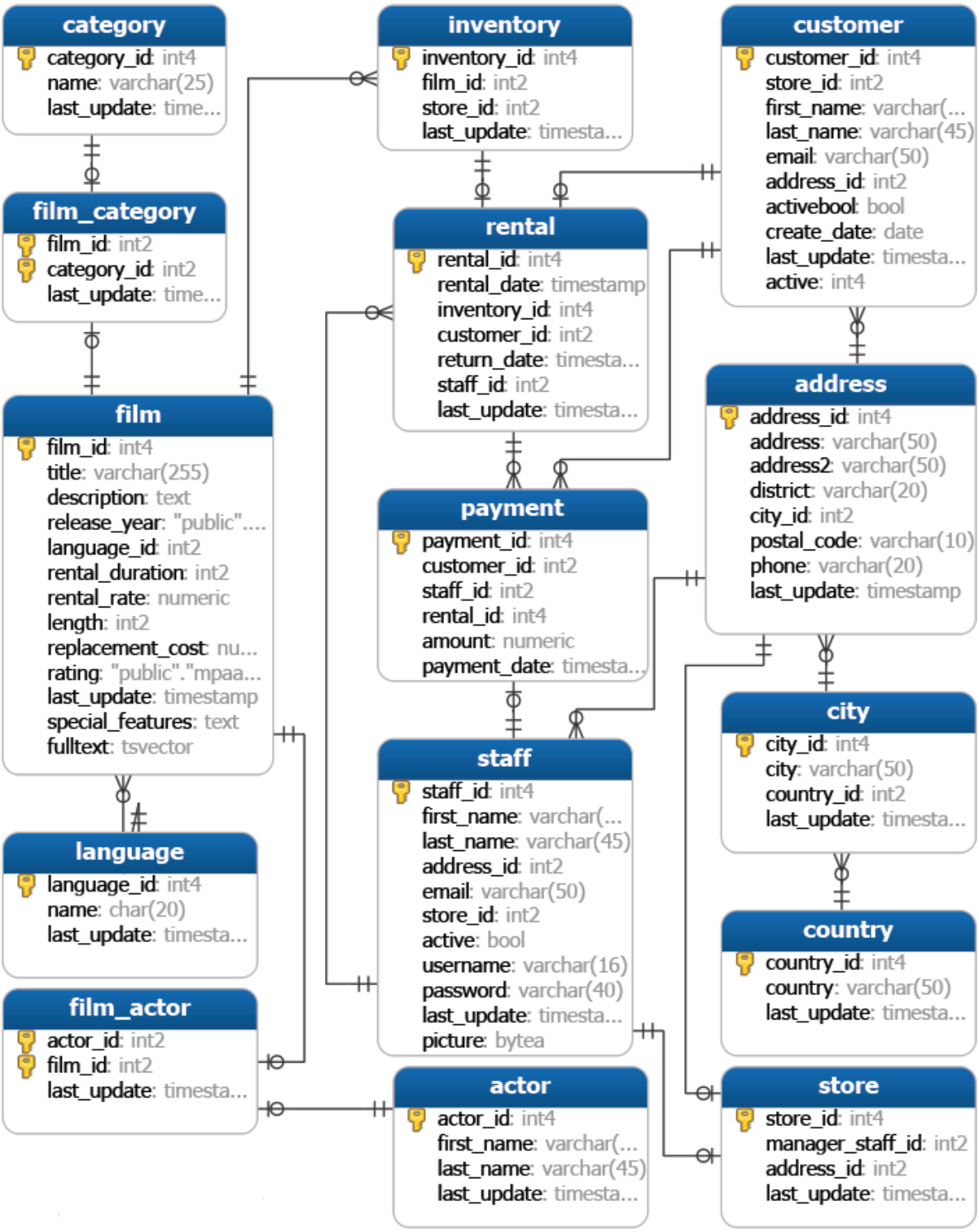
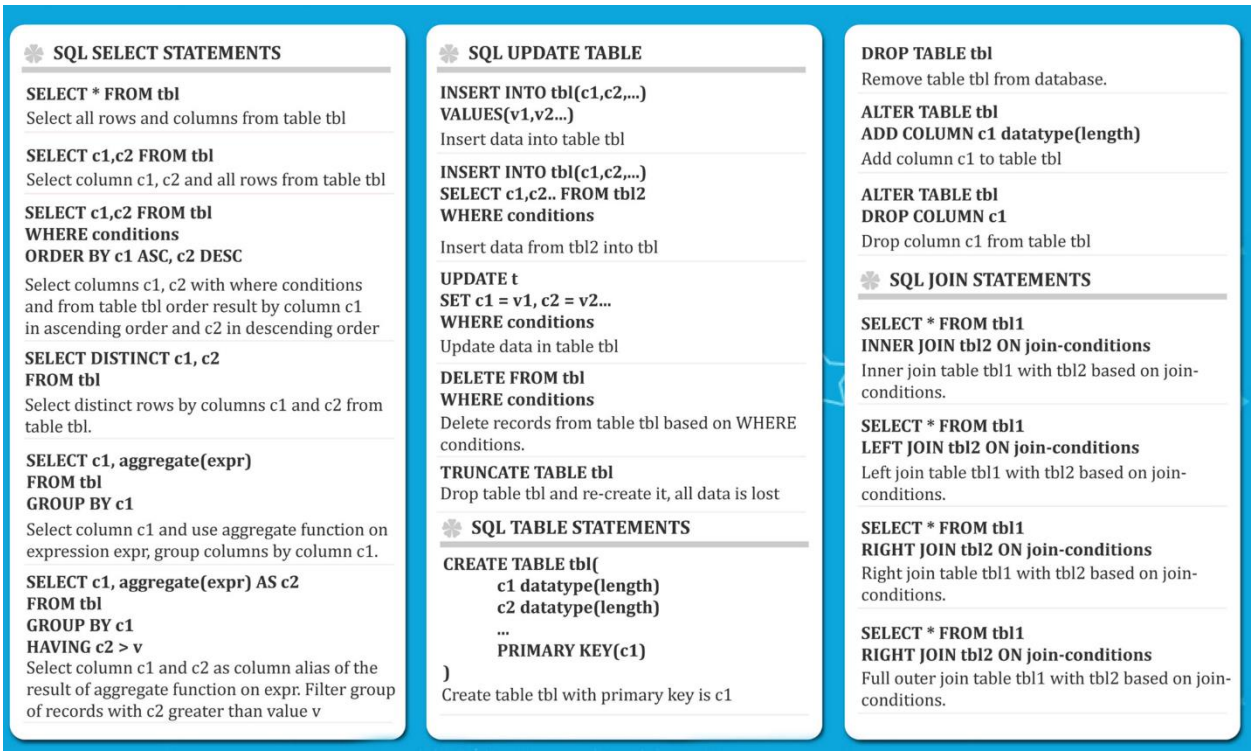

category
category_id: int4
name: varchar(25)
last_update: time...
inventory
inventory_id: int4
film_id: int2
store_id: int2
last_update: timesta...
customer
customer_id: int4
store_id: int2
first_name: varchar(...
last_name: varchar(45)
email: varchar(50)
address_id: int2
activebool: bool
create_date: date
last_update: timesta...
active: int4
film_category
film_id: int2
category_id: int2
last_update: time...
rental
rental_id: int4
rental_date: timestamp
inventory_id: int4
customer_id: int2
return_date: timesta...
staff_id: int2
last_update: timesta...
address
address_id: int4
address: varchar(50)
address2: varchar(50)
district: varchar(20)
city_id: int2
postal_code: varchar(10)
phone: varchar(20)
last_update: timestamp
film
film_id: int4
title: varchar(255)
description: text
release_year: "public"....
language_id: int2
rental_duration: int2
rental_rate: numeric
length: int2
replacement_cost: nu...
rating: "public"."mpaa...
last_update: timestamp
special_features: text
fulltext: tsvector
payment
payment_id: int4
customer_id: int2
staff_id: int2
rental_id: int4
amount: numeric
payment_date: timesta...
city
city_id: int4
city: varchar(50)
country_id: int2
last_update: timesta...
staff
staff_id: int4
first_name: varchar(...
last_name: varchar(45)
address_id: int2
email: varchar(50)
store_id: int2
active: bool
username: varchar(16)
password: varchar(40)
last_update: timesta...
picture: bytea
language
language_id: int4
name: char(20)
last_update: timesta...
country
country_id: int4
country: varchar(50)
last_update: timesta...
film_actor
actor_id: int2
film_id: int2
last_update: timesta...
actor
actor_id: int4
first_name: varchar(...
last_name: varchar(45)
last_update: timesta...
store
store_id: int4
manager_staff_id: int2
address_id: int2
last_update: timesta...



SQL SELECT STATEMENTS

SELECT * FROM tbl
Select all rows and columns from table tbl

SELECT c1,c2 FROM tbl
Select column c1, c2 and all rows from table tbl

SELECT c1,c2 FROM tbl
WHERE conditions
ORDER BY c1 ASC, c2 DESC
Select columns c1, c2 with where conditions and from table tbl order result by column c1 in ascending order and c2 in descending order

SELECT DISTINCT c1, c2
FROM tbl
Select distinct rows by columns c1 and c2 from table tbl.

SELECT c1, aggregate(expr)
FROM tbl
GROUP BY c1
Select column c1 and use aggregate function on expression expr, group columns by column c1.

SELECT c1, aggregate(expr) AS c2
FROM tbl
GROUP BY c1
HAVING c2 > v
Select column c1 and c2 as column alias of the result of aggregate function on expr. Filter group of records with c2 greater than value v

SQL UPDATE TABLE

INSERT INTO tbl(c1,c2,...)
VALUES(v1,v2...)
Insert data into table tbl

INSERT INTO tbl(c1,c2,...)
SELECT c1,c2.. FROM tbl2
WHERE conditions
Insert data from tbl2 into tbl

UPDATE t
SET c1 = v1, c2 = v2...
WHERE conditions
Update data in table tbl

DELETE FROM tbl
WHERE conditions
Delete records from table tbl based on WHERE conditions.

TRUNCATE TABLE tbl
Drop table tbl and re-create it, all data is lost

SQL TABLE STATEMENTS

CREATE TABLE tbl(
c1 datatype(length)
c2 datatype(length)
...
PRIMARY KEY(c1)
)
Create table tbl with primary key is c1

DROP TABLE tbl
Remove table tbl from database.

ALTER TABLE tbl
ADD COLUMN c1 datatype(length)
Add column c1 to table tbl

ALTER TABLE tbl
DROP COLUMN c1
Drop column c1 from table tbl

SQL JOIN STATEMENTS

SELECT * FROM tbl1
INNER JOIN tbl2 ON join-conditions
Inner join table tbl1 with tbl2 based on join-conditions.

SELECT * FROM tbl1
LEFT JOIN tbl2 ON join-conditions
Left join table tbl1 with tbl2 based on join-conditions.

SELECT * FROM tbl1
RIGHT JOIN tbl2 ON join-conditions
Right join table tbl1 with tbl2 based on join-conditions.

SELECT * FROM tbl1
RIGHT JOIN tbl2 ON join-conditions
Full outer join table tbl1 with tbl2 based on join-conditions.

