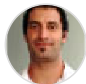
An Introduction to Using SQL Aggregate Functions with JOINs



Francisco Claria
Engineer @ Axones

Tags: AGGREGATE FUNCTIONS JOIN SQL BASICS

Previously, we've discussed the USE OF SQL AGGREGATE FUNCTIONS WITH THE GROUP BY STATEMENT. Regular readers of the our blog will also remember our recent TUTORIAL ABOUT JOINS. If you're a bit rusty on either subject, I encourage you to review them before continuing this article. That's because we will dig further into aggregate functions by pairing them with JOINs. This duo unleashes the full possibilities of SQL aggregate functions and allows us to perform computations on multiple tables in a single query.

What Do SQL Aggregate Functions Do?

Here's a quick overview of the most common **SQL aggregate functions**:

FUNCTION	PURPOSE	EXAMPLE
MIN	Returns the smallest value in a column.	`SELECT MIN(column) FROM table_name`
MAX	Returns the largest value in a column	`SELECT MAX(column) FROM table_name`

FUNCTION	PURPOSE	EXAMPLE
SUM	Calculates the sum of all numeric values in a column	`SELECT SUM(column) FROM table_name`
AVG	Returns the average value for a column	`SELECT AVG(column) FROM table_name`
COUNT(column)	Counts the number of non-null values in a column	`SELECT COUNT(column) FROM table_name`
COUNT(*)	Counts the total number of rows (including NULLs) in a column	`SELECT COUNT(*) FROM table_name`

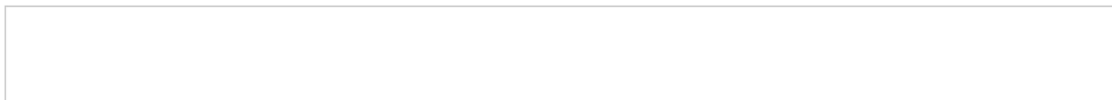
It's also important to remember that the `GROUP BY` statement, when used with aggregates, computes values that have been grouped by column. (For more info, see A BEGINNER'S GUIDE TO SQL AGGREGATE FUNCTIONS.) We can use GROUP BY with any of the above functions. For instance, we use the MIN() function in the *example*below:

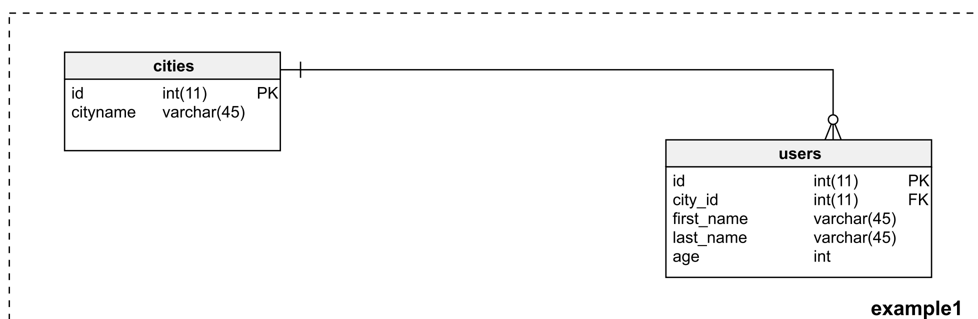
```
SELECT MIN(column_name)
FROM table_name
GROUP BY group_column
```

This would retrieve the minimum value found in `column_name` for each set of values in a group based on the `group_column` column. The same idea applies for `MAX`, `SUM`, `AVG`, and `COUNT` functions.

Parent-Child JOINs

Now let's dig into some common situations where you will use *group by JOINs* with aggregate functions. If you've read A Beginner's Guide to SQL Aggregate Functions, the following diagram will already be familiar:





EDIT MODEL IN YOUR BROWSER

If you have used this model before (e.g. doing the examples from the previous article) please be sure to clear any existing records from your table. You can do this by executing the following commands:

```
TRUNCATE cities;
TRUNCATE users;
```

Let's enter some fresh data into the tables:

```
INSERT INTO `cities` VALUES
    (1,'Miami'),
    (2,'Orlando'),
    (3,'Las Vegas'),
    (4,'Coyote Springs');
INSERT INTO `users` VALUES
    (1,1,'John','Doe',22),
    (2,1,'Albert','Thomson',15),
    (3,2,'Robert','Ford',65),
    (4,3,'Samantha','Simpson',9),
    (5,2,'Carlos','Bennet',42),
    (6,2,'Mirtha','Lebrand',81),
    (7,3,'Alex','Gomez',31);
```

So we have a table called `users` and another table called `cities`. These two tables have something in common: a numerical **city id value**. This value is stored in

the `id`column in the `cities` table and in the `city_id` column in the **`users`** table. The `city_id` column holds a reference (a.k.a. a foreign key) that connects a user record to a city. These matching records allow us to `JOIN` both tables together.

