Week 6 cliff notes

Last week we covered ‘simple’ functions, or functions that operate on each and every row encountered by the query. The grammar we have to date is:

**SELECT** [either one or more columns and/or functions, or another sql]

**FROM** [a rowsource, like a table or a view]

**WHERE** [a list of predicates that evaluate to TRUE or FALSE for each row in the rowsource]

**ORDER BY** [a list of output columns that you would like to sort by]

This week we will cover group functions, or functions that act on a range of rows and output (potentially) fewer rows. We will also introduce the newest member of the grammar, GROUP BY. Again, here is the link for the SQL Server documentation on functions - today we are talking about what Microsoft calls “Aggregate Functions”:

<https://msdn.microsoft.com/en-us/library/ms174318.aspx>

Starting with something really simple, let’s get a count of all the rows in table using our [SQL Fiddle](http://sqlfiddle.com/#!9/60125).

select count(\*) From movies;

|  |
| --- |
| **count(\*)** |
| 26 |

From previous weeks, we know we typically see several columns and several rows when we use ‘\*’ to select from a table. The COUNT function merely counts the number of rows and gives us a single-row result. We could have used just about any column inside the COUNT() function, for example,

select count(substr(title, 1)) From movies;

Which gives the exact same result, so really it is a waste of sql. Some people like to use ‘select count(1) from table;’ with the hope that an index will make the count faster - some database systems implement that shortcut differently (and if I had a class on database performance, we would talk about that). For where we are at in this course, either way works just fine. I am used to count(\*) so that is what I will use.

Then you have a few group functions that should be pretty familiar to most people, MAX and MIN. MAX finds the maximum value, while MIN finds the minimum value.

select max(year\_released) From movies;

|  |
| --- |
| **max(year\_released)** |
| 2001 |

select min(year\_released) From movies;

|  |
| --- |
| **min(year\_released)** |
| 1925 |

The same thing can be done for non-numeric data (ie, columns with letters), sorted alphabetically, like the title:

select max(title) From movies;

|  |
| --- |
| **max(title)** |
| Ying hung boon sik |

select min(title) From movies;

|  |
| --- |
| **min(title)** |
| Annie Hall |

And just to make your brain hurt a little, let’s put all four in one sql statement:

select max(year\_released), min(year\_released), max(title), min(title) From movies;

|  |  |  |  |
| --- | --- | --- | --- |
| **max(year\_released)** | **min(year\_released)** | **max(title)** | **min(title)** |
| 2001 | 1925 | Ying hung boon sik | Annie Hall |

These group functions are taking all 26 rows and comparing them, outputting a single value. So why does putting all four group functions in one query return only one row, but with four different columns? This is allowed because we use the same number of input rows and output rows; in each case, we take in all 26 rows, and for each function (albeit different columns), we only output one value. To think about it algorithmically, try this pseudo-code out (in your head or on paper, or in a spreadsheet):

for each row (

if the year\_released in this row is larger than my current maximum year\_released, set the maximum to the year\_released of this row

if the year\_released in this row is smaller than my current minimum year\_released, set the minimum to the year\_released of this row

if the title in this row is larger than my current maximum title, set the maximum to the title of this row

if the title in this row is smaller than my current minimum title, set the minimum to the title of this row

)

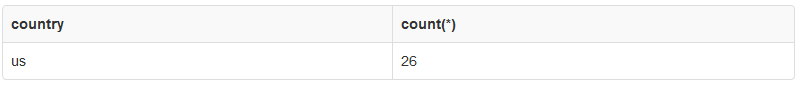
Up until now, we have used the entire table (or rowset) as a single group. But now we will put the data into smaller groups and do stuff to them; this is the essence of “aggregating” - putting things in buckets.

Write a sql query that tells us how many movies are from each country.

For this kind of question, it is important to find keywords. In this exercise, the keywords are “how many” and “each”. This tells us that we want to count movies per country. We could write the following:

select country, count(\*) from movies;

Some database systems do not allow this query. In mysql, we get the following result:



But that’s not right - just looking at the data we see other countries like ‘ru’, ‘jp’ and ‘nz’. So what is wrong? Somehow we have to tell the database to put the countries into groups. We have done something similar in the past with DISTINCT: select distinct country from movies; In fact, we can even count those:

select count(distinct country) from movies;

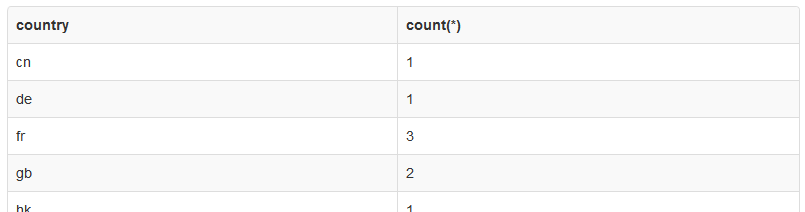


[Note how I put “distinct country” inside the COUNT function]

So we know that in all 26 rows, we have 12 unique countries, which means we should end up with 12 different buckets for this exercise.

