2021-07-13T18:00:00+02:00 | Andrew Bone

What Is the SQL GROUPING SETS Clause, and How Do You Use it?

GROUPING SETS are groups, or sets, of columns by which rows can be grouped together. Instead of writing multiple queries and combining the results with a UNION, you can simply use GROUPING SETS.

`GROUPING SETS` in SQL can be considered an extension of the `GROUP BY` clause. It allows you to define multiple grouping sets in the same query.

Let's look at its syntax and how it can be equivalent to a `GROUP BY` with multiple `UNION ALL` clauses.

SQL GROUPING SETS Syntax

The general syntax of the `GROUPING SETS` is as follows:

```
SELECT
    aggregate_function(column_1)
    column_2,
    column_3,
FROM
    table_name
GROUP BY
    GROUPING SETS (
        (column_2, column_3),
        (column_2),
        (column_3),
        ()
);
```

You can see how we are grouping by the different sets.

This syntax is equivalent to the following lengthier query that uses `GROUP BY` with `UNION ALL` to combine the results:

```
SELECT SUM(column_1), column_2, column_3
FROM table_name
GROUP BY
```

```
    column_2,
    column_3

UNION ALL

SELECT SUM(column_1), column_2, NULL
FROM table_name
GROUP BY column_2

UNION ALL

SELECT SUM(column_1), NULL, column_3
FROM table_name
GROUP BY column_3

UNION ALL

SELECT SUM(column_1), NULL, NULL
FROM table_name
```

If you use the `GROUP BY` like this, you need multiple `UNION ALL` clauses to combine the data from different sources. `UNION ALL` also requires all result sets to have the same number of columns with compatible data types, so you need to adjust the queries by adding a `NULL` value where required.

Even though the query will work as you expect, it has two main problems:

> It is lengthy and not very manageable.
>
> It can lead to a performance issue, because SQL has to scan the sales table each time.

The `GROUPING SETS` clause addresses these problems. But how does it affect the output compared to a traditional `GROUP BY` clause? Time to look at an example!

SQL GROUPING SETS Example

We need some sample data. Let's create a table called `payments` that contains all payments our company has received in January, February, and March for the past four years, 2018 to 2021. The exact store where the payment occurred is denoted by the `store_id` column.

To create this table, execute the following query:

```
CREATE TABLE payment (payment_amount decimal(8,2), payment_date date, store_id
int);


INSERT INTO payment
VALUES
(1200.99, '2018-01-18', 1),
(189.23, '2018-02-15', 1),
(33.43, '2018-03-03', 3),
(7382.10, '2019-01-11', 2),
(382.92, '2019-02-18', 1),
(322.34, '2019-03-29', 2),
(2929.14, '2020-01-03', 2),
(499.02, '2020-02-19', 3),
(994.11, '2020-03-14', 1),
(394.93, '2021-01-22', 2),
(3332.23, '2021-02-23', 3),
(9499.49, '2021-03-10', 3),
(3002.43, '2018-02-25', 2),
(100.99, '2019-03-07', 1),
(211.65, '2020-02-02', 1),
(500.73, '2021-01-06', 3);
```

You can view the data using this simple `SELECT` clause:

```
SELECT * FROM payment ORDER BY payment_date;
```

Executing this query yields the result:

payment_amount	payment_date	store_id
1200.99	2018-01-18	1
189.23	2018-02-15	1
3002.43	2018-02-25	2
33.43	2018-03-03	3
7382.10	2019-01-11	2
382.92	2019-02-18	1
100.99	2019-03-07	1
322.34	2019-03-29	2
2929.14	2020-01-03	2

payment_amount	payment_date	store_id
211.65	2020-02-02	1
499.02	2020-02-19	3
994.11	2020-03-14	1
500.73	2021-01-06	3
394.93	2021-01-22	2
3332.23	2021-02-23	3
9499.49	2021-03-10	3

You can see that there are multiple entries for some stores. Imagine we are preparing a report and we want to see one total for each store. The `SUM()` aggregate function can help us with this. We will also use the `GROUP BY` clause to group our results by year and store.

```
SELECT
  SUM(payment_amount),
  YEAR(payment_date) AS 'Payment Year',
  store_id AS 'Store'
FROM payment
GROUP BY YEAR(payment_date), store_id
ORDER BY YEAR(payment_date), store_id;
```

SUM(payment_amount)	Payment Year	Store
1390.22	2018	1
3002.43	2018	2
33.43	2018	3
483.91	2019	1
7704.44	2019	2
1205.76	2020	1
2929.14	2020	2
499.02	2020	3
394.93	2021	2
13332.45	2021	3

The results are aggregated by each unique combination of year and store.

However, we can't see the total payments by year: the total payments for 2018, 2019, 2020, or 2021. We cannot see the totals by store either,

which would be a useful metric to have. Using `GROUPING SETS` allows us to view these totals.

```
SELECT
  SUM(payment_amount),
  YEAR(payment_date) AS 'Payment Year',
  store_id AS 'Store'
FROM payment_new
GROUP BY GROUPING SETS (YEAR(payment_date), store_id)
ORDER BY YEAR(payment_date), store_id;
```

SUM(payment_amount)	Payment Year	Store
3079.89	NULL	1
14030.94	NULL	2
13864.90	NULL	3
4426.08	2018	NULL
8188.35	2019	NULL
4633.92	2020	NULL
13727.38	2021	NULL

Wow, our results changed drastically! Now, we see just the grand totals for each store along with the grand totals for each year.

For the columns by which the rows are not grouped, you see the `NULL` values.

