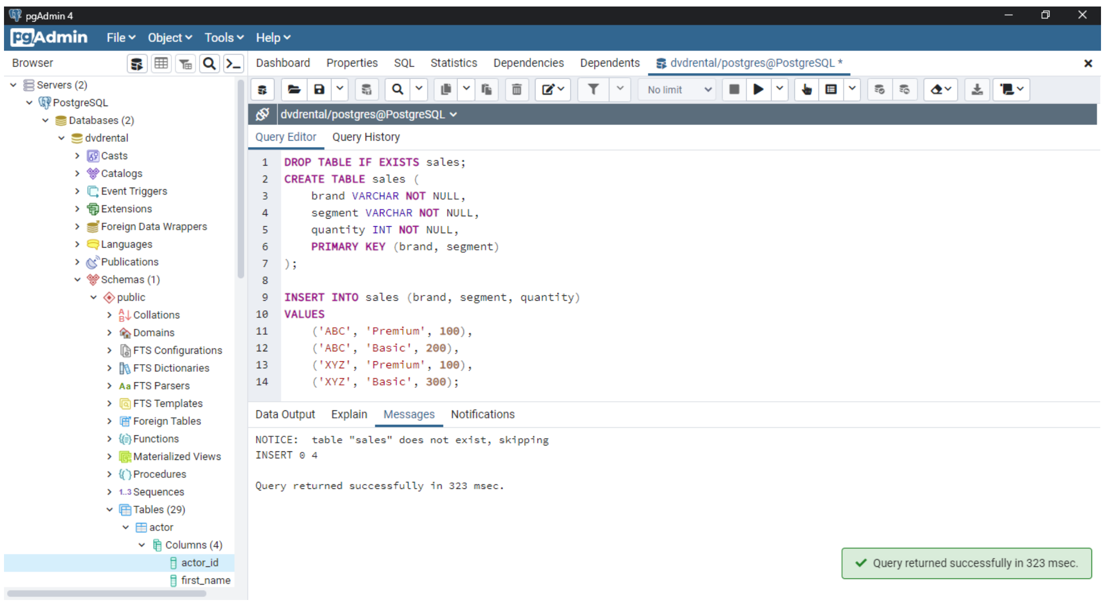
PostgreSQL GROUPING SETS

Setup a sample table

Let’s get started by creating a new table called sales for the demonstration.



```sql
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (
    brand VARCHAR NOT NULL,
    segment VARCHAR NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (brand, segment)
);

INSERT INTO sales (brand, segment, quantity)
VALUES
    ('ABC', 'Premium', 100),
    ('ABC', 'Basic', 200),
    ('XYZ', 'Premium', 100),
    ('XYZ', 'Basic', 300);
```

```
NOTICE:  table "sales" does not exist, skipping
INSERT 0 4

Query returned successfully in 323 msec.
```

The sales table stores the number of products sold by brand and segment.

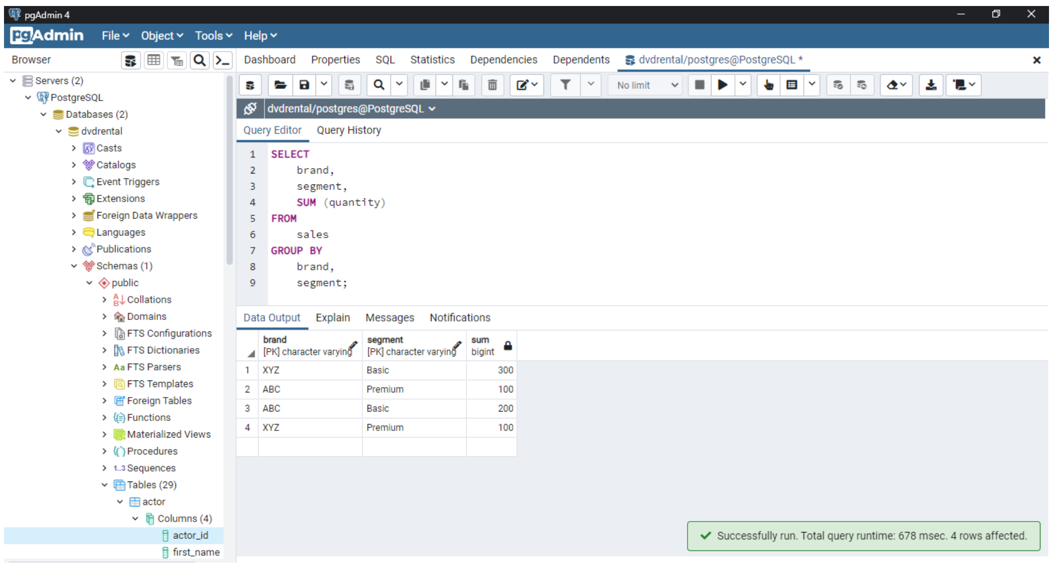
Introduction to PostgreSQL GROUPING SETS

A grouping set is a set of columns by which you group by using the GROUP BY clause.

