of JOIN in SQL

There are many types of Joins in SQL. Depending on the use case, you can use different type of SQL JOIN clause. Here are the frequently used SQL JOIN types:

* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN
* NATURAL JOIN

## **SQL INNER JOIN**

The [**INNER JOIN**](https://www.geeksforgeeks.org/sql-inner-join) keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

### **Syntax**:

The syntax for SQL INNER JOIN is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**INNER JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **table1**: First table.
* **table2**: Second table
* **matching\_column**: Column common to both the tables.

***Note****: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*



### **INNER JOIN Example**

Let’s look at the example of INNER JOIN clause, and understand it’s working.

This query will show the names and age of students enrolled in different courses.

**SELECT** StudentCourse.COURSE\_ID, Student.NAME, Student.AGE **FROM** Student  
**INNER JOIN** StudentCourse  
**ON** Student.ROLL\_NO = StudentCourse.ROLL\_NO;

**Output**:



## **SQL LEFT JOIN**

LEFT JOIN returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.

### **Syntax**

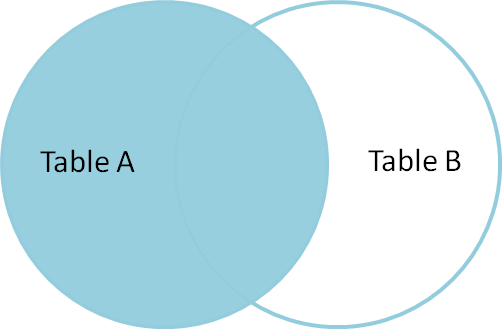
The syntax of LEFT JOIN in SQL is**:**

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**LEFT JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **table1:** First table.
* **table2**: Second table
* **matching\_column**: Column common to both the tables.

***Note****: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*



### **LEFT JOIN Example**

Let’s look at the example of LEFT JOIN clause, and understand it’s working

**SELECT** Student.NAME,StudentCourse.COURSE\_ID   
**FROM** Student  
**LEFT JOIN** StudentCourse   
**ON** StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output**:



## **SQL RIGHT JOIN**

[**RIGHT JOIN**](https://www.geeksforgeeks.org/sql-right-join) returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join.It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

### **Syntax:**

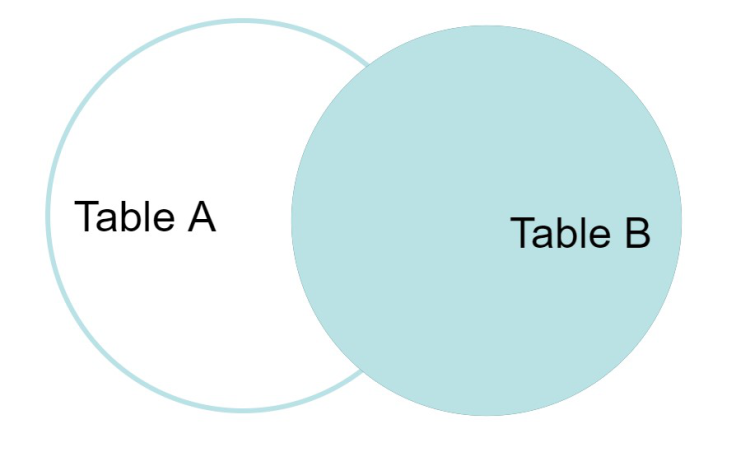
The syntax of RIGHT JOIN in SQL is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**RIGHT JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **table1**: First table.
* **table2**: Second table
* **matching\_column**: Column common to both the tables.

**Note**: We can also use **RIGHT OUTER JOIN** instead of RIGHT JOIN, both are the same.



### **RIGHT JOIN Example**:

Let’s look at the example of RIGHT JOIN clause, and understand it’s working

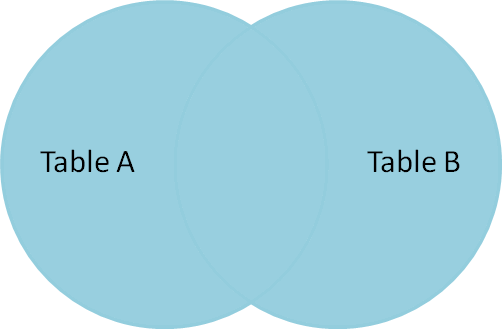
**SELECT** Student.NAME,StudentCourse.COURSE\_ID   
**FROM** Student  
**RIGHT JOIN** StudentCourse   
**ON** StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**



## **SQL FULL JOIN**

[**FULL JOIN**](https://www.geeksforgeeks.org/sql-full-join) creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.



