**Advance Database Systems**

**PostgreSQL Tutorial**

**UBUNTU SERVER 20.04 64 bits**

**PGADMIN 4**

**By: Ronald M. Baclayon**

In this PostgreSQL Tutorial I will show you how to use POSTGRESQL using PGADMIN 4.

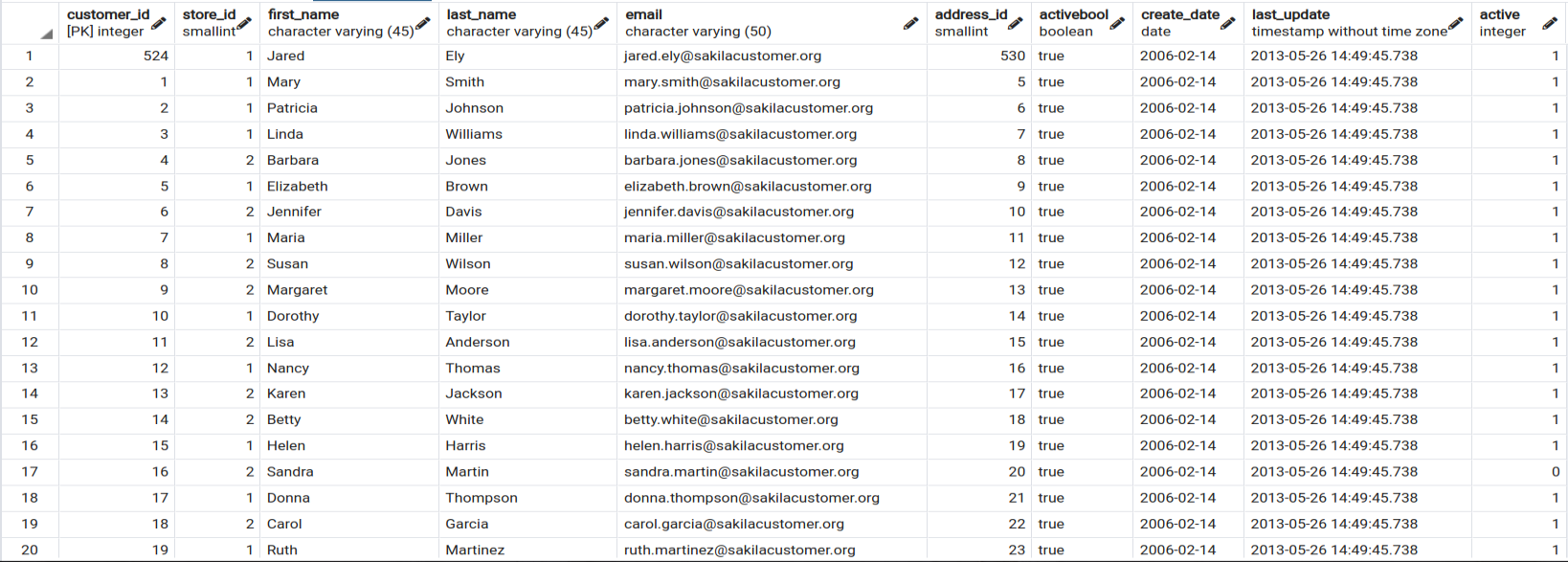
You can download PGADMIN 4 here:

[**https://www.pgadmin.org/download/**](https://www.pgadmin.org/download/)

**Section 1**

**Querying Data**

**SAMPLE TABLE: customer**



|  |  |
| --- | --- |
| **Select** – show you how to query data from a single table. | |
| **SAMPLE SYNTAX** | **SELECT**  column\_name  **FROM**  table\_name; |
| You can directly specify what table you wanted to view using the syntax.  SELECT  *first\_name*,  *last\_name*  FROM  customer; |  |
| If you wanted to view all columns inside the table. Just use the syntax below.  SELECT  *\**  FROM  customer;  As you can see, I’m using asterisk \* instead of a specific column name. |  |