A grouping set is denoted by a comma-separated list of columns placed inside parentheses:

```
(column1, column2, ...)
```

For example, the following query uses the GROUP BY clause to return the number of products sold by brand and segment. In other words, it defines a grouping set of the brand and segment which is denoted by (brand, segment)

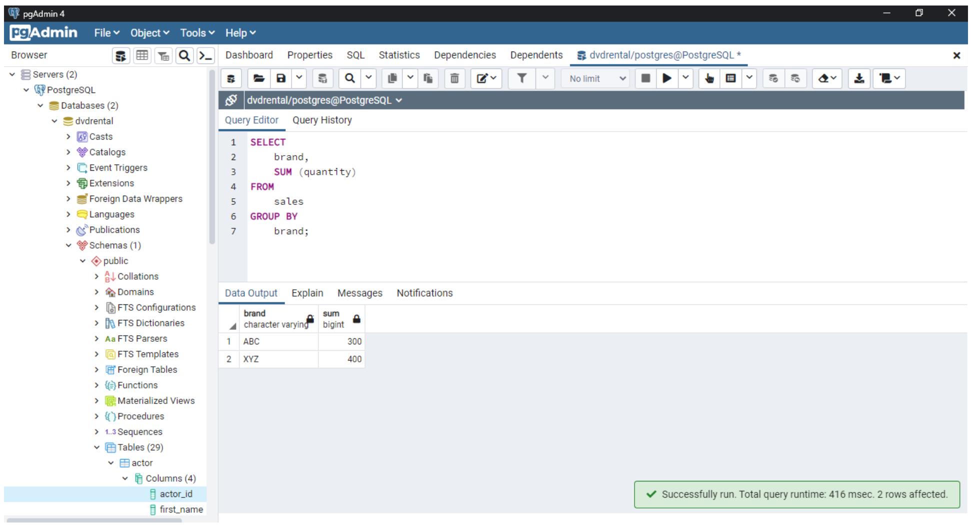


```sql
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand,
    segment;
```

	brand [PK] character varying	segment [PK] character varying	sum bigint
1	XYZ	Basic	300
2	ABC	Premium	100
3	ABC	Basic	200
4	XYZ	Premium	100

Successfully run. Total query runtime: 678 msec. 4 rows affected.

The following query finds the number of products sold by a brand. It defines a grouping set (brand):

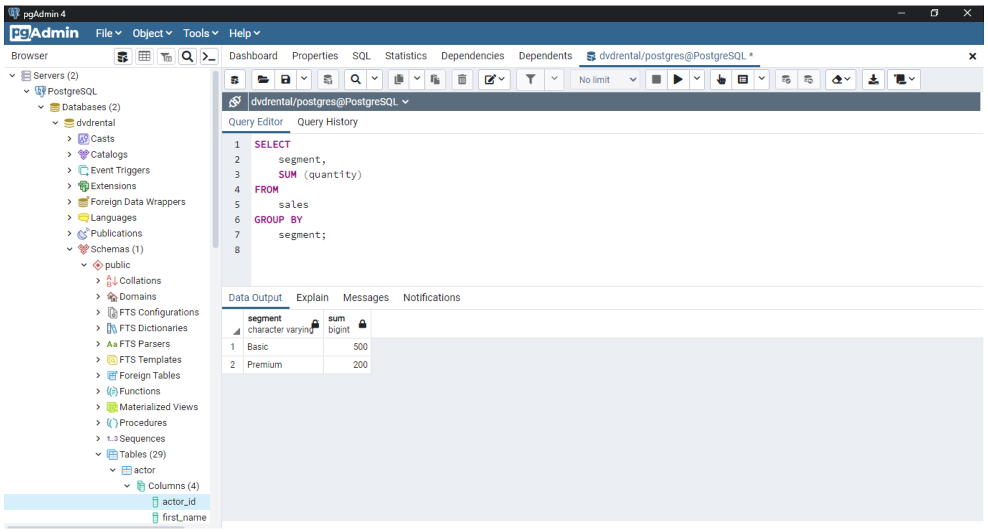


```sql
SELECT
    brand,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand;
```

```
  brand               sum
  character varying   bigint
1 ABC                 300
2 XYZ                 400
```

Successfully run. Total query runtime: 416 msec. 2 rows affected.

The following query finds the number of products sold by segment. It defines a grouping set (segment):



```sql
SELECT
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    segment;
```

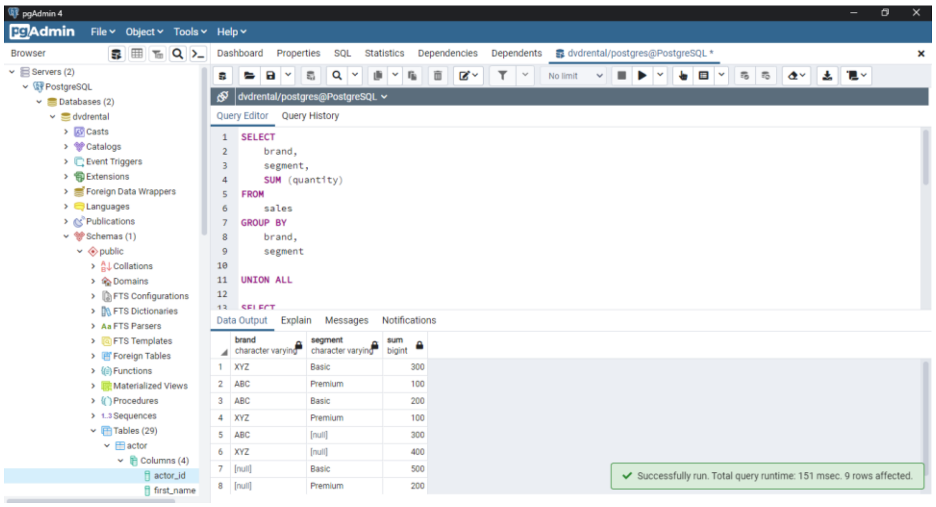
```
  segment             sum
  character varying   bigint
1 Basic               500
2 Premium             200
```

