SQL

Steven Rashin

2024-04-24

Preliminaries

First we're going to create a few datasets using r. The majority of the tutorial doesn't use them because the combination of SQL and Quarto is a bit buggy but it's a useful exercise anyway.

```
library(odbc)
library(tidyverse)
library(DBI)
library(here)
library(RSQLite)

here::i_am("SQL-published-version.qmd")
```

```
sample_data <- tibble(
  id = 1:1000,
  x1 = rnorm(1000, 0, 1),
  x2 = rnorm(1000, 10, 15),
  y = 3 * x1 + 4 * x2 + rnorm(1000, 0, 10),
  g = rbinom(1000, size = 1, prob = 0.3)
)
sample_data2 <- tibble(
  id = 1:1000,
  x3 = rnorm(1000, 15, 30),
  x4 = rnorm(1000, 20, 5),
  y2 = 10 * x3 + 40 * x4 + rnorm(1000, 0, 40),
  g2 = rbinom(1000, size = 1, prob = 0.3)
)

mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbWriteTable(conn = mydb, name = "sample", value = sample_data, overwrite = T)
dbWriteTable(conn = mydb, name = "other_sample", value = sample_data2, overwrite = T)

#### Try to load in postgresql - doesn't currently work

# https://caltechlibrary.github.io/data-carpentry-R-ecology-lesson/05-r-and-databases.html
# https://jtr13.github.io/cc21fall2/how-to-integrate-r-with-postgresql.html
# https://solutions.posit.co/connections/db/databases/postgresql/
#https://medium.com/geekculture/a-simple-guide-on-connecting-rstudio-to-a-postgresql-datab
```

```r
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbListTables(mydb) # returns a list of tables in your database
```

```
[1] "other_sample" "sample"
```

Basic Syntax

The basic syntax is you `SELECT` variables `FROM` a database.

$$\underbrace{\text{SELECT}}_{\text{Select vars}} \underbrace{*}_{\text{* is all variables}}$$

$$\underbrace{\text{FROM}}_{\text{from where}} \underbrace{\text{db\_name}}_{\text{name}}$$

See e.g.,

```r
SELECT * ----- select all
FROM sample ----- select from this dataframe
LIMIT 10 ----- for demo purposes, limit output to 10 rows
```

You can run these commands in r using the following syntax. Since you can't run the sql commands in quarto without compiling, I've used this method to check that my sql commands actually work.

```r
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbGetQuery(mydb,'
  select *
  from "sample"
  limit 10
')
```

```
  id         x1         x2          y g
1  1 -0.6183267  -2.237845   4.429335 1
2  2  0.6966853  -2.208384  -3.683680 0
3  3 -0.9733263  21.117396  73.705869 1
4  4 -0.4735464  -8.064834 -22.659203 0
```

```
5   5 -1.2576574 -13.388658 -64.142352 0
6   6  2.0352141  21.441742  89.046810 0
7   7  0.5070065  45.288301 160.481134 0
8   8 -1.5639607  -6.724773 -28.893768 0
9   9  0.2531457  11.867015  49.288773 0
10 10 -2.5613665  22.544334  73.960261 0
```

This can be modified (obviously!). Suppose you need two variables, `x1` and `x2`.

```
SELECT x1, x2
FROM sample
```

Here's a more advanced query where we select rows where `var_1` > 10

```
SELECT x1, x2
FROM sample
WHERE x2 >= 10
```

```
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbGetQuery(mydb,'
  select "x1","x2"
  from "sample"
  limit 10
')
```

Often you need summaries or operations by group. This is easy, just add the `GROUP BY` clause. Additionally you can perform an operation on the groups themselves using `HAVING`. Below, for example, we take the average of variable 1 by group having var2 > 2

```
SELECT avg(var1)
FROM db_name
GROUP BY group_var
HAVING var2 > 0
```

```
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbGetQuery(mydb,'
  select avg("x1")
  from "sample"
  group by "g"
```

```
  having x2 >= 0
')
```

```
[1] avg("x1")
<0 rows> (or 0-length row.names)
```

What is the difference? `HAVING` applies to groups as a whole whereas `WHERE` applies to individual rows. If you have both, the `WHERE` clause is applied first, the `GROUP BY` clause is second - so only the individual rows that meet the `WHERE` clause are grouped. The `HAVING` clause is then applied to the output. Then only the groups that meet the `HAVING` condition will appear.