- SELECT is the most common statement used, and it allows us to retrieve information from a table
- Later on will learn how to combine SELECT with other statement to perform more complex queries.
- Syntax for SELECT statement;
  - SELECT comlumn_name FROM table_name
- In general it is not good practice to use an asterisk (*) in the SELECT statement if you don't really need all columns.
- It will automatically query everything, wich increases traffic between the database sever and application, wich can slow down the retrieval of results.
- If you only need certain columns, do your best to only query for those columns
- Let's walk through some examples in our dvdrental database to get some practice

SELECT DISTINCT

- Sometimes a table contains a column that has duplicate values, and you may find yourself in a situation where you only want to list the unique/distinct values.
- The DISTINCT keyword can be used to return only the distinct values in a column
- The DISTINCT keyword operate on a column. The syntax looks like this:
  - SELECT DISTINCT column FROM table

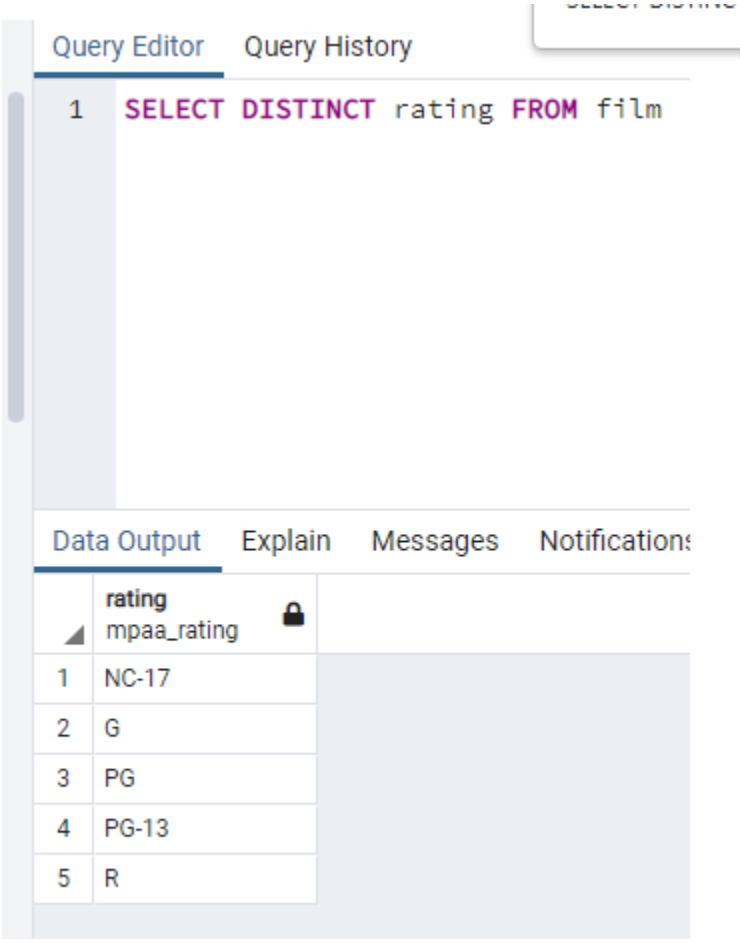
Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

```
SELECT DISTINCT choice
FROM color_table
```

Choice
Green
Yellow
Red

Here we have duplicate values, so we have selected distinct value from choice column.

- To clrarify which column DISTINCT is being applied to, you can also use parenthesis for clarity:
  - SELECT DISTINCT(column) FROM table
- It will work with or without parenthesis
- Later on when we learn about adding more calls such as COUNT and DISTINCT together, the parenthesis will be necessary.

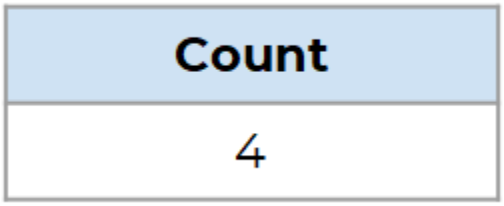


unique value in 'rating' column.

- SELECT DISTINCT column FROM table

COUNT

- The COUNT function return the number of input rows that match a specific condition of a query.
- We can apply COUNT on a specific column or just pass COUNT(*), we will soon see this should return the same result.



Count
4

- SELECT COUNT(name) FROM table;
- This is simply returning the number of rows in the table.
- In fact, it should be the same regardless of the column.
- SELECT COUNT(choice) FROM table;
- SELECT COUNT(*) FROM table;
- All return the same thing, since the original table had 4 rows
- Because of this COUNT by itself simply returns back a count of the number of rows in a table
- COUNT is much more useful when combined with other commands, such as DISTINCT

SELECT WHERE

- SELECT and WHERE are the most fundamental SQL statements and you will find yourself using them often!
- The WHERE statement allows us to specify conditions on columns for the row to be returned.
- Basic syntax example:
  - SELECT column1, column2
    FROM table
    WHERE conditions;
- The WHERE clause appears immediately after the FROM clause of the SELECT statement.
- The conditions are used to filter the rows returned from the SELECT statement.
- PostgreSQL provides a variety of standard operators to construct the conditionals
- Comparison Operators

Operator	Description
=	Equal
>	Greater than
<	Less Than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to

Image source : PostgreSQL

- = Equal
- > Greater than
- < Less than
- >= Greater than or equal to
- <= less than or equal to
- <> or != not equal to

- Logical Operators
  - Allow us to combine multiple comparison operators
    - AND
    - OR
    - NOT

```
SELECT email FROM customer

WHERE first_name = 'Nancy'

AND last_name = 'Thomas';
```

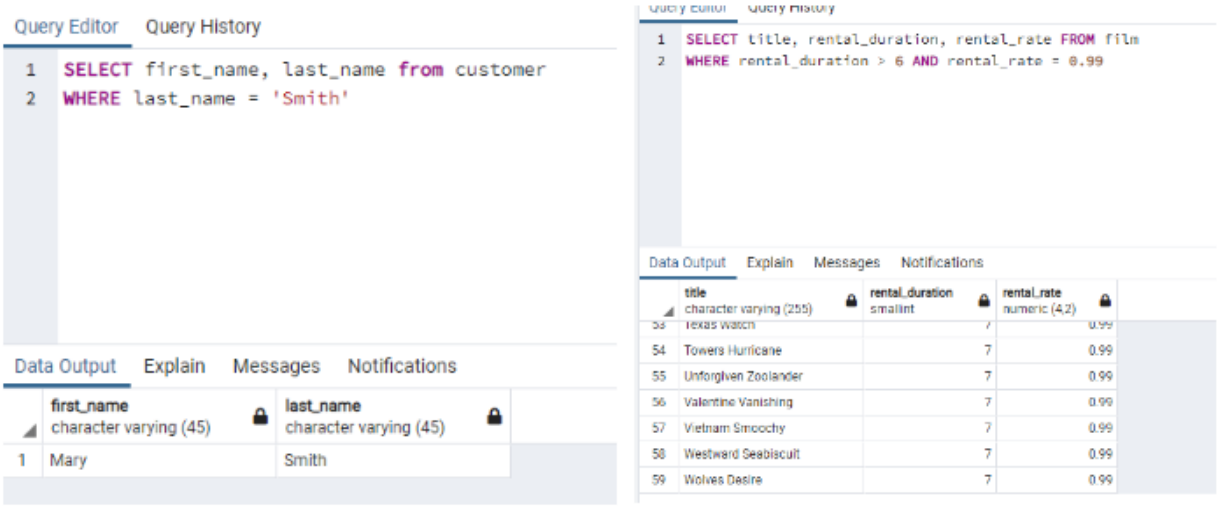


Image source : pgAdmin

SELECT description FROM film

WHERE title = 'Outlaw Hanky';

ORDER BY

- You may have noticed PostgreSQL sometimes returns the same request query results in a different order.
- You can use ORDER BY to sort rows based on a column value, in either ascending or descending order.
- Basic syntax for ODER BY
  - SELECT column_1,column_2
    FROM table
    ORDER BY column_1 ASC_DESC
- Notice ORDER BY towards the end of a query, since we want to do any selection and filtering first, before finally sorting.
  - SELECT column_1,column_2
    FROM table
    ORDER BY column_1 ASC/DESC

Query Editor | Query History

```
1  SELECT payment_id, rental_id, amount FROM payment
2  WHERE amount > 5.99
3  ORDER BY amount
4
```

Data Output | Explain | Messages | Notifications

	payment_id [PK] integer	rental_id integer	amount numeric (5,2)
1	30706	4271	6.99
2	27402	6271	6.99
3	23764	12575	6.99
4	23770	13193	6.99
5	31713	4198	6.99
6	23782	12842	6.99
7	23786	14643	6.99

Image source : pgAdmin

- Use ASC to sort in ascending order
- Use DESC to sort in descending order
- If you leave it blank, ORDER BY uses ASC by default.
- You can also ORDER by multiple columns
- This makes sense when one column has duplicate entries.

LIMIT

- The limit command allow us to limit the number of rows returned for a query.
- Useful for not wanting to return every single row in a table, but only view the top few rows to get an idea of the table layout.
- LIMIT also becomes useful in combination with ORDER BY
- LIMIT goes at the very end of a query request and is the last command to be executed.
- Example

```sql
SELECT * FROM payment
WHERE amount != 0.00
ORDER BY payment_date DESC
LIMIT 5;
```

output Messages Notifications

payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
31917	267	2	12066	7.98	2007-05-14 13:44:29.996577
31919	269	1	13025	3.98	2007-05-14 13:44:29.996577
31921	274	1	13486	0.99	2007-05-14 13:44:29.996577
31923	282	2	15430	0.99	2007-05-14 13:44:29.996577

```sql
SELECT customer_id FROM payment
ORDER BY payment_date ASC
LIMIT 10;

SELECT title,length FROM film
ORDER BY length ASC
LIMIT 5;
```

Data output | Messages | Notifications

	title character varying (255)	length smallint
1	Labyrinth League	46
2	Alien Center	46
3	Iron Moon	46
4	Kwai Homeward	46

BETWEEN

- The BETWEEN operator can be used to match a value against a range of values;
  - Value BETWEEN low and high
- The BETWEEN operator is the same as:
  - Value >= low AND value <= high
  - Value BETWEEN low AND high
- You can also combine BETWEEN with the NOT logical operator:
  - Value NOT BETWEEN low AND high
- The NOT BETWEEN operator is the same as:
  - Value < low OR value > high
  - Value NOT BETWEEN low AND high
- The BETWEEN operator is the same as:
  - Value >= low AND value <= high
  - Value BETWEEN low AND high

- The BETWEEN operator can also be used with dates. Note that you need to format dates in the ISO 8601 standard format, which is YYYY-MM-DD
- date BETWEEN '2007-01-01'
  AND '2007-02-01';
- when using BETWEEN operator with dates that also include timestamp information, pay careful attention to using BETWEEN versus <=,>= comparison operators, due to the fact that a datetime start at 0:00.
- Later on we will study more specific methods for datetime information types.