The following query finds the number of products sold for all brands and segments. It defines an empty grouping set which is denoted by ().

```sql
SELECT SUM (quantity) FROM sales;
```

Suppose that you want to all the grouping sets by using a single query. To achieve this, you may use the UNION ALL to combine all the queries above.

Because UNION ALL requires all result sets to have the same number of columns with compatible data types, you need to adjust the queries by adding NULL to the selection list of each as shown below:



```sql
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand,
    segment

UNION ALL

SELECT
```

```
  brand               segment             sum
  character varying   character varying   bigint
1 XYZ                 Basic               300
2 ABC                 Premium             100
3 ABC                 Basic               200
4 XYZ                 Premium             100
5 ABC                 [null]              300
6 XYZ                 [null]              400
7 [null]              Basic               500
8 [null]              Premium             200
```

Successfully run. Total query runtime: 151 msec. 9 rows affected.

This query generated a single result set with the aggregates for all grouping sets.

Even though the above query works as you expected, it has two main problems.

First, it is quite lengthy.

Second, it has a performance issue because PostgreSQL has to scan the sales table separately for each query.

To make it more efficient, PostgreSQL provides the GROUPING SETS clause which is the sub clause of the GROUP BY clause.

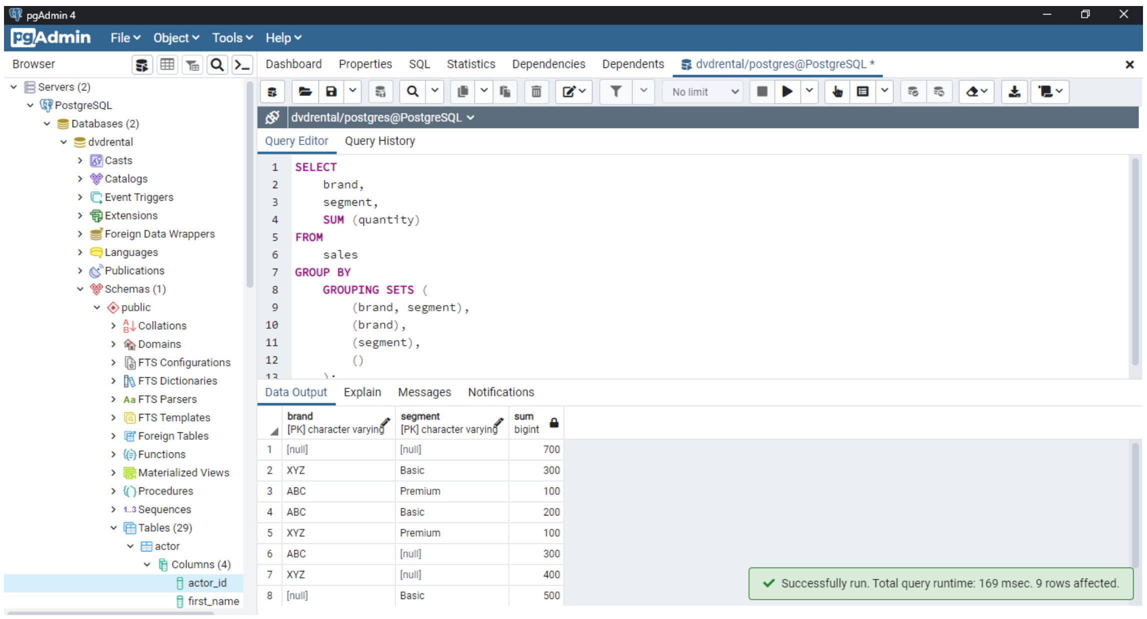
The GROUPING SETS allows you to define multiple grouping sets in the same query.

The general syntax of the GROUPING SETS is as follows:

```sql
SELECT
    c1,
    c2,
    aggregate_function(c3)
FROM
    table_name
GROUP BY
    GROUPING SETS (
        (c1, c2),
        (c1),
        (c2),
        ()
);
```

In this syntax, we have four grouping sets (c1, c2), (c1), (c2), and ().

To apply this syntax to the above example, you can use GROUPING SETS clause instead of the UNION ALL clause like this:



```sql
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    GROUPING SETS (
        (brand, segment),
        (brand),
        (segment),
        ()
    );
```

```
   brand                  segment                 sum
   [PK] character varying [PK] character varying  bigint
1  [null]                 [null]                  700
2  XYZ                    Basic                   300
3  ABC                    Premium                 100
4  ABC                    Basic                   200
5  XYZ                    Premium                 100
6  ABC                    [null]                  300
7  XYZ                    [null]                  400
8  [null]                 Basic                   500
```

```
Successfully run. Total query runtime: 169 msec. 9 rows affected.
```

This query is much shorter and more readable. In addition, PostgreSQL will optimize the number of times it scans the sales table and will not scan multiple times.