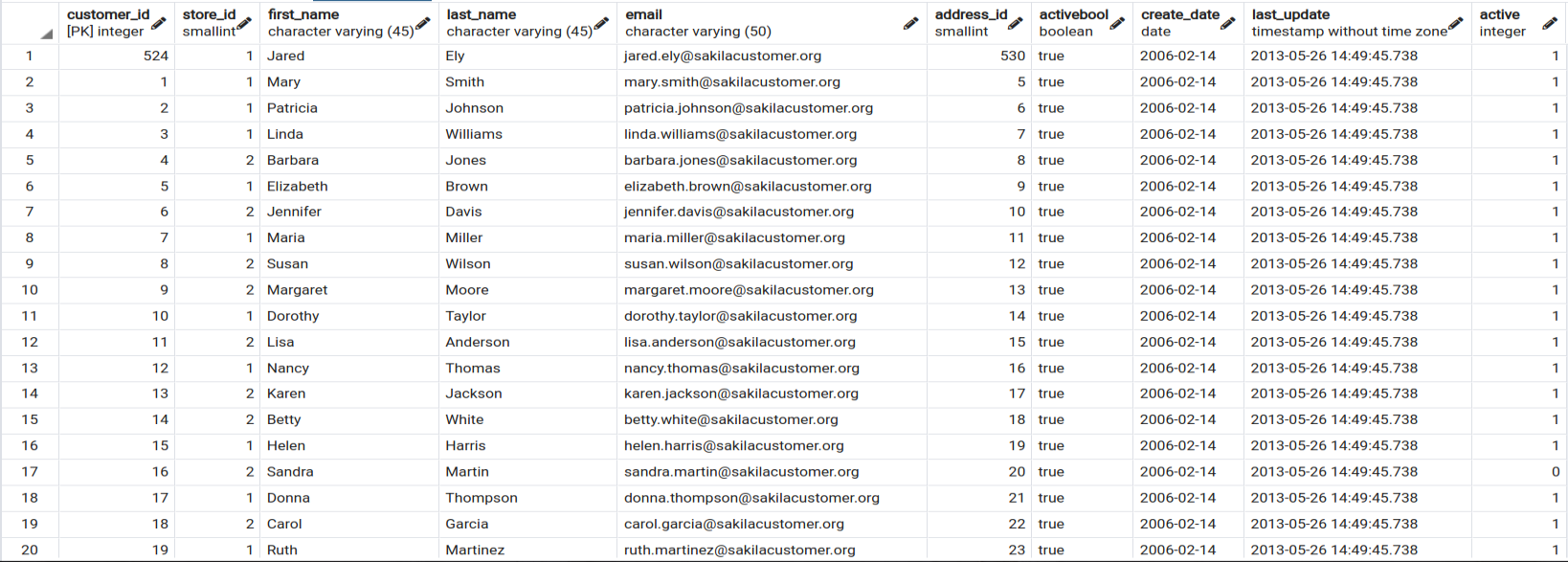
|  |  |
| --- | --- |
| **Column aliases –** learn how to assign temporary names to columns or expressions in a query. | |
| **SAMPLE SYNTAX** | **SELECT**  column\_name **AS** alias\_name  **FROM**  table\_name; |
| If you want to change the column name of your output you can use the syntax below.  SELECT  first\_name AS customer\_name  FROM  customer; | |
| ***WITHOUT*** COLUMN ALIAS | ***WITH*** COLUMN ALIAS |
| You can also combine 2 columns into 1 column. *first\_name || ' ' || last\_name AS "full name"*  Note: you can add space or additional characters between single quote ‘ ’  *||* is use as concatenating operator  If your new column name has spaces, you can instead put it inside a double quote “new column name”.  SELECT  first\_name || ’, ’ || last\_name  AS “customer full name”  FROM  Customer; |  |

|  |  |
| --- | --- |
| **Order By** – guide you on how to sort the result set returned from a query. | |
| **SAMPLE SYNTAX** | **SELECT**  column\_name  **FROM**  table\_name  **ORDER BY**  column\_name **ASC/DSC;** |
| You can use “Order By” to sort a column either in ascending or descending order.  ASC – ascending (default)  DSC – descending  If you will not specify if you want to sort the column by ASC or DSC, by default automatically it will be sorted by ASC.  SELECT  last\_name || ‘, ’ || first\_name  AS “full name”  FROM  customer  ORDER BY  “full name” DESC; | |
| **SORT BY ASC** | **SORT BY DESC** |

|  |  |
| --- | --- |
| **Select Distinct –** provide you a clause that removes duplicate rows in the result set. | |
| **SAMPLE SYNTAX** | **SELECT**  **DISTINCT** column\_name  **FROM**  table\_name; |
| You can use distinct to remove duplicates in a column.  **SELECT**  **DISTINCT** first\_name  **FROM**  customer  **ORDER** **BY**  first\_name **DESC**; | You can use DISTINCT ON if you have multiple columns output.  **SELECT**  **DISTINCT** **ON** (first\_name) first\_name, last\_name  **FROM**  Customer  **ORDER** **BY**  first\_name DESC, last\_name DESC; |
| **WITHOUT DISTINCT** | **WITH DISTINCT** |
| **WITHOUT ON (2 COLUMNS )** | **WITH ON (2 COLUMNS)** |