Query | Query History

```
1  SELECT * FROM payment
2  WHERE amount BETWEEN 8 AND 9;
3
4  SELECT COUNT(*) FROM payment
5  WHERE amount NOT BETWEEN 8 AND 9;
6
7  SELECT * FROM payment
8  WHERE payment_date BETWEEN '2007-02-01' AND '2007-02-15';
9
```

Data output | Messages | Notifications

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	17610	368	1	1186	0.99	2007-02-14 23:25:11.996577
2	17617	370	2	1190	6.99	2007-02-14 23:33:58.996577
3	17743	402	2	1194	4.99	2007-02-14 23:53:34.996577
4	17793	416	2	1158	2.99	2007-02-14 21:21:59.996577

Total rows: 27 of 27 | Query complete 00:00:00.086

IN

- In certain cases you want to check for multiple possible value options, for example, if a user's name shows up IN a list of known names.
- We can use the IN operator to create a condition that checks to see if a value in included in a list of multiple operators.
- The general syntax is:
  - Value IN (option1, option2,…, option_n)
- Example query:
  - SELECT color FROM table

WHERE color IN(‘red’, ‘blue’)

- SELECT color FROM table
  WHERE color NOT IN(‘red’, ‘blue’)

LIKE and ILIKE

- We’ve already been able to perform direct comparisons against strings, such as:
  - WHERE first_name = ‘John’
- But what if we want to match against a general pattern in a string?
  - All emails ending in ‘@gmail.com’
  - All names that begin with an ‘A’
- The LIKE operator allows us to perform pattern matching against string data with the use of wildcard characters:
  - Percent %
    Matches any sequences of characters
  - Underscore_
    Matches any single character
- All names that begin with an ‘A’
  - WHERE name LIKE ‘A%’
- All names that end with an ‘A’
  - WHERE name LIKE ‘%a’
- Notice that LIKE case-sensitive, we can use ILIKE which is case-insensitive
- Using the underscore allows us to replace just a single character
  - Get all Mission Impossible films
  - WHERE title LIKE ‘Mission impossible_’
- You can use multiple underscores
- Imagine we had version string codes in the format ‘Version#A4’, ‘Version#A4’, etc…
  - WHERE value LIKE ‘‘Version#_ _’,’

Query Editor | Query History

```
1  SELECT first_name, last_name, address_id FROM customer
2  WHERE first_name LIKE '_her%'
```

Data Output | Explain | Messages | Notifications

	first_name character varying (45)	last_name character varying (45)	address_id smallint
1	Cheryl	Murphy	63
2	Theresa	Watson	76
3	Sherry	Marshall	123
4	Sherri	Rhodes	302

'LIKE' example.

- We can also combine pattern matching operators to create more complex patterns
  - WHERE name LIKE '_her%'
    - Cheryl
    - Theresa
    - Sherri

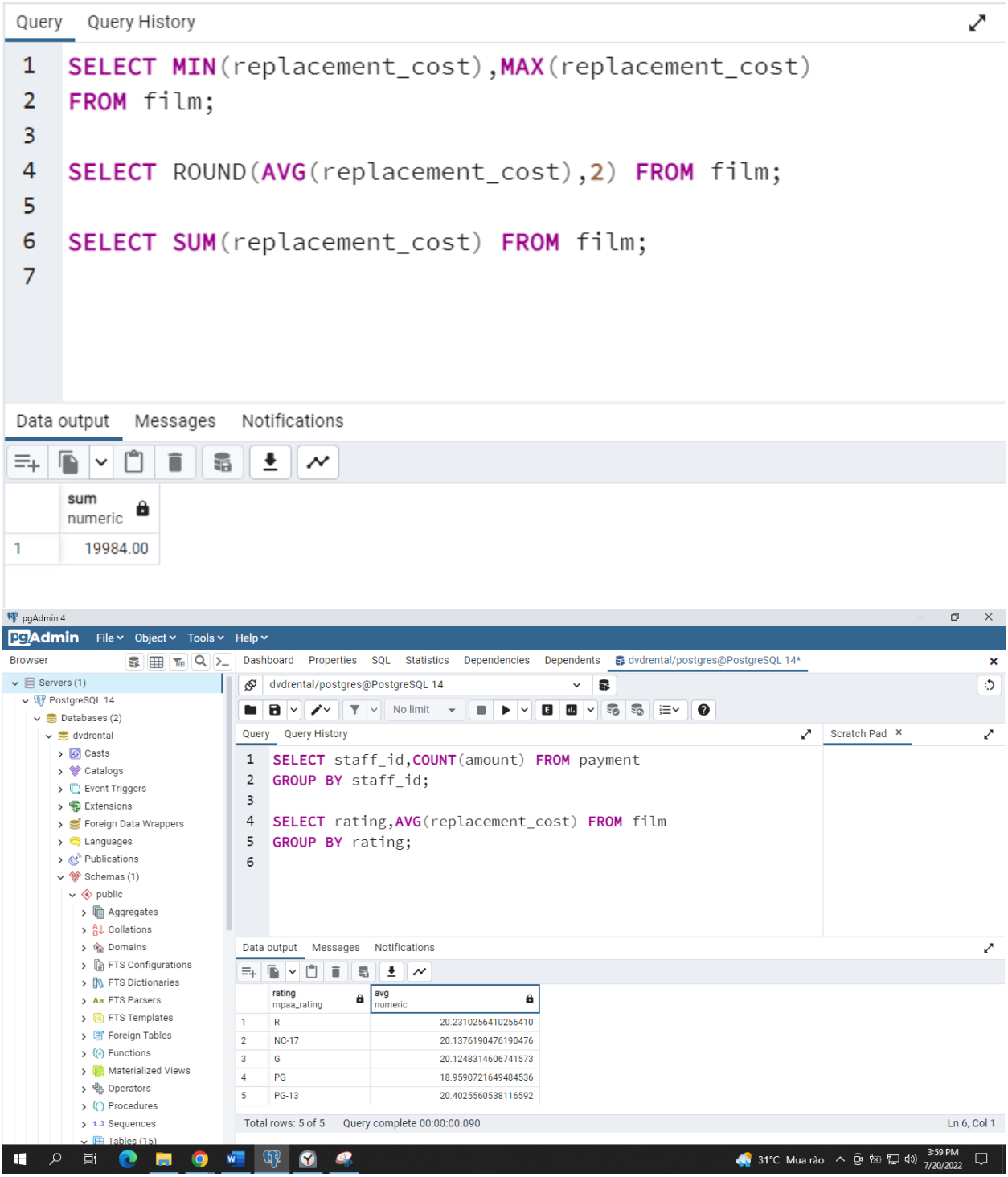
      _: chi co mot ki tu duoc phep o do

Aggregate Functions

- SQL provides a variety of aggregate function.
- The main idea behind an aggregate function is to take multiple inputs and return a single output
- Most Common aggregate Function:
  - AVG() – returns average value
  - COUNT() – returns number of value
  - MAX() – returns maximum value
  - MIN() – returns minimum value
  - SUM() – returns the sum of all value
- Aggregate function calls happened only in the SELECT clause or the HAVING clause
- Special notes

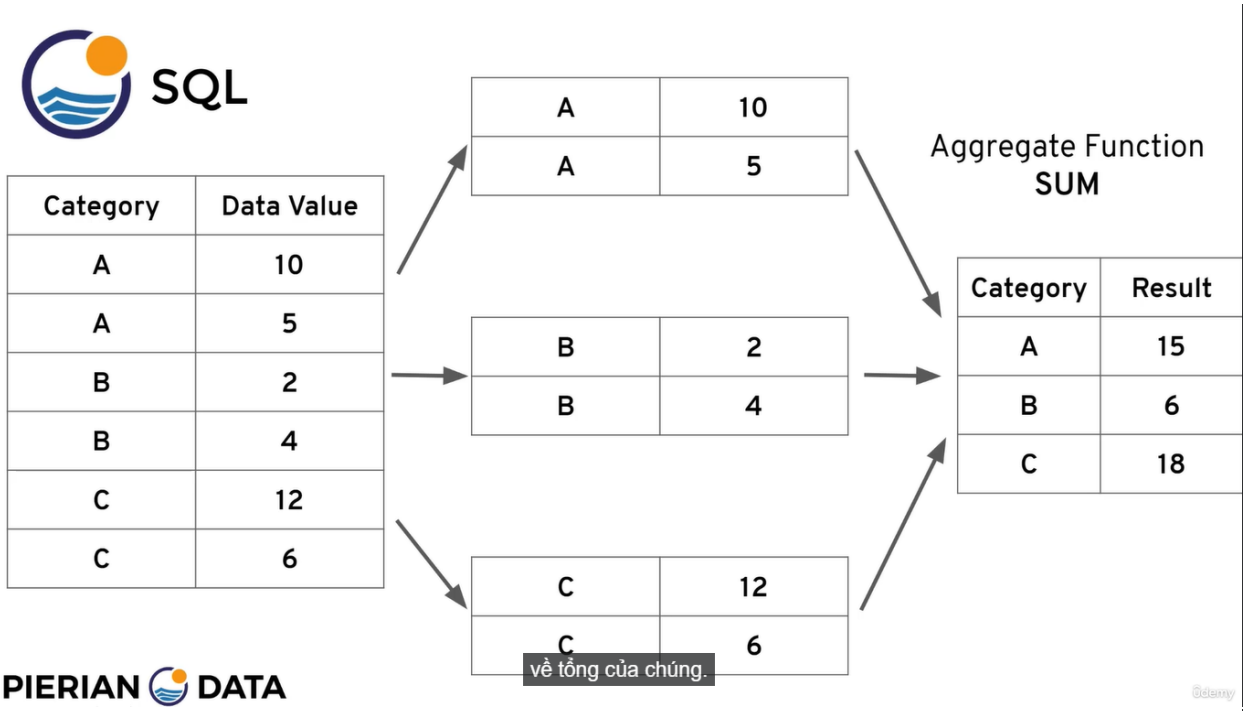
- AVG() returns a floating point value many decimal places (e.g 2.342418…)
  You can use ROUND() to specify precision after the decimal
- COUNT() simply returns the number of rows, which mean by convention we just use COUNT(*)