In other words, we know a user's city when we grab the record from the **`cities`** table that has an `id` value equal to the value in `users.city_id` . In the following query, we can see this in action:

```
SELECT  cities.*, users.*
FROM cities
JOIN users
  ON cities.id = users.city_id;
```

cities		users				
cityname	id	**city_id**	id	first_name	last_name	age
Miami	1	1	1	John	Doe	22
Miami	1	1	2	Albert	Thomson	15
Orlando	2	2	3	Robert	Ford	65
Las Vegas	3	3	4	Samantha	Simpson	9
Orlando	2	2	5	Carlos	Bennet	42
Orlando	2	2	6	Mirtha	Lebrand	81
Las Vegas	3	3	7	Alex	Gomez	31

Since the **`users`** table connects to one city via the `city_id` foreign key, we can say that a user belongs to a city and thus the city has many users. This is a parent-child relationship (cities-users); the **`users`** table shares a link to the **`cities`** table.

With this relationship in mind, let’s move on and see how we can compute some interesting summarized data that links both tables together.

Stop confusing INNER and OUTER JOINs with our interactive SQL JOINs course!

Aggregate + GROUP BY + JOIN

Now let's start addressing some practical situations where we will be `GROUP`ing values from `JOIN`ed tables.

MIN + GROUP BY + JOIN

Computing values based on child records that are grouped by a parent column is pretty common. Let's build a query that will retrieve the lowest `users.age` (child record) for each `cityname` (parent record):

```
SELECT cities.cityname, MIN(users.age)
FROM cities
JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

This will return:

cityname	MIN(users.age)
Las Vegas	**9**
Miami	**15**
Orlando	**42**

There's something very important to point out about the way JOIN works. It will be more obvious if we look at all cities:

```
SELECT cities.cityname
FROM cities
```

cityname
Coyote Springs
Las Vegas
Miami
Orlando

As you can see, "Coyote Springs" was not listed before because it has no users. If you wanted to get that city listed in the summarized results, you should use a `LEFT JOIN` instead:

```
SELECT cities.cityname, MIN(users.age)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

This will return:

cityname	MIN(users.age)
Coyote Springs	**null**
Las Vegas	9
Miami	15
Orlando	42

Whether this makes sense or not will depend on your use case, but it's important that you keep this situation in mind when joining tables.

MAX + GROUP BY + JOINS

We can find the *greatest* age for each city using the `MAX()` function:

```
SELECT cities.cityname, MAX(users.age)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

The query above will retrieve:

cityname	MAX(users.age)
Coyote Springs	null
Las Vegas	31
Miami	22
Orlando	81

Note that I have used `LEFT JOIN`. I want a list of all the cities, not only those with associated user records.

SUM + GROUP BY + JOIN

Let's now see how to total ages for each city. We can use the `SUM()` function to do this:

```
SELECT cities.cityname, SUM(users.age)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

Which returns:

cityname	SUM(users.age)

cityname	SUM(users.age)
Coyote Springs	null
Las Vegas	40
Miami	37
Orlando	188

COUNT + GROUP BY + JOIN

Suppose we want to see the number of users in each city. We would use the `COUNT()` function, like this:

```
SELECT cities.cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

Which returns:

cityname	COUNT(users.id)
Coyote Springs	0
Las Vegas	2
Miami	2
Orlando	3

AVERAGE + GROUP BY + JOIN

Using the number of users in each city (`COUNT`) and the `SUM` of each city's combined user ages, we can compute the average age for each city. We simply divide the summed age by the number of users for each city:

```
SELECT
    cities.cityname,
    SUM(users.age) AS sum,
    COUNT(users.id) AS count,
    SUM(users.age) / COUNT(users.id) AS average
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

Returning:

cityname	sum	count	average
Coyote Springs	null	0	null
Las Vegas	40	2	20.0000
Miami	37	2	18.5000
Orlando	188	3	62.6667

Notice how the sum and calculated average results in a NULL value for Coyote Springs. This is because Coyote Springs has no users and therefore the summarized column cannot compute a numerical value.