Grouping function

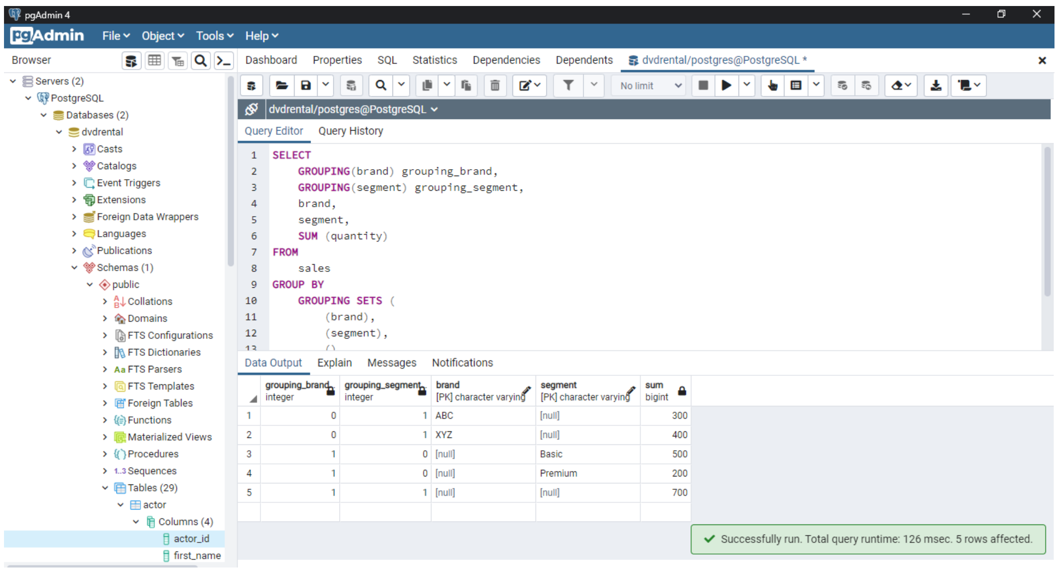
The GROUPING () function accepts an argument which can be a column name or an expression:

```
GROUPING( column_name | expression)
```

The column_name or expression must match with the one specified in the GROUP BY clause.

The GROUPING () function returns bit 0 if the argument is a member of the current grouping set and 1 otherwise.

See the following example:



```
SELECT
    GROUPING(brand) grouping_brand,
    GROUPING(segment) grouping_segment,
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    GROUPING SETS (
        (brand),
        (segment),
        ()
```

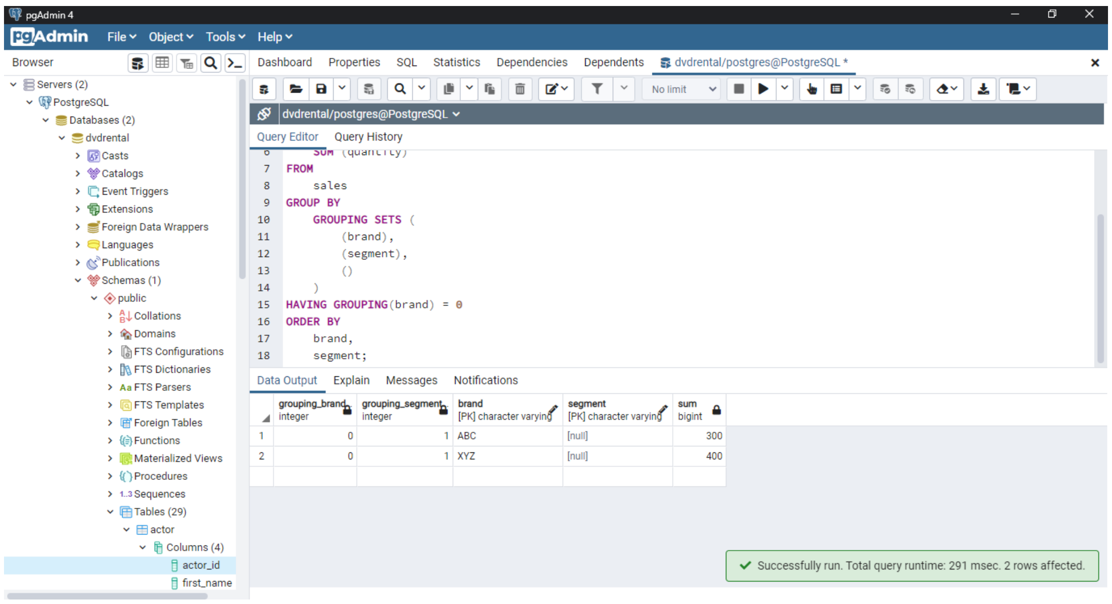
	grouping_brand integer	grouping_segment integer	brand [PK] character varying	segment [PK] character varying	sum bigint
1	0	1	ABC	[null]	300
2	0	1	XYZ	[null]	400
3	1	0	[null]	Basic	500
4	1	0	[null]	Premium	200
5	1	1	[null]	[null]	700

Successfully run. Total query runtime: 126 msec. 5 rows affected.