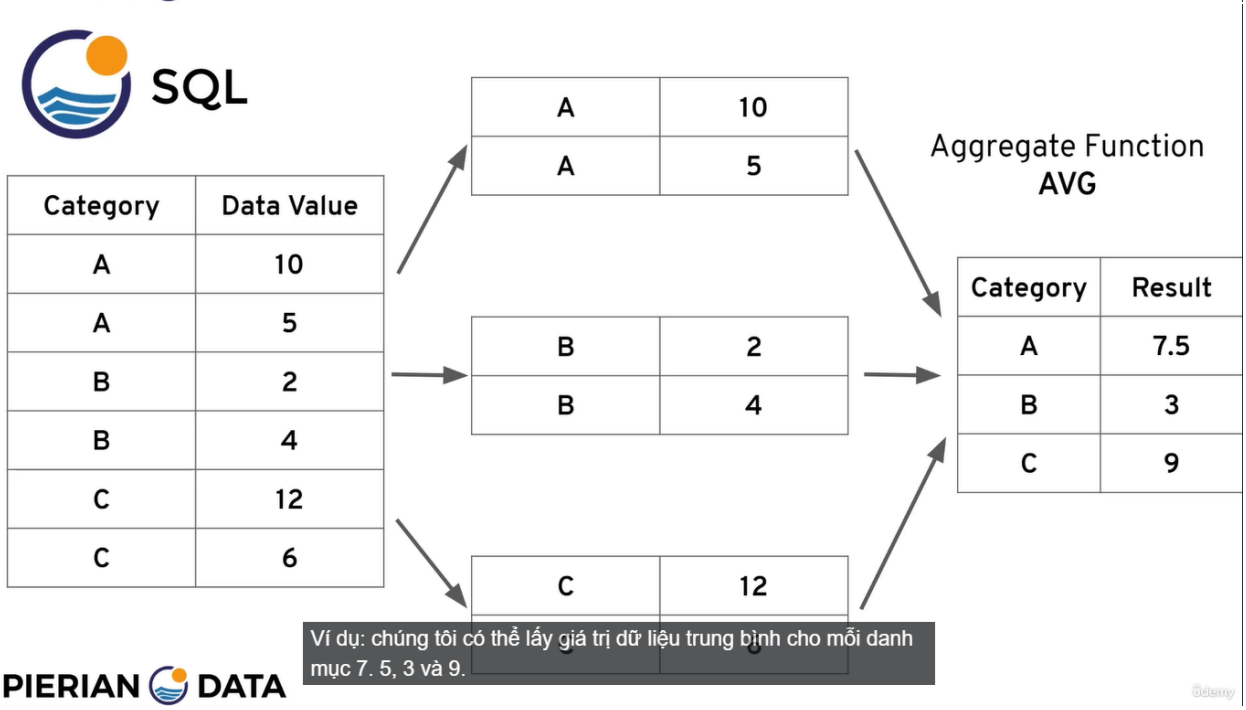
- Example



GROUP BY – Part one

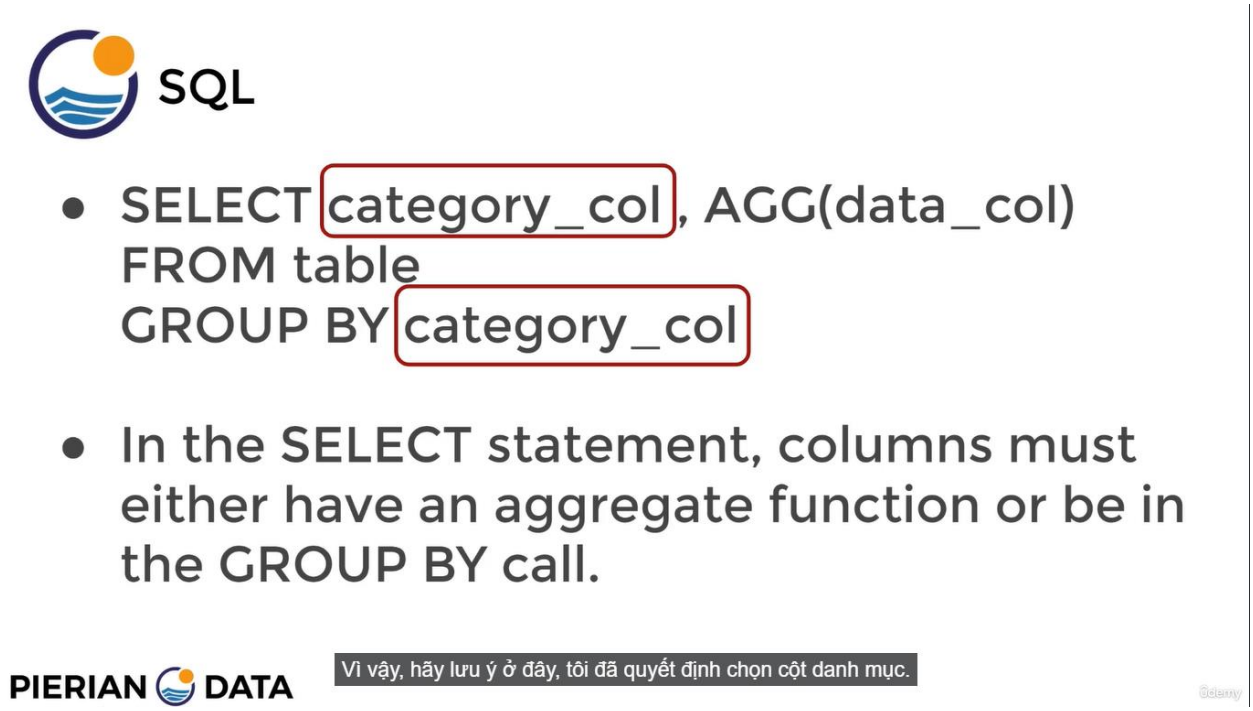
- GROUP BY will allow us to aggregate data and apply functions to better understand how data is distributed per category.
- GROUP BY allows us to aggregate columns per some category.
- Let's explore this idea with a simple example





- Syntax

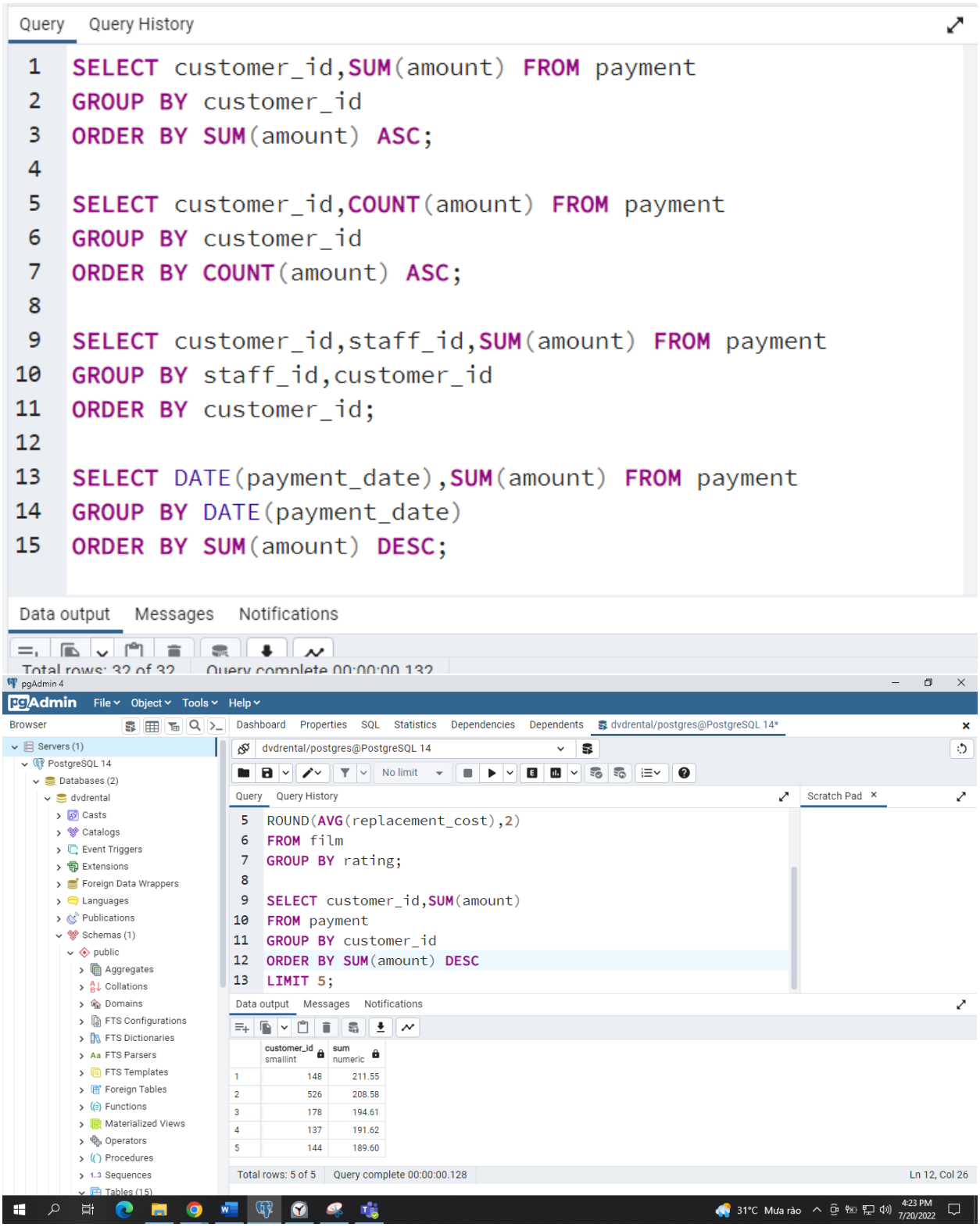
- SELECT category_col, ACG(data_col)
  FROM table
  WHERE category_col != ‘A’
  GROUP BY category_col



- The GROUP BY clause must appear right after a FROM or WHERE statement.
- In the SELECT statement, columns must either have an aggregate function or be in the GROUP BY call.
- SELECT company, division, SUM(sales)
  FROM finance_table
  GROUP by company,division
- SELECT company, division, SUM(sales)
  FROM finance_table
  WHERE division IN(‘marketing’, transport)
  GROUP BY company, division
- WHERE statements should not refer to the aggregation result, later on we will learn to use HAVING to filter on those result
- SELECT company, division, SUM(sales)
  FROM finance_table
  GROUP by company
  ORDER BY SUM(sales)
  LIMIT 5;

- If you want to sort results based on the aggregate, make sure to reference the entire function

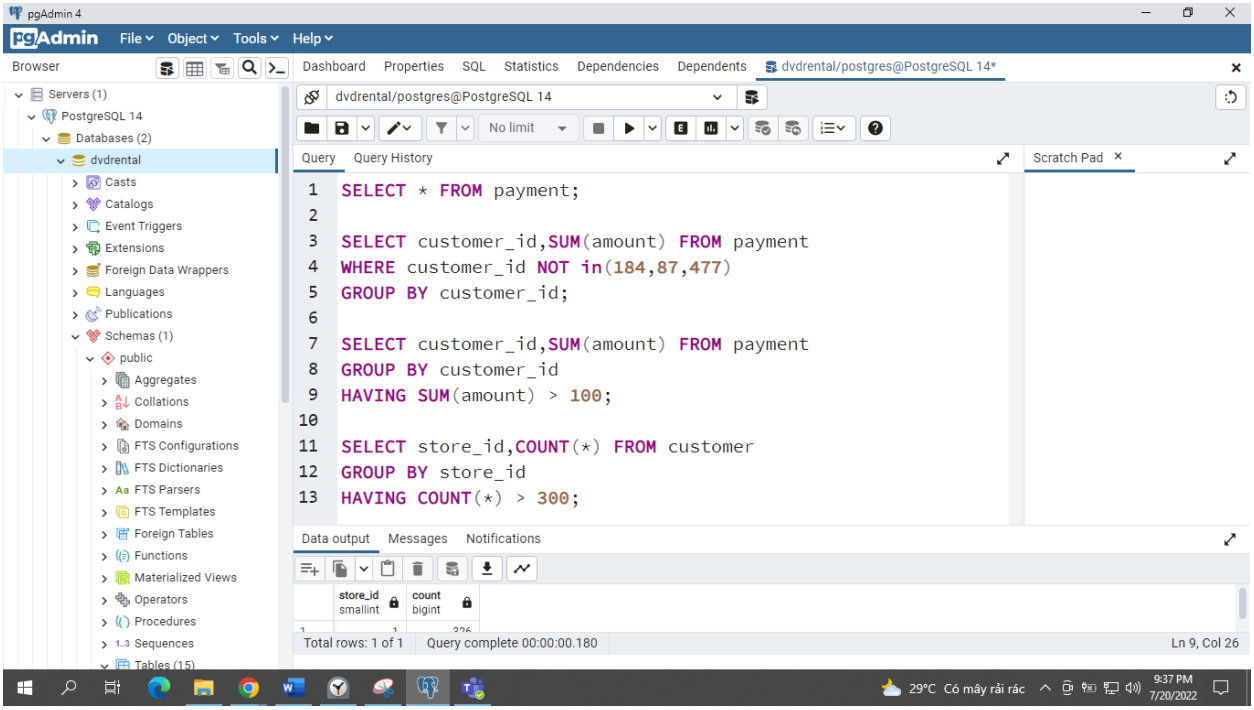
GROUP BY – Part two

- 

HAVING

- The HAVING clause allows us to filter after an aggregation has already taken place.