**Section 2**

**Filtering Data**

**SAMPLE TABLE: customer**

|  |  |
| --- | --- |
| **Where –** filter rows based on a specified condition. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **WHERE**  **Condition;** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Operator | Description |  | Operator | Description | | = | Equal |  | AND | Logical operator AND | | > | Greater than |  | OR | Logical operator OR | | < | Less than |  | IN | Return true if a value matches any value in a list | | >= | Greater than or equal |  | BETWEEN | Return true if a value is between a range of values | | <= | Less than or equal |  | LIKE | Return true if a value matches a pattern | | <> or != | Not equal |  | IS NULL | Return true if a value is NULL | |  |  |  | NOT | Negate the result of other operators | | |
| **SELECT**  **last\_name,**  **first\_name**  **FROM**  **customer**  **WHERE**  **last\_name LIKE ‘An%’;**  On this syntax, we specify that we only wanted to view customers with last\_name starting with AN. |  |

|  |  |
| --- | --- |
| **Limit** – get a subset of rows generated by a query. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **LIMIT amount;** |
| **Limit is used to filter number of results. For example you only wanted to view up to 10 result per query.** |  |

|  |  |
| --- | --- |
| **Fetch–** limit the number of rows returned by a query. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **FETCH FIRST amount ROW ONLY;** |
| **FETCH is same as LIMIT, it limits the number of result.**  **OFFSET 5 ROWS**  **– means that you wanted to skip the first 5 results**  **– so instead of having 1 to 10 as a result we have instead 6 to 15.** |  |

|  |  |
| --- | --- |
| **In** – select data that matches any value in a list of values. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **WHERE**  **Column\_name IN (value, value, ….);** |
| **IN is used to provide specific results base on the given list, instead of using OR.** |  |

|  |  |
| --- | --- |
| **Between –** select data that is a range of values. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **WHERE**  **Column\_name BETWEEN value AND value;** |
| **BETWEEN is used to provide specific results base on given range.** |  |

|  |  |
| --- | --- |
| **Like – filter data based on pattern matching.** | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **WHERE**  **Column\_name BETWEEN value AND value;** |
| On this syntax, we specify that we only wanted to view customers with last\_name starting with AN. |  |

|  |  |
| --- | --- |
| **Like – filter data based on pattern matching.** | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name**  **FROM**  **Table\_Name**  **WHERE**  **Column\_name IS NULL;** |
| The query results empty rows because there is no row that contains null value. It means all customer information has been supplied with data. |  |

**Section 3**

**Joining Multiple Tables**

|  |  |
| --- | --- |
| **TABLE NAME: film\_actor** | **TABLE NAME: actor** |

|  |  |
| --- | --- |
| **Inner Join –** select rows from one table that has the corresponding rows in other tables. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name *(Note: Primary Key)***  **Column\_Name**  **FROM**  **Table\_Name\_1**  **INNER JOIN**  **Table\_Name\_2**  **ON Table\_Name\_1.Primary\_Key = Table\_Name2.Foreign\_Key;** |
|  | |

|  |  |
| --- | --- |
| **Left Join –** select rows from one table that may or may not have the corresponding rows in other tables. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name *(Note: Primary Key)***  **Column\_Name**  **FROM**  **Table\_Name\_1**  **LEFT JOIN**  **Table\_Name\_2**  **ON Table\_Name\_1.Primary\_Key = Table\_Name2.Foreign\_Key;** |
|  | |

|  |  |
| --- | --- |
| **Full Outer Join – use the full join to find a row in a table that does not have a matching row in another table** | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name *(Note: Primary Key)***  **Column\_Name**  **FROM**  **Table\_Name\_1**  **FULL OUTER JOIN**  **Table\_Name\_2**  **ON Table\_Name\_1.Primary\_Key = Table\_Name2.Foreign\_Key;** |
|  | |

|  |  |
| --- | --- |
| **Cross Join** – produce a Cartesian product of the rows in two or more tables. | |
| **SAMPLE SYNTAX** | **SELECT**  **Column\_Name *(Note: Primary Key)***  **Column\_Name**  **FROM**  **Table\_Name\_1**  **CROSS JOIN Table\_Name\_2;** |
| **There is no result because natural join only works if both tables has at least only one the same column name, such as the PRIMARY KEY and the FOREIGN KEY only.**  **On this scenario both table FILM\_ACTOR and ACTOR has column “LAST UPDATE”.** | |