As shown in the screenshot, when the value in the grouping_brand is 0, the sum column shows the subtotal of the brand.

When the value in the grouping_segment is zero, the sum column shows the subtotal of the segment.

You can use the GROUPING () function in the HAVING clause to find the subtotal of each brand like this:



```
    SUM (quantity)
FROM
    sales
GROUP BY
    GROUPING SETS (
        (brand),
        (segment),
        ()
    )
HAVING GROUPING(brand) = 0
ORDER BY
    brand,
    segment;
```

	grouping_brand integer	grouping_segment integer	brand [PK] character varying	segment [PK] character varying	sum bigint
1	0	1	ABC	[null]	300
2	0	1	XYZ	[null]	400

Successfully run. Total query runtime: 291 msec. 2 rows affected.

PostgreSQL CUBE

Introduction to the PostgreSQL CUBE

PostgreSQL CUBE is a subclause of the GROUP BY clause. The CUBE allows you to generate multiple grouping sets.

A grouping set is a set of columns to which you want to group. For more information on the grouping sets, check it out the GROUPING SETS tutorial.

The following illustrates the syntax of the CUBE subclause:

```
SELECT
    c1,
    c2,
    c3,
    aggregate (c4)
FROM
    table_name
GROUP BY
    CUBE (c1, c2, c3);
```

In this syntax:

- First, specify the CUBE subclause in the the GROUP BY clause of the SELECT statement.
- Second, in the select list, specify the columns (dimensions or dimension columns) which you want to analyze and aggregation function expressions.
- Third, in the GROUP BY clause, specify the dimension columns within the parentheses of the CUBE subclause.

The query generates all possible grouping sets based on the dimension columns specified in CUBE. The CUBE subclause is a short way to define multiple grouping sets so the following are equivalent:

```
CUBE(c1,c2,c3)

GROUPING SETS (
    (c1,c2,c3),
    (c1,c2),
    (c1,c3),
    (c2,c3),
    (c1),
    (c2),
    (c3),
    ()
 )
```

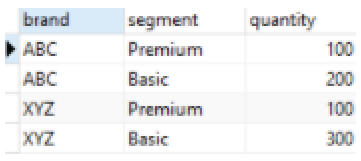
In general, if the number of columns specified in the CUBE is n, then you will have 2n combinations.

PostgreSQL allows you to perform a partial cube to reduce the number of aggregates calculated. The following shows the syntax:

```
SELECT
    c1,
    c2,
    c3,
    aggregate (c4)
FROM
    table_name
GROUP BY
    c1,
    CUBE (c1, c2);
```

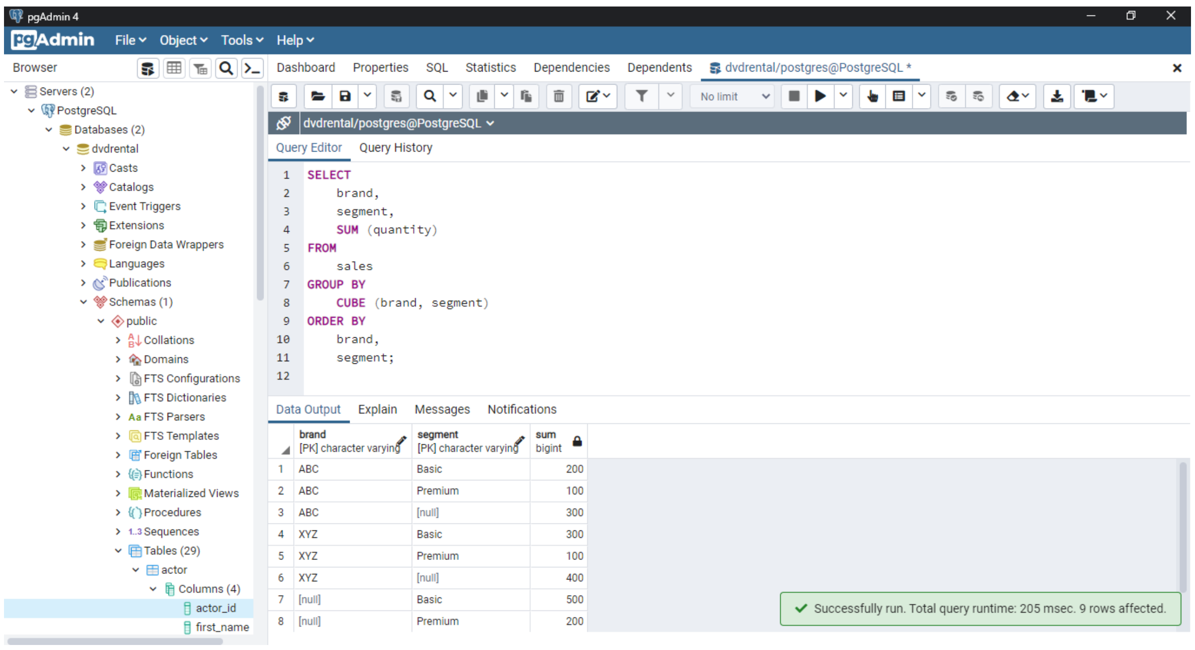
PostgreSQL CUBE examples

We will use the sales table created in the GROUPING SETS tutorial for the demonstration.