- Let’s take a look back at one of our previous examples.
- SELECT company, SUM(sales)
  FROM finance_table
  GROUP BY company
- We’ve already seen we can filter before executing the GROUP BY, but what if we want to filter based on SUM(sales)?
- SELECT company, SUM(sales)
  FROM finance_table
  WHERE company != ‘Google’
  GROUP BY company
- We can not use WHERE to filter based off of aggregate results, because those happen after a WHERE is executed
- SELECT company, SUM(sales)
  FROM finance_table
  WHERE company != ‘Google’
  GROUP BY company
  HAVING SUM(sales) >1000;
  HAVING allows us to use the aggregate result as a filter along with a GROUP BY



TEST 1
SELECT * FROM payment;

SELECT customer_id,SUM(amount) FROM payment
WHERE customer_id NOT in(184,87,477)
GROUP BY customer_id;

SELECT customer_id,SUM(amount) FROM payment

```
GROUP BY customer_id
HAVING SUM(amount) > 100;

SELECT store_id,COUNT(*) FROM customer
GROUP BY store_id
HAVING COUNT(*) > 300;

SELECT * FROM payment;
SELECT customer_id,COUNT(*) FROM payment
GROUP BY customer_id
HAVING COUNT(*) >= 40;

SELECT customer_id,SUM(amount) FROM payment
WHERE staff_id = '2'
GROUP BY customer_id
HAVING SUM(amount) > 100;

SELECT customer_id,SUM(amount)
FROM payment
WHERE staff_id = 2
GROUP BY customer_id
HAVING SUM(amount) > 110;

SELECT COUNT(*) FROM film
WHERE title LIKE 'J%';

SELECT first_name,last_name FROM customer
WHERE first_name LIKE 'E%'
AND address_id <500
ORDER BY customer_id DESC
LIMIT 1;
```

AS

- Before we learn about JOINs, let's quickly cover the AS clause which allows us to create an 'alias' for a column or result.
- Example syntax:
  - SELECT column AS new_name
    FROM table
  - SELECT SUM(column) AS new_name
    FROM table

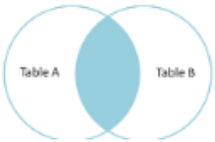
INNER JOIN

- What is a JOIN operation?

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

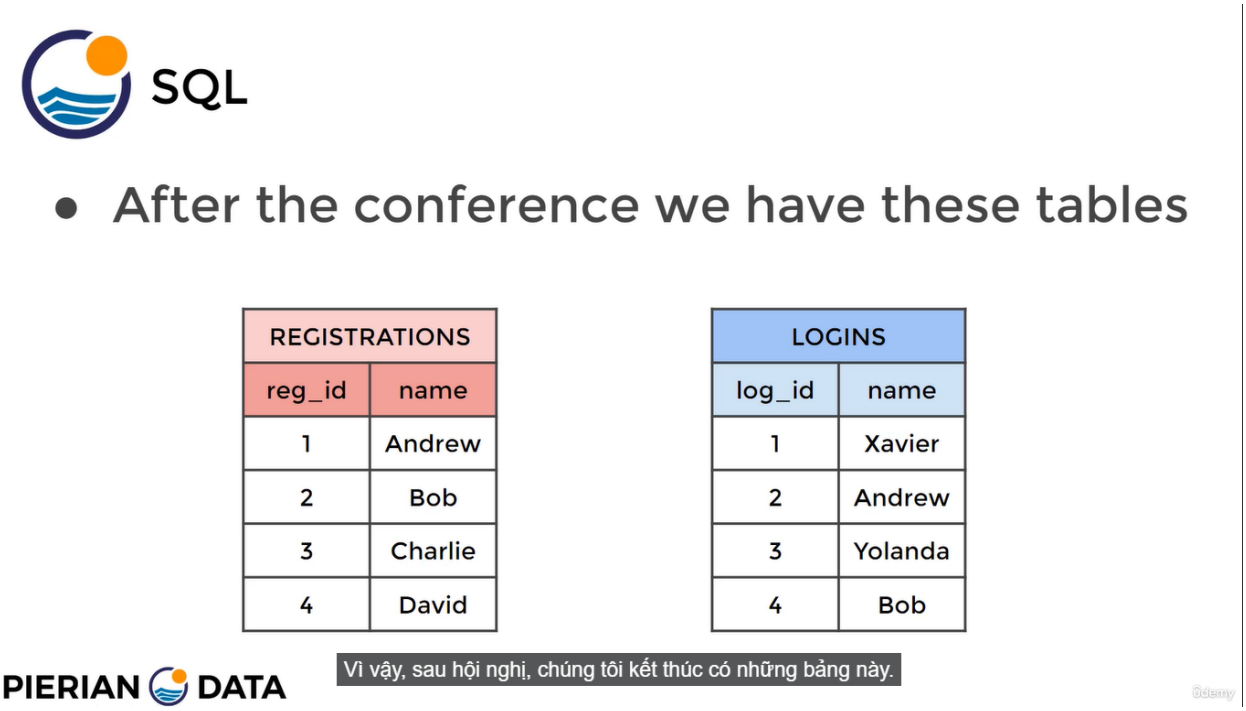
```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.col_match = TableB.col_match
```



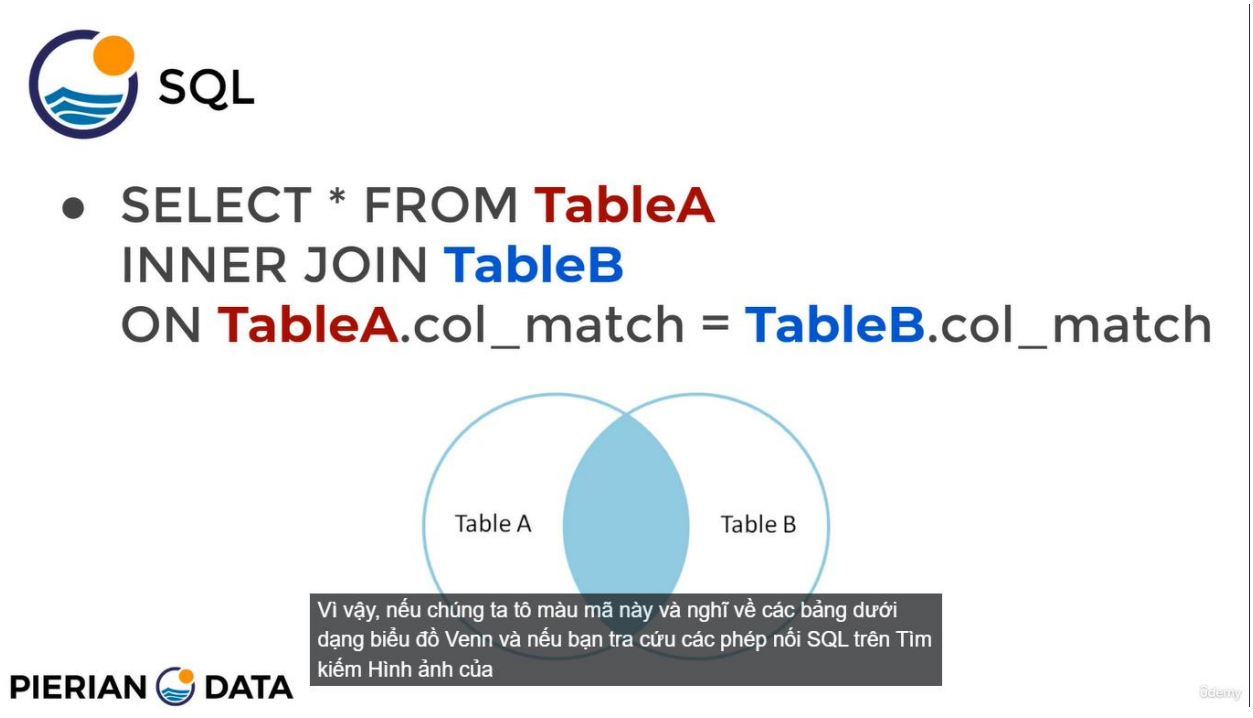

```
SELECT * FROM Registrations
INNER JOIN Logins
ON Registrations.name = Logins.name
```

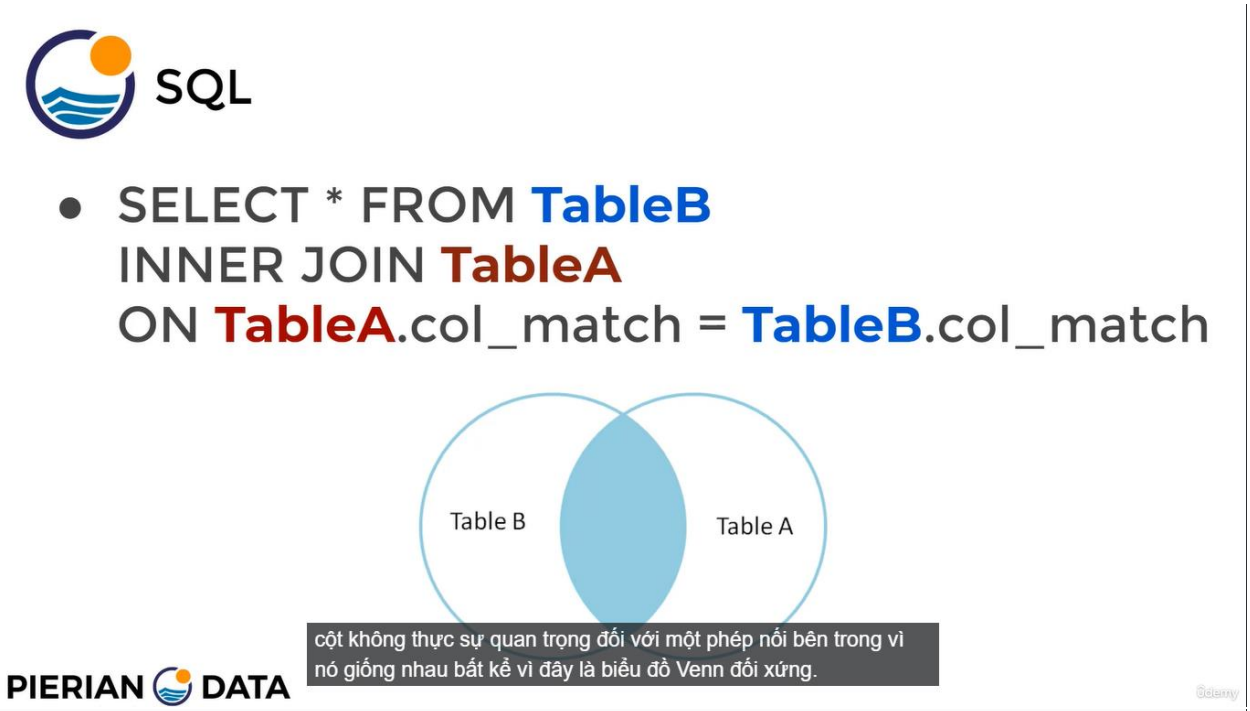
from above two images we can see the use of INNER JOIN.