To tell the database to group the country, we have to use the GROUP BY clause:

select country, count(\*) from movies group by country;



Untitled.png

In essence, GROUP BY applies the function (in this case, COUNT) to each bucket. What is *in* each bucket? Just a collection of countries (or rather, their abbreviations). So using functions like MAX and MIN don’t really make much sense, because the MAX of the ‘us’ bucket would be ‘us’, and the MIN would be exactly the same thing. We could find the maximum YEAR\_RELEASED for each country:

select country, max(year\_released) from movies group by country;



Note that in our grammar, GROUP BY must come after any WHERE clauses and before any ORDER BY clauses:

**SELECT** [either one or more columns and/or functions, or another sql]

**FROM** [a rowsource, like a table or a view]

**WHERE** [a list of predicates that evaluate to TRUE or FALSE for each row in the rowsource]

**GROUP BY** [a list of columns around which to form buckets or groups]

**ORDER BY** [a list of output columns that you would like to sort by]

Like before, let’s put MAX and MIN in the same query - we are still using the same buckets (distinct countries), so we don’t have to worry about altering the GROUP BY clause, only what we select. Heck, let’s throw the COUNT back in there as well:

select country, count(\*), max(year\_released), min(year\_released) from movies group by country;



So if a bucket, like ‘cn’, only has a count of 1 (meaning, one row), the MAX and the MIN will be the same. Makes sense, right? But other buckets, like ‘fr’, that have more than one row, it makes sense that the MAX is different than the MIN. Remember, the important part here is the GROUP BY is creating buckets, and then we are performing work (aka, functions) on each bucket. I am going to go over more complex examples in just a moment, but first let us take a peek at the SUM function.

By this time, you should have a basic idea of how group functions, and the GROUP BY clause works. We don’t have any columns that make sense to SUM, so I am going to make a new table called SALARIES - here is the new [SQL Fiddle](http://sqlfiddle.com/#!9/7f7cc). I now ask you to find the sum of all salaries for each ROLE. Since I used the keyword ***EACH***, automatically alarms should be going off in your head and you should starting thinking about GROUPS or BUCKETS.

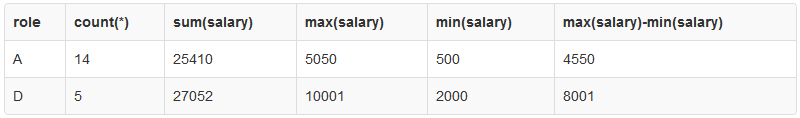
In SQL, we would write:

select role, sum(salary) from salaries group by role;



As you can see from the data, we only have two types of “roles”, the ‘A’s (actors) and ‘D’s (directors); it looks like the Directors win out. That’s all good and dandy, but since we are talking about money, people always have more questions. Who makes the most? Who makes the least? What are the ranges? Just like the MOVIES example, we could answer each question with an individual sql, or we can combine them altogether in one query:

select role, count(\*), sum(salary), max(salary), min(salary), max(salary)-min(salary) from salaries group by role;



Sorry if that gave you a headache. :) Let’s break it down a bit. For each bucket (ROLE), we asked several questions (aka, applied functions):

* The count or number of rows in each bucket: count(\*)
* The sum of all salaries in each bucket: sum(salary)
* The maximum salary in each bucket: max(salary)
* The minimum salary in each bucket: min(salary)
* The difference between the maximum and minimum salary in each bucket: max(salary)-min(salary)

Wicked, eh? From this kind of analytics (information or data that is used to analyze data), we can tell that there is a huge amount of disparity for Directors, and that the highest paid Actor gets more money than the lowest paid Director.

Take a little breather here. We have 2 more things to cover yet in this unit:

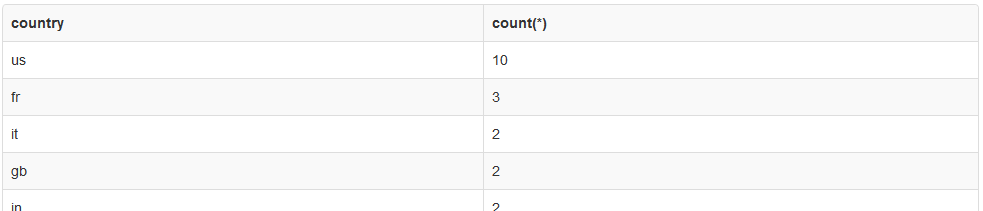
* Top N query with GROUPS
* using the HAVING clause