brand	segment	quantity
ABC	Premium	100
ABC	Basic	200
XYZ	Premium	100
XYZ	Basic	300

The following query uses the CUBE subclause to generate multiple grouping sets:


pgAdmin 4
File Object Tools Help
Browser
Dashboard Properties SQL Statistics Dependencies Dependents dvdrental/postgres@PostgreSQL *
No limit
dvdrental/postgres@PostgreSQL
Query Editor Query History
Data Output Explain Messages Notifications

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    CUBE (brand, segment)
ORDER BY
    brand,
    segment;
```

	brand [PK] character varying	segment [PK] character varying	sum bigint
1	ABC	Basic	200
2	ABC	Premium	100
3	ABC	[null]	300
4	XYZ	Basic	300
5	XYZ	Premium	100
6	XYZ	[null]	400
7	[null]	Basic	500
8	[null]	Premium	200

✓ Successfully run. Total query runtime: 205 msec. 9 rows affected.

PostgreSQL ROLLUP

Introduction to the PostgreSQL ROLLUP

The PostgreSQL ROLLUP is a subclause of the GROUP BY clause that offers a shorthand for defining multiple grouping sets. A grouping set is a set of columns by which you group. Check out the grouping sets tutorial for the detailed information.

Different from the CUBE subclause, ROLLUP does not generate all possible grouping sets based on the specified columns. It just makes a subset of those.

The ROLLUP assumes a hierarchy among the input columns and generates all grouping sets that make sense considering the hierarchy. This is the reason why ROLLUP is often used to generate the subtotals and the grand total for reports.

For example, the CUBE (c1, c2, c3) makes all eight possible grouping sets:

```
(c1, c2, c3)
(c1, c2)
(c2, c3)
(c1,c3)
(c1)
(c2)
(c3)
()
```

However, the ROLLUP (c1, c2, c3) generates only four grouping sets, assuming the hierarchy c1 > c2 > c3 as follows:

```
(c1, c2, c3)
(c1, c2)
(c1)
()
```

A common use of ROLLUP is to calculate the aggregations of data by year, month, and date, considering the hierarchy year > month > date

The following illustrates the syntax of the PostgreSQL ROLLUP:

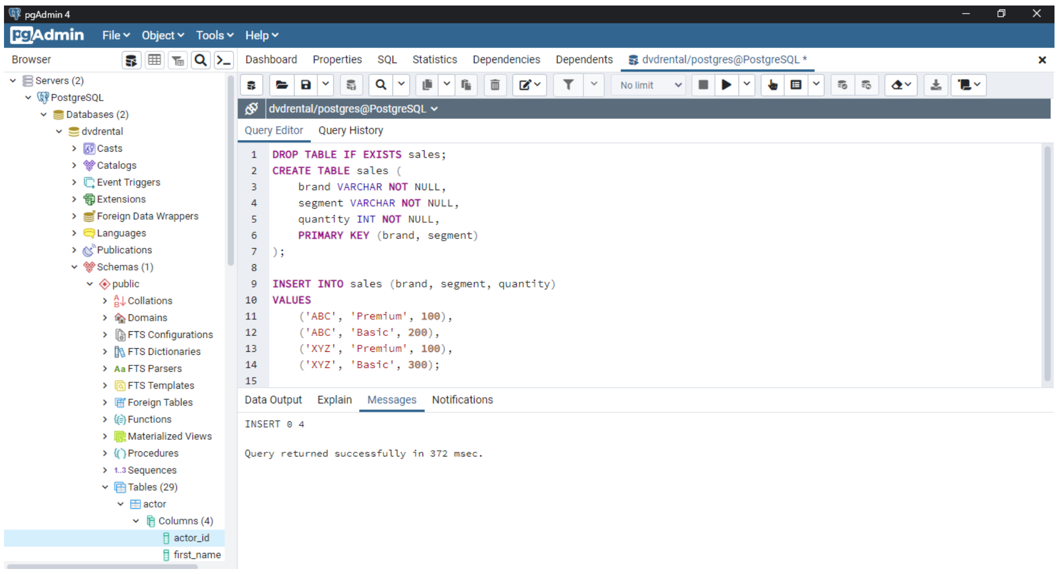
```
SELECT
    c1,
    c2,
    c3,
    aggregate(c4)
FROM
    table_name
GROUP BY
    ROLLUP (c1, c2, c3);
```

It is also possible to do a partial roll up to reduce the number of subtotals generated.