**Syntax**

The syntax of SQL FULL JOIN is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**FULL JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **table1**: First table.
* **table2**: Second table
* **matching\_column**: Column common to both the tables.

### **FULL JOIN Example**

Let’s look at the example of FULL JOIN clause, and understand it’s working

**SELECT** Student.NAME,StudentCourse.COURSE\_ID   
**FROM** Student  
**FULL JOIN** StudentCourse   
**ON** StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**

| **NAME** | **COURSE\_ID** |
| --- | --- |
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | NULL |
| ROHIT | NULL |
| NIRAJ | NULL |
| NULL | 4 |
| NULL | 5 |
| NULL | 4 |

## SQL Natural join (?)

Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables.

Both table must have at least one common column with same column name and same data type.

The two table are joined using **Cross join**.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

### Natural join Example:

Look at the two tables below- Employee and Department

| **Employee** | | |
| --- | --- | --- |
| Emp\_id | Emp\_name | Dept\_id |
| 1 | Ram | 10 |
| 2 | Jon | 30 |
| 3 | Bob | 50 |

| **Department** | |
| --- | --- |
| **Dept\_id** | **Dept\_name** |
| 10 | IT |
| 30 | HR |
| 40 | TIS |

**Problem**: Find all Employees and their respective departments.

**Solution Query**: (Employee) ? (Department)

| **Emp\_id** | **Emp\_name** | **Dept\_id** | **Dept\_id** | **Dept\_name** |
| --- | --- | --- | --- | --- |
| 1 | Ram | 10 | 10 | IT |
| 2 | Jon | 30 | 30 | HR |
| Employee data | | | Department data | |

1. How to locate the 4th highest value in a column in a row. Create your table.

# SQL Query to find the Nth Largest Value in a Column using Limit and Offset

Prerequisite – [How to find Nth highest salary from a table](https://www.geeksforgeeks.org/find-nth-highest-salary-table/)   
**Problem Statement :** Write an SQL query to find the nth largest value from the column using [LIMIT](https://www.geeksforgeeks.org/sql-limit-clause/) and [OFFSET](https://www.geeksforgeeks.org/sql-offset-fetch-clause/).

**Example-1 :**

**Table –** BILLS

| **FLATNo.** | **ElectricityBill** |
| --- | --- |
| 101 | 1000 |
| 102 | 1500 |
| 103 | 1300 |
| 201 | 2300 |
| 202 | 6700 |
| 203 | 7500 |
| 204 | 1300 |
| 301 | 2300 |

The above table has the electricity bills of all the flats in an apartment. You have to find the nth largest electricity bill in the table.

SELECT DISTINCT ElectricityBill AS NthHighestElectricityBill

FROM Bills

ORDER BY ElectricityBill DESC

LIMIT 1

OFFSET n-1;

Here n should be an integer whose value must be greater than zero.

**Explanation :**  
In the above query, we are sorting the values of ElectricityBill column in descending order using [Order By clause](https://www.geeksforgeeks.org/sql-order-by/) and by selecting only distinct values. After sorting it in descending order we have to find the Nth value from the top, so we use OFFSET n-1 which eliminates the top n-1 values from the list, now from the remaining list we have to select only its top element, to do that we use LIMIT 1.

If we want to find the 3rd highest electricity bill the query will be –

SELECT DISTINCT ElectricityBill AS 3rdHighestElectricityBill

FROM Bills

ORDER BY ElectricityBill DESC

LIMIT 1

OFFSET 2;

The result of the above query will be –

| **3rdHighestElectricityBill** |
| --- |
| 2300 |

**Example-2 :**

**Table –** EmployeeSalary

| **EmployeeID** | **SalaryInThousands** |
| --- | --- |
| 1A234 | 450 |
| 1D765 | 259 |
| 5A567 | 320 |
| 3B653 | 450 |
| 3A980 | 259 |
| 9R345 | 128 |
| 2A748 | 316 |

The above table has the salaries of employees working in a small company. Find the employee id who is earning the 4th highest salary.

SELECT EmployeeID AS 4thHighestEarningEmployee

FROM EmployeeSalary

ORDER BY SalaryInThousands DESC

LIMIT 1

OFFSET 3;

**Explanation :**   
Here distinct is not used because we need employee whose earnings stand at 4th place among all the employee’s (i.e 316k not 259k).

The result of the above query will be –

| **4thHighestEarningEmployee** |
| --- |
| 2A748 |