Remember that you can include multiple `GROUP BY` clauses in your `GROUPING SETS`. Applying this to our query yields the following:

```
SELECT
  SUM(payment_amount),
  YEAR(payment_date) AS 'Payment Year',
  store_id AS 'Store'
FROM payment_new
GROUP BY GROUPING SETS
(
(YEAR(payment_date), store_id),
(YEAR(payment_date)),
(store_id)
)
ORDER BY YEAR(payment_date), store_id;
```

SUM(payment_amount)	Payment Year	Store
3079.89	NULL	1
14030.94	NULL	2
13864.90	NULL	3

SUM(payment_amount)	Payment Year	Store
4426.08	2018	NULL
1390.22	2018	1
3002.43	2018	2
33.43	2018	3
8188.35	2019	NULL
483.91	2019	1
7704.44	2019	2
4633.92	2020	NULL
1205.76	2020	1
2929.14	2020	2
499.02	2020	3
13727.38	2021	NULL
394.93	2021	2
13332.45	2021	3

Before ending this tutorial, we should mention two other SQL `GROUP BY` extensions that could prove useful for your particular project or scenario: `ROLLUP` and `CUBE`. These topics are covered in great detail in this Advanced SQL learning track from LearnSQL.com, which features window functions, `GROUP BY` extensions, and recursive queries.

It's time to stop being an SQL beginner – take another step towards being an expert with our Advanced SQL track!

SQL ROLLUP Example

Similar to `GROUPING SETS`, you can use the `ROLLUP` option in a single query to generate multiple grouping sets.

`ROLLUP` assumes a hierarchy among the input columns. For example, if the input columns are:

```
GROUP BY ROLLUP(column_1,column_2)
```

the hierarchy for this is `column_1 > column_2`, and `ROLLUP` generates the following grouping sets:

```
(column_1, column_2)
(column_1)
()
```

`ROLLUP` generates all grouping sets that make sense in this hierarchy. It generates a subtotal row every time the value of `column_1` changes; this is the hierarchy we have provided. For this reason, we often use `ROLLUP` to generate subtotals and grand totals in reporting. The ordering of your columns in `ROLLUP` is very important.

Let's look at a query that uses `ROLLUP`:

```
SELECT
  SUM(payment_amount),
  YEAR(payment_date) AS 'Payment Year',
  store_id AS 'Store'
FROM payment
GROUP BY ROLLUP (YEAR(payment_date), store_id)
ORDER BY YEAR(payment_date), store_id
```

SUM(payment_amount)	Payment Year	Store
30975.73	NULL	NULL
4426.08	2018	NULL
1390.22	2018	1
3002.43	2018	2
33.43	2018	3
8188.35	2019	NULL
483.91	2019	1
7704.44	2019	2
4633.92	2020	NULL
1205.76	2020	1
2929.14	2020	2
499.02	2020	3
13727.38	2021	NULL
394.93	2021	2
13332.45	2021	3

The grand total is shown at the top of the result:

30975.73 NULL NULL

The rest of the result is structured as follows. First, the yearly total is shown:

4426.08 2018 NULL

This is followed by the totals by store by year:

1390.22	2018	1
3002.43	2018	2
33.43	2018	3

As you can see, `ROLLUP` generates a subtotal row every time the value of `Payment Year` changes, since this is the hierarchy we provided. This example shows how useful `ROLLUP` can be for reporting purposes.

SQL CUBE Example

Similar to `ROLLUP`, `CUBE` is an extension of the `GROUP BY` clause. It allows you to generate subtotals for all combinations of the grouping columns specified in the `GROUP BY` clause.

The `CUBE` is like combining `GROUPING SETS` and `ROLLUP`. It shows the detailed output of both.

```
SELECT
  SUM(payment_amount),
  YEAR(payment_date) AS 'Payment Year',
  store_id AS 'Store'
FROM payment
GROUP BY CUBE (YEAR(payment_date), store_id)
ORDER BY YEAR(payment_date), store_id
```

SUM(payment_amount)	Payment Year	Store
30975.73	NULL	NULL
3079.89	NULL	1
14030.94	NULL	2
13864.90	NULL	3
4426.08	2018	NULL
1390.22	2018	1
3002.43	2018	2

SUM(payment_amount)	Payment Year	Store
33.43	2018	3
8188.35	2019	NULL
483.91	2019	1
7704.44	2019	2
4633.92	2020	NULL
1205.76	2020	1
2929.14	2020	2
499.02	2020	3
13727.38	2021	NULL
394.93	2021	2
13332.45	2021	3

The main difference in this output from the `ROLLUP` example is that the grand total for each store is also shown here.

3079.89	NULL	1
14030.94	NULL	2
13864.90	NULL	3

Apart from these rows, all the rows in this result are the same as the result of the `ROLLUP`.

This concludes our comparison of `GROUPING SETS`, `ROLLUP`, and `CUBE`! You can find more examples in this article about grouping, rolling and cubing data.

Get to know the GROUP BY extensions in SQL.

Group Your Data Effectively With SQL GROUP BY Extensions

Gaining mastery of the SQL `GROUP BY` extensions will take practice. Options like `GROUPING SETS`, `ROLLUP`, and `CUBE` allow you to manipulate the results of your queries in different ways. Knowing how to use these extensions effectively reduces the need for manually formatting your data before passing it on to relevant stakeholders.

To expand your knowledge further in this area, consider this GROUP BY extensions course from LearnSQL.com that covers `GROUPING SETS`, `ROLLUP`, and `CUBE`.