```
SELECT
    c1,
    c2,
    c3,
    aggregate(c4)
FROM
    table_name
GROUP BY
    c1,
    ROLLUP (c2, c3);
```

PostgreSQL ROLLUP examples

If you haven't create the sales table, you can use the following script:



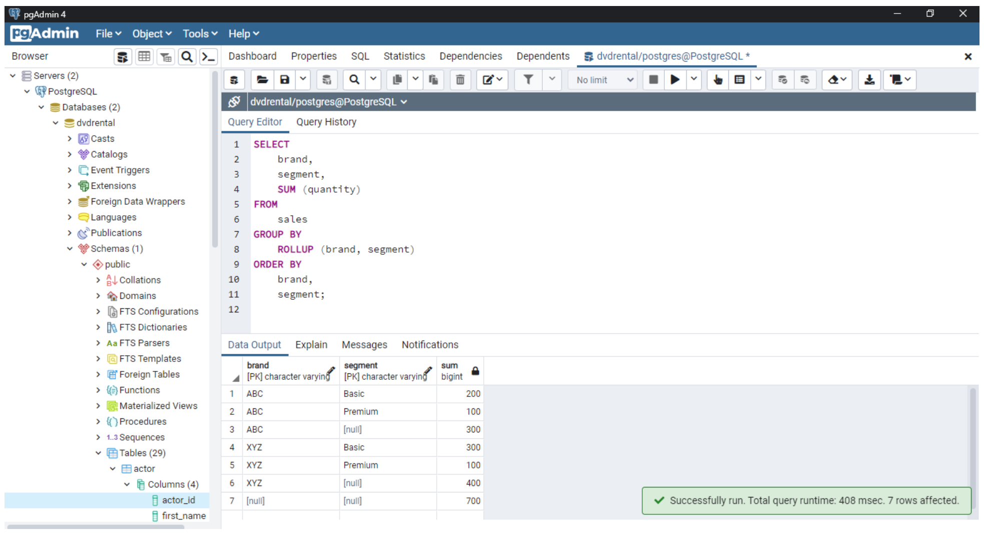
```
DROP TABLE IF EXISTS sales;
CREATE TABLE sales (
    brand VARCHAR NOT NULL,
    segment VARCHAR NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (brand, segment)
);

INSERT INTO sales (brand, segment, quantity)
VALUES
    ('ABC', 'Premium', 100),
    ('ABC', 'Basic', 200),
    ('XYZ', 'Premium', 100),
    ('XYZ', 'Basic', 300);
```