### Section 4

### Grouping Data

### TABLE: customer

### 

### TABLE: payment

### 

|  |  |
| --- | --- |
| **Group By** – divide rows into groups and applies an aggregate function on each.  **Having** – apply conditions to groups. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **FROM**  **Table\_Name\_1**  **GROUP BY**  **column\_name\_1,**  **column\_name\_2**  **HAVING condition;** |
| **NOTE:**   1. **The columns in the GROUP BY clause must much the columns in the SELECT clause.** 2. **Using \* in the SELECT clause is not allowed, columns must be specified.** 3. **If you have columns first\_name and last\_name in your SELECT clause, you must also specify it in the GROUP BY clause, else you will receive and error.** | |

### Section 5

### Set Operations

### TABLE: top\_rated\_films;

### 

### TABLE: most\_popular\_films;

### 

|  |  |
| --- | --- |
| **Union** – combine result sets of multiple queries into a single result set. | |
| **SAMPLE SYNTAX** | **SELECT column\_name\_1 FROM Table\_Name\_1**  **UNION/UNION ALL**  **SELECT column\_name\_1 FROM Table\_Name\_2;** |
| **UNION** | **UNION ALL** |
| **NOTE:**   1. **UNION** only works if both tables have the same number of columns and data type. | |

|  |  |
| --- | --- |
| **Intersect** – combine the result sets of two or more queries and returns a single result set that has the rows appear in both result sets. | |
| **SAMPLE SYNTAX** | **SELECT column\_name\_1 FROM Table\_Name\_1**  **INTERSECT**  **SELECT column\_name\_1 FROM Table\_Name\_2;** |
|  | |
| **NOTE:**   1. **INTERSECT** only works if both tables have the same number of columns and data type. 2. **INTERSECT** will only show output from both tables that has the same values. 3. In the example, both tables have the title “The Godfather” and release\_year “1972”. | |

|  |  |
| --- | --- |
| **Except** – return the rows in the first query that does not appear in the output of the second query. | |
| **SAMPLE SYNTAX** | **SELECT column\_name\_1 FROM Table\_Name\_1**  **EXCEPT**  **SELECT column\_name\_1 FROM Table\_Name\_2;** |
|  | |
| **NOTE:**   1. **EXCEPT** only works if both tables have the same number of columns and data type. 2. **EXCEPT** works oppositely with **INTERSECT.** | |

**Section 6**

**Grouping sets, Cube, and Rollup**

### TABLE: customer

### 

### TABLE: payment

### 

|  |  |
| --- | --- |
| **Grouping Sets** – generate multiple grouping sets in reporting. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3,**  **aggregate\_function**  **FROM Table\_Name\_1;**  **GROUP BY**  **GROUP SETS (column\_name\_1, column\_name\_2, … );** |
|  | |
| **NOTE:**   1. **GROUPING SETS** will provide results for each group. Example, you have 2 groups, each group has 10 results, then it means that you will have a total of 20 results. | |

|  |  |
| --- | --- |
| **Cube** – define multiple grouping sets that include all possible combinations of dimensions. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3,**  **aggregate\_function**  **FROM Table\_Name\_1;**  **GROUP BY**  **CUBE (column\_name\_1, column\_name\_2, … );** |
|  | |
| **NOTE:**   1. **CUBE** works like an auto group and would generates all possible grouping sets. | |

|  |  |
| --- | --- |
| **Cube** – define multiple grouping sets that include all possible combinations of dimensions. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3,**  **aggregate\_function**  **FROM Table\_Name\_1;**  **GROUP BY**  **ROLLUP (column\_name\_1, column\_name\_2, … );** |
|  | |
| **NOTE:**   1. **ROLLUP** works like an auto group and would generates all possible grouping sets but base on hierarchy. | |

**Section 7**

**Subquery**

### TABLE: customer

### 

### TABLE: payment

### 

|  |  |
| --- | --- |
| **Subquery** – write a query nested inside another query. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3**  **FROM Table\_Name\_1;**  **WHERE**  **column\_name\_3 < (subquery);** |
|  | |
| **NOTE:**   1. **Subquery** means a query inside another query. | |

|  |  |
| --- | --- |
| **ANY** – retrieve data by comparing a value with a set of values returned by a subquery. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3**  **FROM Table\_Name\_1;**  **WHERE**  **column\_name\_3 = ANY (subquery);** |
|  | |

|  |  |
| --- | --- |
| **ALL** – query data by comparing a value with a list of values returned by a subquery. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3**  **FROM Table\_Name\_1;**  **WHERE**  **column\_name\_3 > ALL (subquery);** |
|  | |

|  |  |
| --- | --- |
| **EXISTS** – check for the existence of rows returned by a subquery. | |
| **SAMPLE SYNTAX** | **SELECT**  **column\_name\_1,**  **column\_name\_2,**  **column\_name\_3**  **FROM Table\_Name\_1;**  **WHERE**  **EXIST (subquery);** |
|  | |