**TOP N queries**

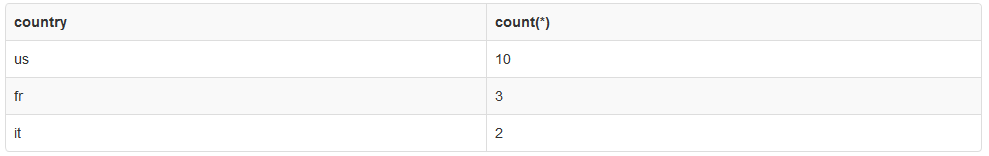
In a previous unit we talked about using the LIMIT clause to reduce the rows we see in the output. This is very common when you want to see the top 10, or the top 5 or what have you. So back to our MOVIES example (original [SQL Fiddle](http://sqlfiddle.com/#!9/60125)) when we grouped by COUNTRY - let’s find the top 3 countries that have the most movies. We already know how to count movies by country and order it, right? (if you don’t remember, ***STOP reading*** here and go look at previous units).

select country, count(\*) from movies group by country order by count(\*) desc;



To get the top 3, all we need to do is add LIMIT 3:

select country, count(\*) from movies group by country order by count(\*) desc limit 3;



You have might have noticed that there is a three-way tie for third place, so how does the database know to give you ‘it’ (Italy), instead of ‘gb’ (Great Britain) or ‘in’ (India)? Frankly, for all intents and purposes it is pretty random. In more advanced SQL, we would talk about ranking functions, which includes options to deal with tie-breakers.

**HAVING clause**

Top N is great if you just want an arbitrary number of rows returned, but what if you don’t know how many rows you will get back, but you definitely want a subset. For instance, how would you write a query to only return countries from the MOVIES table that have more than 1 movie? We can look at the data from the queries above and know that we really want to see 5 rows (until someone changes the data). So maybe we just add a WHERE clause?

select country, count(\*) from movies where count(\*) > 1 group by country;

Untitled.png

Different databases will report this error differently, but in general, this is INVALID syntax. In general, the WHERE clauses attempt to apply a predicate to ***each row***. Our query is asking the database to apply the predicate to an entire bucket. The syntax just does not allow that. So we have to have a different trick. Enter the HAVING clause.

New grammar:

**SELECT** [either one or more columns and/or functions, or another sql]

**FROM** [a rowsource, like a table or a view]

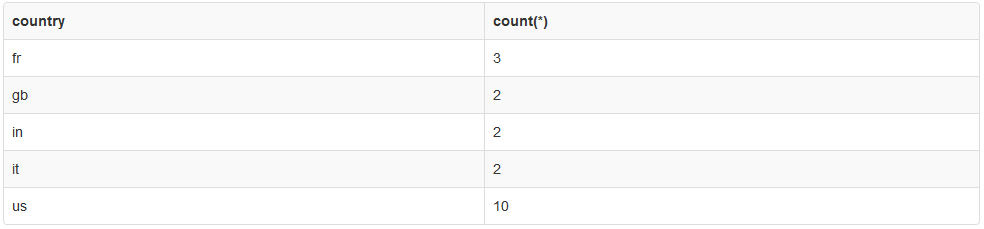
**WHERE** [a list of predicates that evaluate to TRUE or FALSE for each row in the rowsource]

**GROUP BY** [a list of columns around which to form buckets or groups]

**HAVING** [a list of predicates that apply to GROUP BY groups]

**ORDER BY** [a list of output columns that you would like to sort by]

select country, count(\*) from movies group by country having count(\*) > 1;



You can add an ORDER BY to sort the output if you like.

From here, it’s just a matter of figuring out what you really want. Say we want to count all movies in each country, and also find the min and max year\_released, but we only want to see countries that have more than one movie, the groups max year\_released must be from the past 41 years, and the country must be in the Eastern (aka, Oriental) hemisphere. Whew!

Start small, and build from there. We already know part of it:

select country, count(\*), max(year\_released), min(year\_released) from movies group by country;



Now limit it to countries that have more than one movie (which we also already did):

select country, count(\*), max(year\_released), min(year\_released) from movies group by country having count(\*) > 1;



And now to filter for the last 41 years (good thing year\_released is not a true DATE datatype):

select country, count(\*), max(year\_released), min(year\_released) from movies group by country having count(\*) > 1 and max(year\_released) > 1974;



And finally, all the ones that fit in the Eastern Hemisphere (curve ball):

select country, count(\*), max(year\_released), min(year\_released) from movies group by country having count(\*) > 1 and max(year\_released) > 1974 and country in ('fr','in');

-or-

select country, count(\*), max(year\_released), min(year\_released) from movies where country in ('fr','in') group by country having count(\*) > 1 and max(year\_released) > 1974;



Presto Bingo.

Next week we start the dreaded path of joining together multiple tables. You shall quake in your boots and quiver like leaves, bwahahahaha… err. ok.

Here is our grammar to date:

**SELECT** [either one or more columns and/or functions, or another sql]

**FROM** [a rowsource, like a table or a view]

**WHERE** [a list of predicates that evaluate to TRUE or FALSE for each row in the rowsource]

**GROUP BY** [a list of columns around which to form buckets or groups]

**HAVING** [a list of predicates that apply to GROUP BY groups]

**ORDER BY** [a list of output columns that you would like to sort by]