```
INSERT 0 4

Query returned successfully in 372 msec.
```

The following query uses the ROLLUP clause to find the number of products sold by brand (subtotal) and by all brands and segments (total).



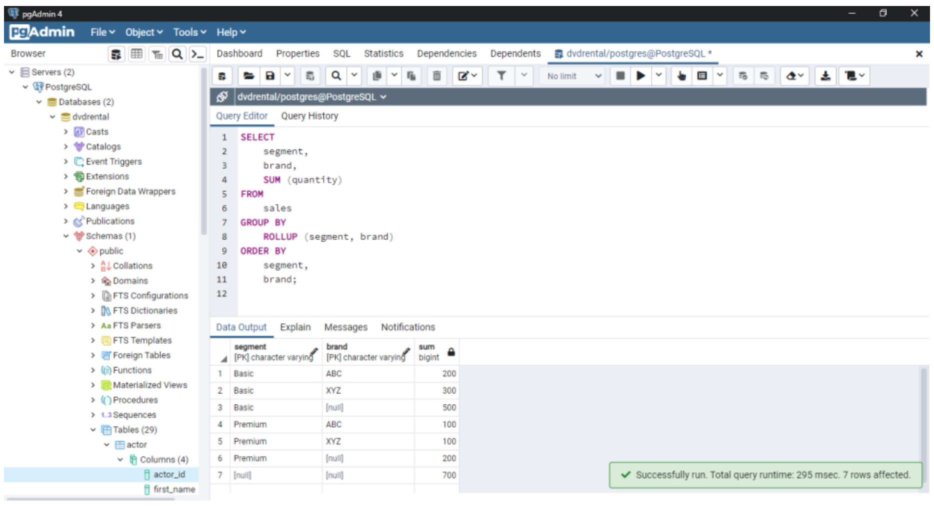
```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    ROLLUP (brand, segment)
ORDER BY
    brand,
    segment;
```

	brand [PK] character varying	segment [PK] character varying	sum bigint
1	ABC	Basic	200
2	ABC	Premium	100
3	ABC	[null]	300
4	XYZ	Basic	300
5	XYZ	Premium	100
6	XYZ	[null]	400
7	[null]	[null]	700

Successfully run. Total query runtime: 408 msec. 7 rows affected.

As you can see clearly from the output, the third row shows the sales of the ABC brand, the sixth row displays sales of the XYZ brand. The last row shows the grand total for all brands and segments. In this example, the hierarchy is brand > segment.

If you change the order of brand and segment, the result will be different as follows:



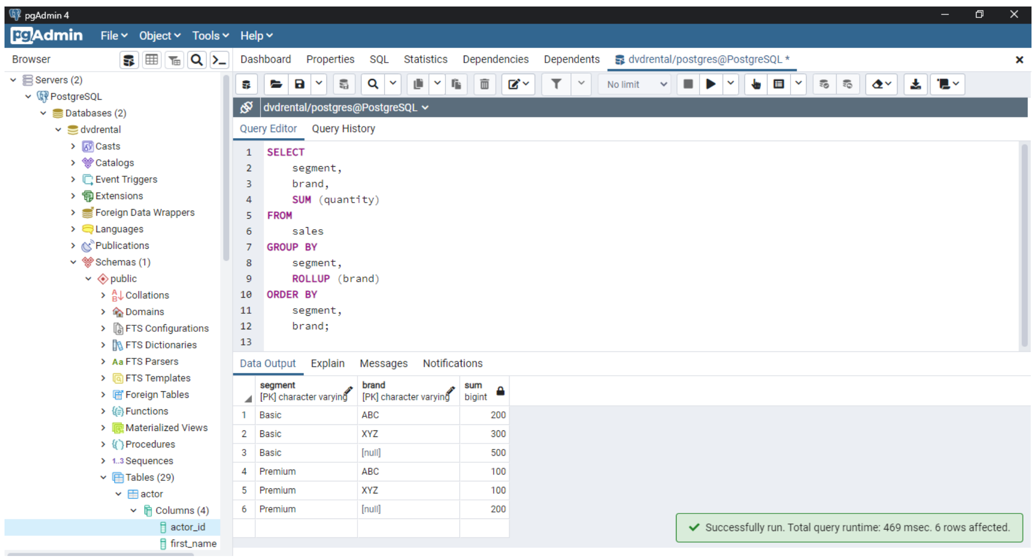
```sql
SELECT
    segment,
    brand,
    SUM (quantity)
FROM
    sales
GROUP BY
    ROLLUP (segment, brand)
ORDER BY
    segment,
    brand;
```

	segment [PK] character varying	brand [PK] character varying	sum bigint
1	Basic	ABC	200
2	Basic	XYZ	300
3	Basic	[null]	500
4	Premium	ABC	100
5	Premium	XYZ	100
6	Premium	[null]	200
7	[null]	[null]	700

Successfully run. Total query runtime: 295 msec. 7 rows affected.

In this case, the hierarchy is the segment > brand.

The following statement performs a partial roll-up:

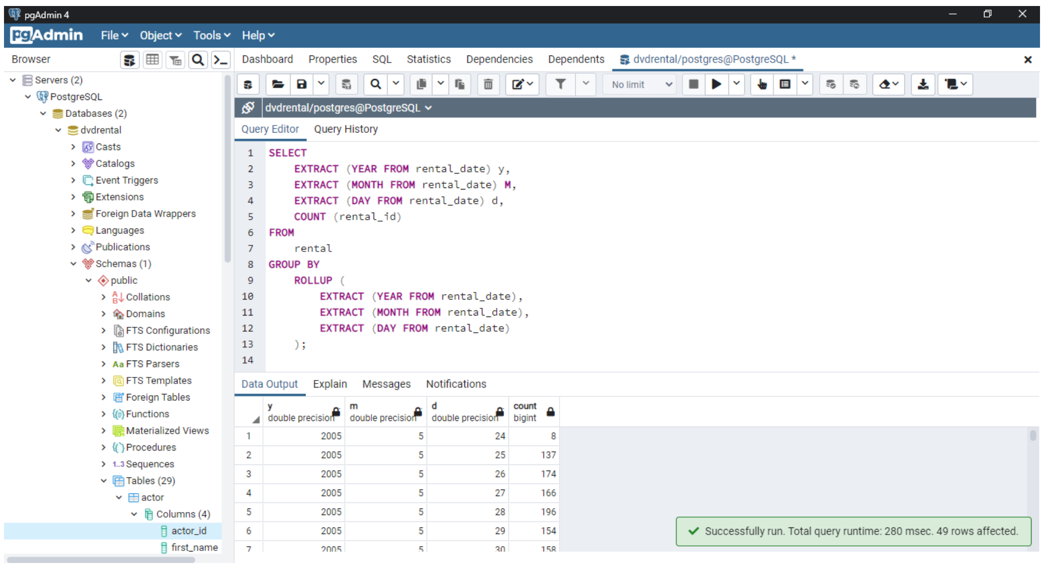


```sql
SELECT
    segment,
    brand,
    SUM (quantity)
FROM
    sales
GROUP BY
    segment,
    ROLLUP (brand)
ORDER BY
    segment,
    brand;
```

	segment [PK] character varying	brand [PK] character varying	sum bigint
1	Basic	ABC	200
2	Basic	XYZ	300
3	Basic	[null]	500
4	Premium	ABC	100
5	Premium	XYZ	100
6	Premium	[null]	200

Successfully run. Total query runtime: 469 msec. 6 rows affected.

The following statement finds the number of rental per day, month, and year by using the ROLLUP:



```sql
SELECT
    EXTRACT (YEAR FROM rental_date) y,
    EXTRACT (MONTH FROM rental_date) M,
    EXTRACT (DAY FROM rental_date) d,
    COUNT (rental_id)
FROM
    rental
GROUP BY
    ROLLUP (
        EXTRACT (YEAR FROM rental_date),
        EXTRACT (MONTH FROM rental_date),
        EXTRACT (DAY FROM rental_date)
    );
```

	y double precision	m double precision	d double precision	count bigint
1	2005	5	24	8
2	2005	5	25	137
3	2005	5	26	174
4	2005	5	27	166
5	2005	5	28	196
6	2005	5	29	154
7	2005	5	30	158

Successfully run. Total query runtime: 280 msec. 49 rows affected.