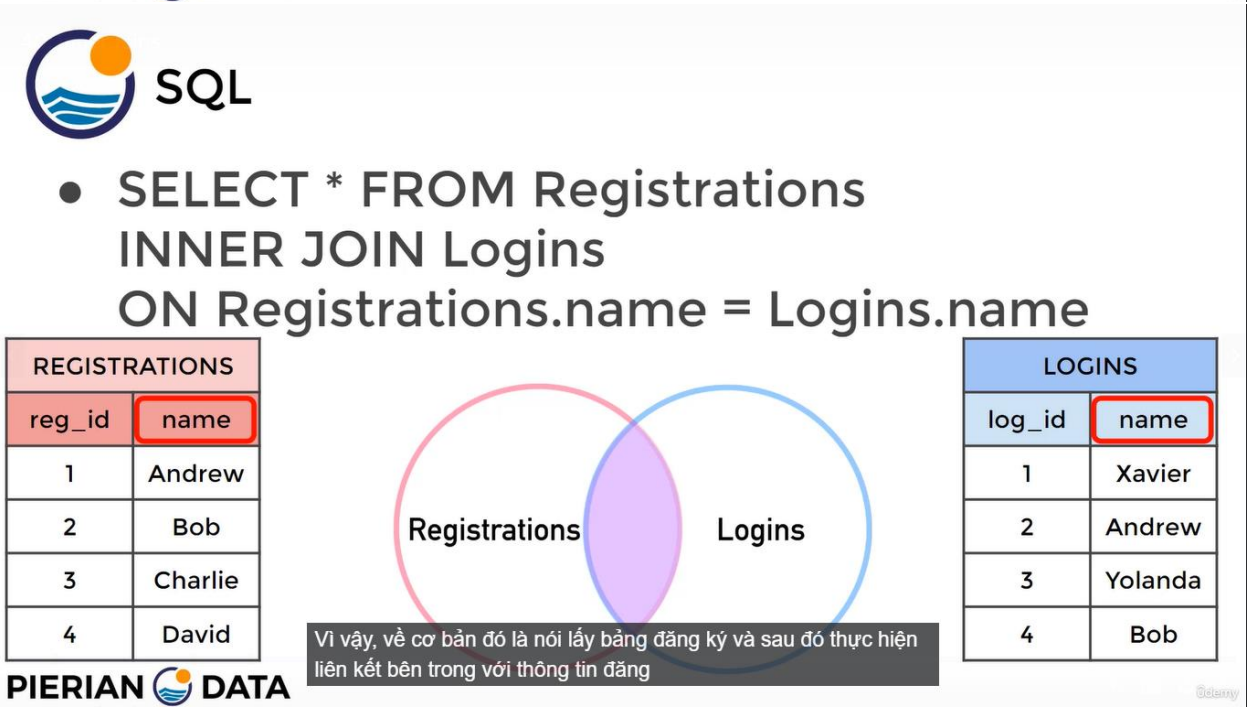
- JOINs allow us to combine multiple tables together
- The main reason for the different JOIN types is to decide how to deal with information only present in one of the joined tables.



- Syntax:
  - SELECT * FROM tableA
    INNER JOIN TableB
    ON TableA.col_match = TableB.col_match







- Remember that table order won't matter in an INNER JOIN
- Also if you see just JOIN without the INNER, PostgreSQL will treat it as an INNER JOIN.

FULL OUTER JOIN

- There are few different types of OUTER JOINs
- They will allow us to specify how to deal with values only present in one of the tables being joined

- These are the more complex JOINs, take your time when trying to understand them!

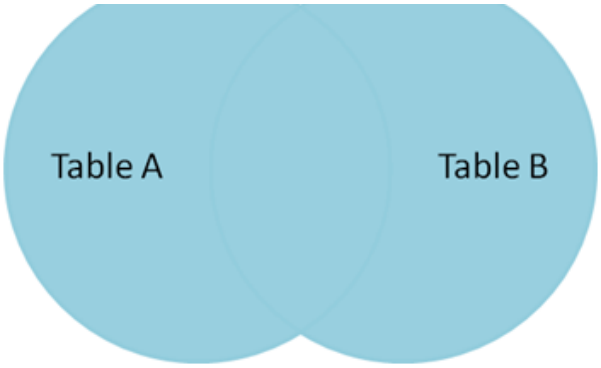


image source : PostgreSQL

SELECT * FROM Registrations FULL OUTER JOIN Logins
ON Registrations.name = Logins.name

REGISTRATIONS

reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

RESULTS

reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

LOGINS

log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

15

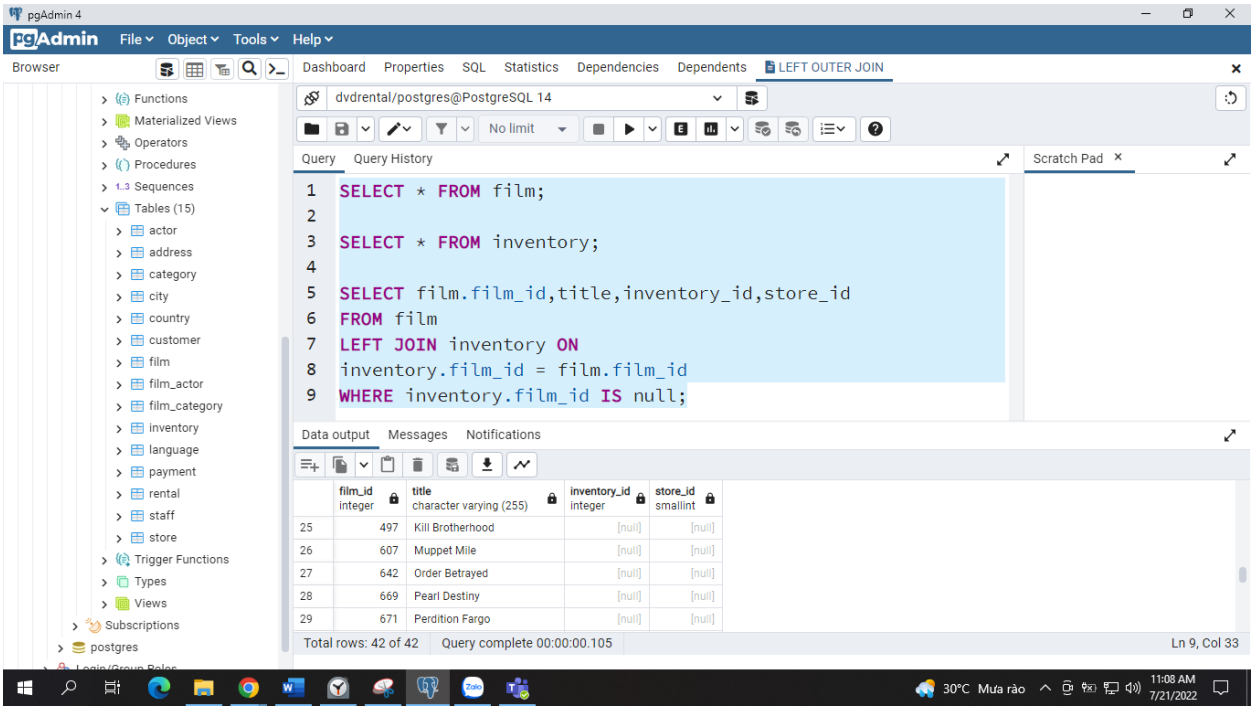
it has joined two table an alue is filled by null value.

- SELECT * FROM TableA
  FULL OUTER JOIN TableB
  ON TableA.col_match = TableB.col_match
- FULL OUTER JOIN with WHERE(Get rows unique to either table)
- SELECT * FROM TableA
  FULL OUTER JOIN TableB
  ON TableA.col_match = TableB.col_match
  WHERE TableA.id IS null OR
  TableB.id IS null

LEFT OUTER JOIN

- A LEFT OUTER JOIN results in the set of records that are in the left table, if there is no match with the right table, the results are null.
- Later on we will learn how to add WHERE statements to further modify a LEFT OUTER JOIN

- SELECT * FROM TableA
  LEFT OUTER JOIN TableB
  ON TableA.col_match = TableB.col_match
- LEFT OUTER JOIN With WHERE
  SELECT * FROM TableA
  LEFT OUTER JOIN TableB
  ON TableA.col_match = TableB.col_match
  WHERE TableB.id IS null



RIGHT JOINS

- A RIGHT JOIN is essentially the same as a LEFT JOIN, except the tables are switched.
- This would be the same as switching the table order in a LEFT OUTER JOIN.
- Let's quickly see some examples of a RIGHT JOIN.
- SELECT * FROM TableA
  RIGHT OUTER JOIN TableB
  ON TableA.col_match = TableB.col_match
  WHERE TableA.id IS null;
- It is up to you and how you have the tables organized 'in your mind' when it comes to choosing a LEFT vs RIGHT join, since depending on the table order you specify in the JOIN, you can perform duplicate JOINs with either method.

UNIONS

- The UNION operator is used to combine the result-set of two or more SELECT statements.