AVG + GROUP BY + JOINS

The previous example used a calculation we entered to find an average age for each city. We could have used the `AVG()` function instead, as shown below:

```
SELECT cities.cityname, AVG(users.age)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

This results in the same values as the previous *example*:

cityname	AVG(users.age)
Coyote Springs	**null**
Las Vegas	**20.0000**
Miami	**18.5000**
Orlando	**62.6667**

Filtering Results

Sometimes you will need to FILTER ROWS BASED ON CERTAIN CONDITIONS. In this type of query, there are three stages where you can do that: `WHERE`, `HAVING`, and `JOIN`.

Depending on the situation, each of these options can have a different outcome. It's important to understand which to use when you want a specific result. Let's look at some examples to illustrate this.

Using the JOIN Predicate

Let's get the number of users under 30 in each city. We will use `LEFT JOIN` to retrieve cities without any user records:

```
SELECT cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
  AND users.age < 30
GROUP BY cities.cityname
ORDER BY cities.cityname;
```

The condition to include only *users with ages lower than 30* is set in the `JOIN` predicate. This returns the following output:

cityname	COUNT(users.id)

cityname	COUNT(users.id)
Coyote Springs	**0**
Las Vegas	**1**
Miami	**2**
Orlando	**0**

All cities are listed, and only those users with ages within range return a non-zero number. Cities without any users matching our criteria return a zero.

What would have happened if we put the same filtering condition in the `WHERE` clause?

> Improve your SQL JOIN skills with our special interactive course, SQL JOINs!

Using WHERE Conditions

If place the same conditions in the `WHERE`, it would look like this:

```
SELECT cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
WHERE users.age < 30
GROUP BY cities.cityname
ORDER BY cities.cityname;
```

This will result in:

cityname	COUNT(users.id)
Las Vegas	**1**
Miami	**2**

This is **not** what I expected; I wanted to get ALL cities and a *count* of their respective users aged less than 30. Even if a city had no users, it should have been listed with a zero count, as returned by the `JOIN` predicate example.

The reason this didn't return those records is because **`WHERE` conditions are applied after the `JOIN`**. Since the condition `users.age < 30` removes all "Coyote Springs" and "Orlando" records, the summarized calculation can't include these values. Only "Las Vegas" and "Miami" meet the `WHERE` conditions, so only "Las Vegas" and "Miami" are returned.

In contrast, when the condition is applied in the `JOIN` predicate, *user records* with no matching age are removed **before** the two tables are joined. Then all the cities are matched by user columns, as you would expect when using a *`LEFT JOIN`*. This means that all cities will be part of the results; only user records that did not meet the `users.age < 30` condition are filtered out. In this case, the JOIN predicate returns the desired outcome.

Using HAVING Conditions

We mentioned this is the first article, but we'll repeat it here: using the `WHERE` clause to filter summarized columns doesn't work. Look at the example below.

```
SELECT cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
WHERE COUNT(users.id) > 2
GROUP BY cities.cityname
ORDER BY cities.cityname;
```

This causes the database to issue a complaint like this one from MySQL:

```
Error Code: 1111. Invalid use of group function
```

Instead, use the `HAVING` clause:

```
SELECT cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
HAVING COUNT(users.id) > 2
ORDER BY cities.cityname;
```

This returns the intended records (only cities with more than two users):

cityname	COUNT(users.id)
Orlando	**3**

Dealing with NULLs

Besides the edge cases already presented, it is important to consider something that isn't so obvious. Let's go back to the `COUNT()` example:

```
SELECT cities.cityname, COUNT(users.id)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

This returns:

cityname	COUNT(users.id)
Coyote Springs	0
Las Vegas	2
Miami	2
Orlando	3

If I had used `COUNT(*)` instead of `COUNT(users.id)`, the **total row count** would have been generated. This would have given us an unintended value � in this case, a false "1" for "Coyote Springs". This result is due to the nature of the `LEFT JOIN`. Here is an *example*:

```
SELECT cities.cityname, COUNT(*)
FROM cities
LEFT JOIN users
  ON cities.id = users.city_id
GROUP BY cities.cityname
```

This would return:

cityname	COUNT(users.id)
Coyote Springs	1
Las Vegas	2
Miami	2
Orlando	3

So `COUNT(*)` is counting a "1" for Coyote Springs because the `LEFT JOIN` is returning a row with NULL values. Remember that in `COUNT(*)`, a row with NULLs still counts.

For the same reason, `COUNT(users.id)` returns the expected count of "0"; the `users.id` column value is null for Coyote Springs.

In other words, always use `Count(column)` with this type of query.

A Final Tip on Working with SQL Aggregate Functions

Finally, I'd like to add that working with *sql aggregate functions* � especially when using `JOIN`s � requires you understand SQL and the data you are working with. Try the queries in a smaller subset of your data first to confirm that all calculations are working as expected. If, possible, check some outputs against a reference value to validate your queries' outcomes.

Keep in mind that using conditions in the `JOIN` predicate (after the `ON`) is not the same as filtering in the `WHERE` (or using `HAVING`). These can create subtle (or not so subtle) differences in your summarized data, which could result in hard-to-spot errors. Pay special attention to your filtering choices.

As always, thanks for reading and please feel free to share your own experiences in the comments section.