- It basically serves to directly concatenate two results together, essentially ‘pasting’ them together.
- Syntax:
  - SELECT column_name(s) FROM table1
    UNION
    SELECT column_name(s) FROM table2;
- Example:
  SELECT * FROM Sales2021_Q1
  UNION
  SELECT * FROM Sales2021_Q2
  ORDER BY name;

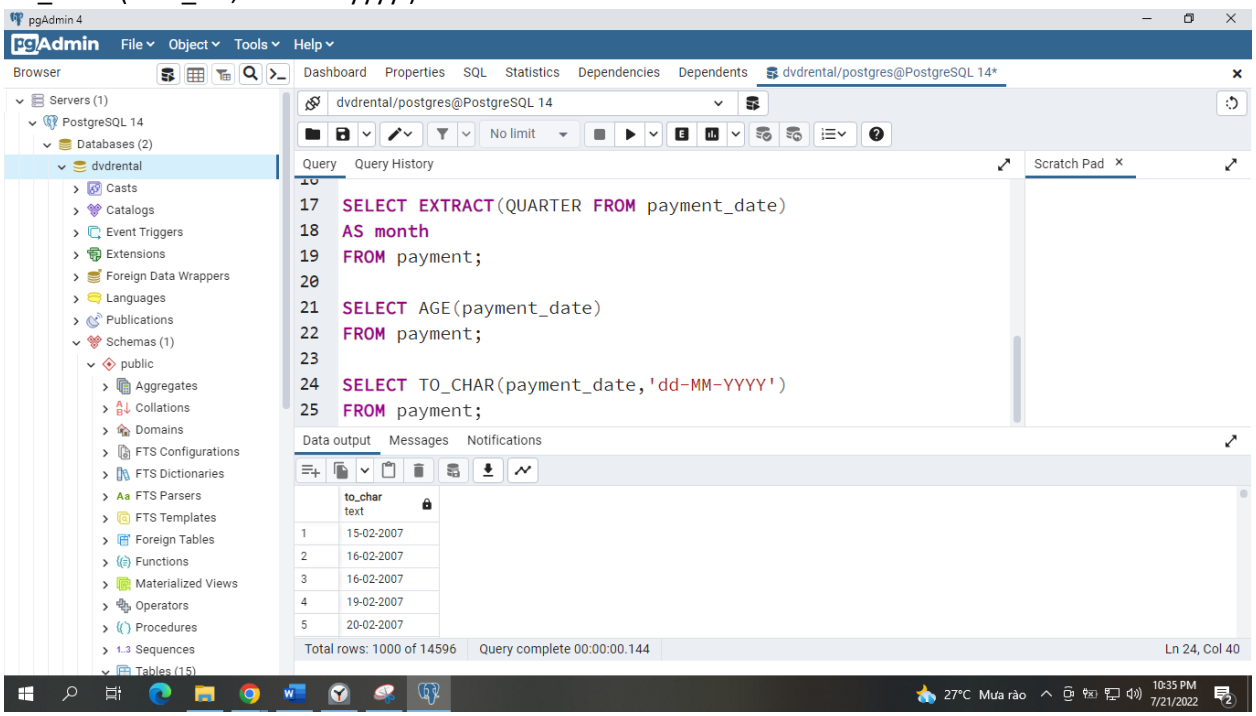
Timestamps and Extract

- In part one, we will go over a few commands that report back time and date information.
- These will be more useful when creating our own tables and databases, rather than when querying a database
- We’ve already seen that PostgreSQL can hold date and time information:
  - TIME – Contains only time
  - DATE – Contains only date
  - TIMESTAMP – Contains date and time
  - TIMESTAMPTZ – Contains date, time and timezone.
- Careful considerations should be made when designing a table and database and choosing a time data type.
- Depending on the situation you may or may not need the full level of TIMSTAMPTZ
- Remember, you can always remove historical information, but you can’t add it!
- Let’s explore function and operators related to these specific data types:
  - TIMEZONE
  - NOW
  - TIMEOFDAY
  - CURRENT_TIME
  - CURRENT_DATE

Timestamps and Extract

- Let’s explore extracting information from a time based data type using:
  - EXTRACT()
  - AGE()
  - TO_CHAR()
- EXTRACT()
  - Allows you to ‘extract’ or obtain a sub-component of a date value
    - YEAR
    - MONTH
    - DAY
    - WEEK

    - ➢ QUARTER
- EXTRACT()
    - EXTRACT(YEAR FROM date_col)
- AGE()
    - Calculates and returns the current age given a timestamp
    - Useage:
      AGE(date_col)
    - Returns
      13 year 1 mon 5 days 01:34……
- TO_CHAR()
    - General function to convert data type to text
    - Useful for timestamp formatting
    - Usage
    - TO_CHAR(date_col, ‘mm-dd-yyyy’)



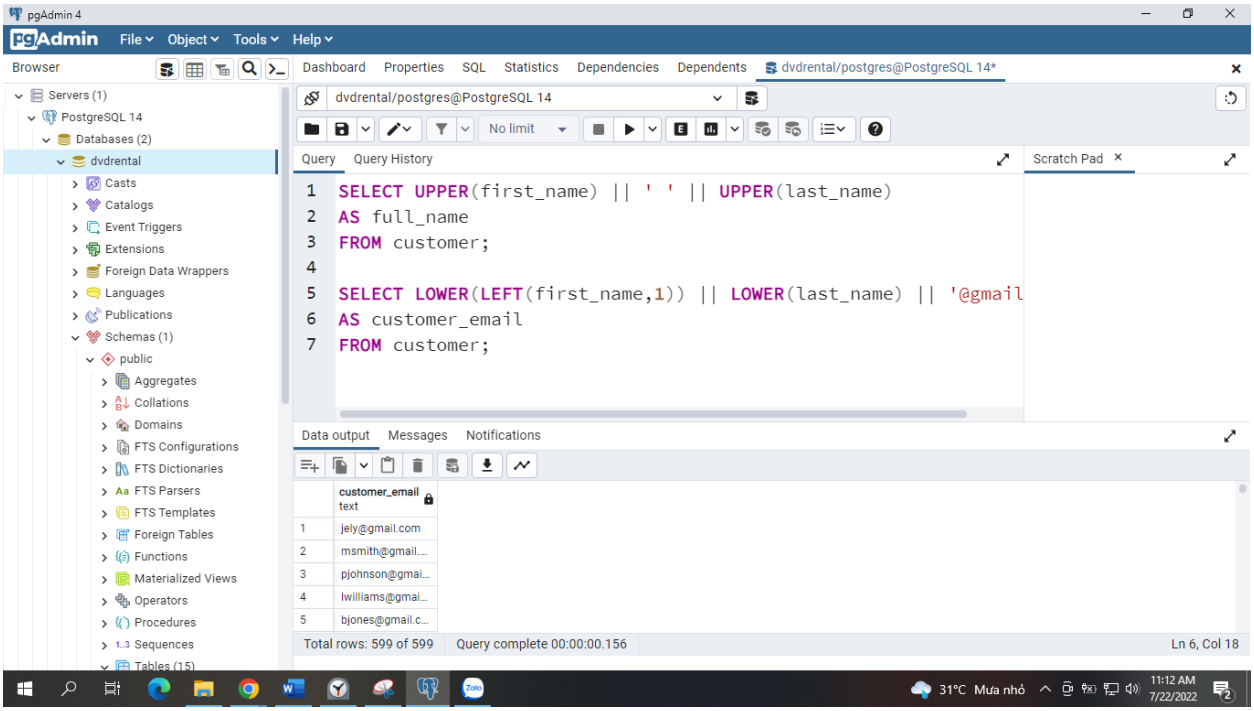
https://www.postgresql.org/docs/12/functions-formatting.html

Mathematical Functions

- https://www.postgresql.org/docs/9.5/functions-math.html

String Functions and Operations

- PostgreSQL also provides a variety of string functions and operators that allow us to edit, combine, and alter text data columns

- 

SUBQUERY

- In this lecture we will discuss how to perform a subquery as well as the EXISTS function.
- A sub query allows you to construct complex queries, essentially performing a query on the results of another query.
- The syntax is straightforward and involves two SELECT statements.
  - SELECT student,grade
    FROM test_score
    WHERE grade > (SELECT AVG(grade)
    FROM test_score)
  - SELECT student,grade
    FROM test_score
    WHERE grade > (70);
  -
- The subquery is performed first since it is inside the parenthesis.
- We can also use the in operator in conjunction with a subquery to check against multiple results returned.
- The EXISTS operator is used to test for existence of rows in a subquery.
- Typically a subquery is passed in the EXISTS() function to check if any rows are returned with the subquery.
- Typical Syntax
  - SELECT column_name
    FROM table_name
    WHERE EXISTS
    (SELECT column_name FROM

Table_name WHERE condition);

Creating Databases and Tables

- We've focused on querying and reading data from existing databases and tables
- Let's now shift our focus to creating our own databases and tables.

DATA TYPES

- We've already encountered a variety of data types, let's quickly review the main data types in SQL
- Boolean
  - True or False
- Character
  - Char, varchar, and text
- Numeric
  - Integer and floating- point number
- Temporal
  - Date, time, timestamp, and interval
- UUID
  - Universally Unique Identifiers
- Array
  - Stores and array of strings, numbers, etc.
- JSON
- Hstore key-value pair
- Special type such as network address and geometric data
- When creating database tables, you should carefully consider which data types should be used for the data to be stored.
- Review the documentation to see limitations of data types: https://www.postgresql.org/docs/current/datatype.html
- Based on the limitations, you may think it makes sense to store is as a BIGINT data type, but we should really be thinking what is best for the situation.
- We don't perform arithmetic with numbers, so it probably makes more sense as a varchar data type instead.
- When creating a database and table, take your time to plan for long term storage
- Remember you can always remove historical information you've decided you aren't using, but you can't go back in time to add in information!

Primary and Foreign Keys

- A primary key is column or a group of columns used to identify a row uniquely in a table.
- For example, in our dvdrental database we saw customers had a unique, non-null customer_id column as their primary key.

- Primary keys are also important since they allow us to easily discern what columns should be used for joining tables together.
- Notice its integer based and unique
- A foreign key is a field or group of fields in a table that uniquely identifier a row in another table.
- A foreign key is defined in a table that references the primary key of the another table.
- The table that contains the foreign key is called referencing table or child table.
- The table to which the foreign key references is called referenced table or parent table.
- A table can have multiple foreign keys depending on its relationships with other tables.
- You may begin to realize primary key and foreign key typically make good column choices for joining together two or more tables.
- When creating table and defining columns, we can use constraints to define column as being a primary key, or attaching a foreign key relationship to another table.

CONSTRAINTS

- Constraints are the rules enforced on data columns on table.
- These are used to prevent invalid data from being entered into the database.
- This ensures the accuracy and reliability of the data in the database.
- Constraints can be divided into two main categories:
  - Column constraints
    Constrains the data in a column to adhere to certain conditions
  - Table constraints
    Applied to entire table rather than to an individual column
- The most common constraints used:
  - Not null Constraint
    Ensures that a column cannot have NULL value.
  - UNIQUE Constraint
    Ensures that all values in column are different.
- The most common constraints used:
  - PRIMARY Key
    Uniquely identifies each row/record in a database table
  - FOREIGN Key
    Constraints data based on columns in other tables.
  - CHECK Constraint
    Ensures that all value in a column satisfy certain conditions.
  - EXCLUSION Constraint
    Ensures that if any two rows are compared on the specified column or expression using the specified operator, not all of these comparisons will return TRUE.
- Table Constraints
  - CHECK(condition)
    - To check a condition when inserting or updating data.
  - REFERENCES
    - To constrain the value stored in the column that must exist in a column in another table.

- UNIQUE(column_list)
  Forces the values stored in the columns listed inside the parentheses to be unique.
- PRIMARY KEY(column_list)
  Allows you to define the primary key that consists of multiple columns

CREATE TABLE

- Syntax:
  - CREATE TABLE table_name (
    Column_name type column_constraint,
    Column_name type column_constraint,
    Table_constraint table_constraint
)INHERITS existing_table_name;
- SERIAL
  - In PostgreSQL, a sequence is a special kind of database object that generates a sequence of integers.
  - A sequence is often used as the primary key column in a table.
  - It will create a sequence object and set the next value generated by the sequence as the default value for the column.
  - This is perfect for a primary key, because it logs unique integer entries for you automatically upon insertion.
  - If a row is later removed, the column with the SERIAL data type will not adjust, marking the fact that a row was removed from the sequence, for example
    1,2,3,5,6,7
    You know row 4 was removed at some point
- Example
  CREATE TABLE player(
  Player_id SERIAL PRIMARY KEY,
  Column_name type column_constraint
  );

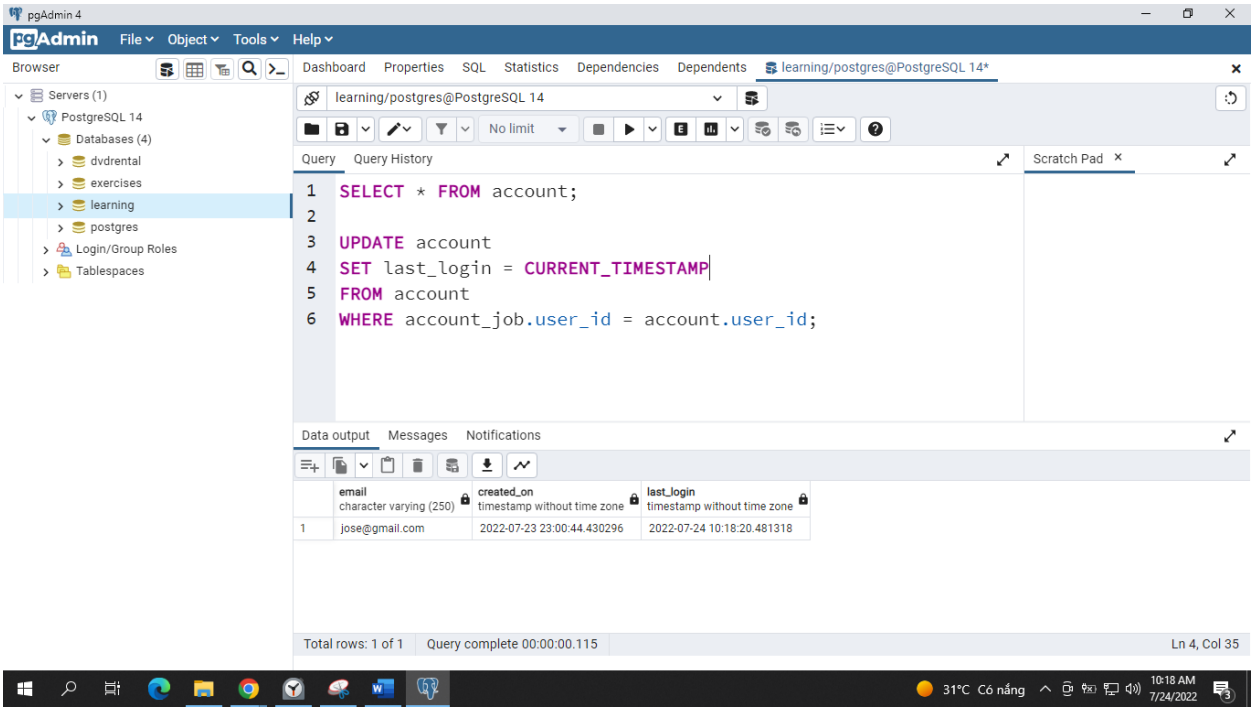
INSERT

- INSERT allows you to add in rows to a table.
- General syntax
  - INSERT INTO table (column1, column2, ….)
    VALUES
    (value1, value2,….),
    (value1, value2,….),….;
- Syntax for inserting values from another table:
  - INSERT INTO table(column1, column2,….)
    SELECT column1, column2,….
    FROM another_table
    WHERE condition;

- Keep in mind, the inserted row values must match up for the table, including contrainsts
- SERIAL columns do not need to be provided a value.

UPDATE TABLE

- Syntax
  - UPDATE table
    SET column1 = value1,
        Column2 = value,…
    WHERE
        Condition;
  - Example:
    UPDATE account
    SET last_login = CURRENT_TIMESTAMP
        WHERE last_login IS NULL;
- Reset everything without WHERE condition
  - UPDATE account
    SET last_login = CURRENT_TIMESTAMP
- Set based on another column
  - UPDATE account
    SET last_login = created_on
- Using another table's values(UPDATE join)
  - UPDATE TableA
    SET original_col = TableB.new _col
    FROM tableB
    WHERE tableA.id = TableB.id
- Return affected rows
  - UPDATE account
    SET last_login = created_on
    RETURNING account_id, last_login



DELETE TABLE

- We can use the DELETE clause to remove rows from a table.
- For example:
  - DELETE FROM table
    WHERE row_id = 1
- We can delete rows based on their presence in other tables
- For example:
  - DELETE FROM tableA
    USING tableB
    WHERE tableA.id = TableB.id
- We can delete all rows from a table
- For example:
  - DELETE FROM table
- Similar to UPDATE command, you can also add in a RETURNING call to return rows that were removed.

ALTER

- The ALTER clause allows for changes to an existing table structure, such as:
  - Adding, dropping, or renaming columns
  - Changing a column's data type
  - Set DEFAULT values for a column
  - Add CHECK constraints
  - Rename table

- General syntax
  - ALTER TABLE table_name action
- Adding columns
  - ALTER TABLE table_name
    ADD COLUMN new_col TYPE
- Removing Columns
  - ALTER TABLE table_name
    DROP COLUMN col_name
- Alter constraints
  - ALTER TABLE table_name
    ALTER COLUMN col_name
    SET DEFAULT value

https://www.postgresql.org/docs/current/sql-altertable.html

DROP

- DROP allows for the complete removal of a column in a table.
- In PostgreSQL this will also automatically remove all of its indexes and constraints involving the column.
- However, it will not remove columns used in views, triggers, or stored procedures without the additional CASCADE clause
- General syntax
  - ALTER TABLE table_name
    DROP COLUMN col_name
- Remove all dependencies
  - ALTER TABLE table_name
    DROP COLUMN col_name CASCADE
- Check for existence to avoid error
  - ALTER TABLE table_name
    DROP COLUMN IF EXISTS col_name
- Drop multiple columns
  - ALTER TABLE table_name
    DROP COLUMN col_name,
    DROP COLUMN col_two

CHECK CONSTRAINT

- The CHECK constraint allow us to create more customized constraints that adhere to a certain condition.
- Such as making sure all inserted integer values fall below a certain threshold
- General syntax
  - CREATE TABLE example(
    ex_id SERIAL PRIMARY KEY,

```
age SMALLINT CHECK(age > 21),
parent_age SMALLINT CHECK (
parent_age > age)
);
```

CASE

- We can use the CASE statement to only execute SQL code when certain conditions are met.
- This is very similar to IF/ELSE statement in other programming
- There are two main ways to use a CASE statement, either a general CASE or a CASE expression
- Both methods can lead to the same result
- Let's first show the syntax for a "general" CASE.
- Syntax general
  - ```
    CASE
        WHEN condition1 THEN result1
        WHEN condition2 then result2
        ELSE some_other _result

    END
    ```

- Example:
  ```
  SELECT a,
  CASE WHEN a = 1 THEN 'one'
  WHEN a = 2 THEN 'two'
  ELSE 'other' AS label
  END
  FROM test;
  ```
- ->The CASE expression syntax first evaluates an expression then compares the result with each value in the WHEN clauses sequentially.
- CASE Expression Syntax
  - ```
    CASE expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ELSE some_other_result
    END
    ```
- example — Rewriting our previous example:
  - ```
    SELECT a,
        CASE a WHEN 1 THEN 'one'
                    WHEN 2 THEN 'two'
            ELSE 'other'
        END
    FROM test;
    ```

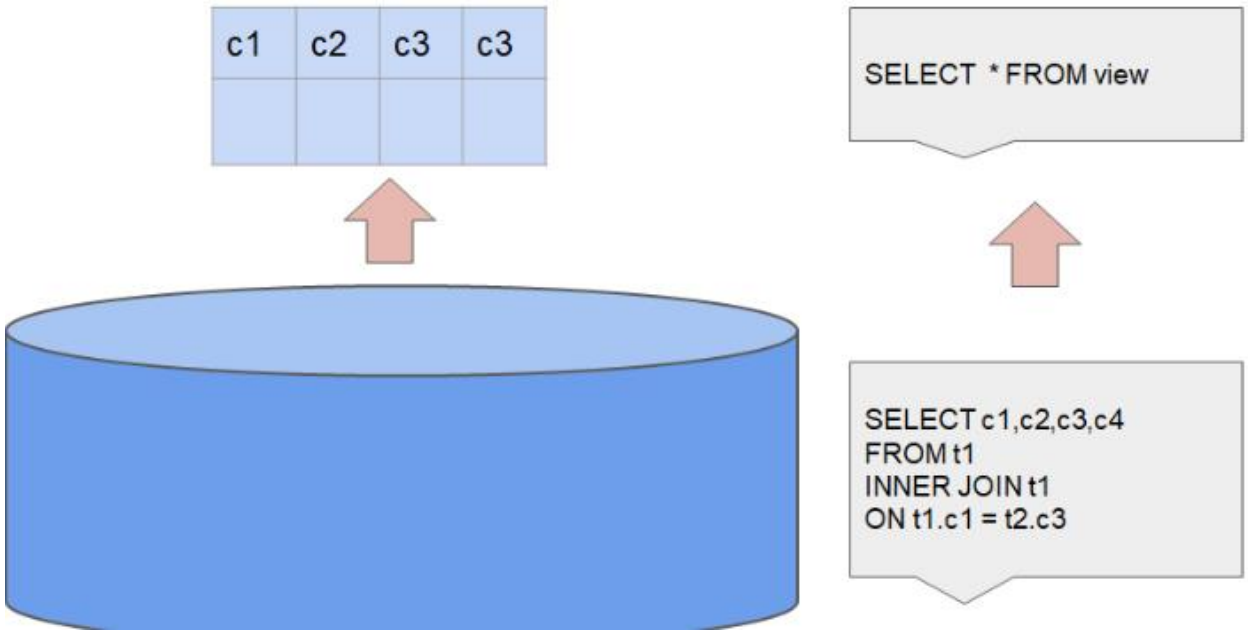
CAST

- The CAST operator let's you convert from one data type into another.
    ```

- Keep in mind not every instance of a data type can be CAST to another data type, it must be reasonable to convert the data, for example '5' to an integer will work, 'five' to an integer will not.
- Syntax for CAST function
  - SELECT CAST('5' AS INTEGER)
- PostgreSQL CAST operator
  - SELECT '5'::INTEGER
- Keep in mind you can then use this in a SELECT query with a column name instead of a single instance.
  - SELECT CAST(date AS TIMESTAMP)
    FROM table

VIEW

- Often there are specific combinations of tables and conditions that you find yourself using quite often for a project.
- Instead of having to perform the same query over and over again as a starting point, you can create a view to quickly see this query with a simple call



- A view is database object that is of a stored query.
- A view can be accessed as a virtual table in PostgreSQL
- Notice that a view does not store data physically, it simply stores the query
- You can also update and alter existing views.