Suppose you need both:

```
SELECT AVG(var_3)
FROM db_name
WHERE var1 >= 10
GROUP BY group_var
HAVING var_2 > 5
```

Above you'll get the average of variable 3 from *db_name* only for the individual rows where *var*_1 is greater than 10 grouped by group_var where, within the groups their associated *var*_2 value is greater than 5.

```
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")

dbGetQuery(mydb,'
  select avg("y")
  from "sample"
  where "x1" >= 3
  group by "g"
  having x2 >= 0
')
```

```
  avg("y")
1 49.22952
```

Data types

Here are a few common data types. For a full list go to https://www.postgresql.org/docs/current/datatype.html.

Table 1: Data Types

Data Type	What does it do?
int	signed four-byte integer
numeric	exact number. use when dealing with money
varchar	variable-length character string
time	time of day (no time zone)
timestamp	date and time (no time zone)
date	calendar date (year, month, day)

For a technical discussion of the difference between float4 and float8 see this post: https://stackoverflow.com/questions/16889042/postgresql-what-is-the-difference-between-float1-and-float24.

NULLS

Use `IS NOT NULL` to get rid of nulls. Usually used after the `WHERE` clause.

Aliasing

Sometimes you need to alias variables. This is especially necessary when merging as you can overwrite columns that have the same name that aren't explicitly part of the merge.

```
SELECT var AS new_var_name
FROM ...
```

You can also alias data frames - this is useful when you have multiple data frames.

```
SELECT var AS new_var_name
FROM df1 a
```

```
mydb <- dbConnect(RSQLite::SQLite(), "sample.sqlite")
```

```
dbGetQuery(mydb,'
  select "g" as "group", ROUND(avg("y"),2) as "new_average"
  from "sample" "b"
  group by "g"
')
```

```
  group new_average
1     0       39.80
2     1       37.17
```

Converting Data Types

Sometimes the data is in one format and you need it in another. You can use `CAST` to do this

```
CAST(variable AS int/numeric/varcar/time)
```

Sometimes you need to get rid of nulls and coerce them to zeroes, to do that use `COALESCE`

```
COALESCE(variable, 0)
```

Extracting

A lot of times when dealing with dates you'll need a range or only part of the information given. To extract this data, you need the command `extract`.

This extracts a year from a date:

```
EXTRACT(Year from date_var)
```

This extracts an epoch (i.e. the time difference) between the end date and the start date:

```
EXTRACT(EPOCH from endvar-startvar)
```

Suppose, however, that you only want the days in the epoch. That's surprisingly easy with the following code:

```
EXTRACT(Day from endvar-startvar)
```

Aggregate Functions

Note that in the table below all of the functions EXCEPT count ignore null values.

Table 2: Aggregate Types

Aggregate Fcn	What does it do?
`MIN()`	returns the smallest value within the selected column
`MAX()`	returns the largest value within the selected column
`COUNT()`	returns the number of rows in a set
`SUM()`	returns the total sum of a numerical column
`AVG()`	returns the *mean* value of a numerical column. Getting the median requires `PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY var1)`. See below for more info
`GREATEST(var1, var2)`	Greatest rowwise among var1 and var2
`LEAST(var1, var2)`	Least rowwise among var1 and var2

Before we go on, a brief digression on getting the median. For reasons known only to the creators of SQL, getting the median is fantastically difficult. Suppose you want the median as a decimal rounded to the second significant digit. You'd need to write `ROUND(PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY var1)::DECIMAL, 2)`.[1]

The functions `MIN()` to `AVG()` operate globally but sometimes you need the biggest/smallest out of a single record (i.e. locally/within a row). You can do that with `GREATEST()` and `LEAST()`. Note that these also work on characters. An example will help clarify why this feature is useful.

Suppose you want to see the number of flights between a pair of cities (e.g. Austin and Dallas) but you don't care about where the plane begins. In this case the command `CONCAT(departure_city, '-' arrival_city)`[2] will create **2** separate entries for Austin-Dallas and Dallas-Austin which is not what the question asked for. So you have to use `GREATEST` and `LEAST`.

Greatest gets the highest alphabetically/greatest date/number out of a variable PER RECORD (i.e per row). Max gets the most **OVER ALL RECORDS**. Why does this difference matter? Suppose Abilene and Amarillo are in the data. Then if you used `MAX()` every row would be in the Abilene and Amarillo group.

[1] If you ask why, I'll give you my favorite coding answer.
[2] See Section **??** for more details

Going back to our Dallas/Austin example, `GREATEST(departure_city, arrival_city)` would give us `Austin` and `LEAST(departure_city, arrival_city)` gives us `Dallas` in a row with a flight from Austin to Dallas or in a row with a flight from Dallas to Austin. In a row with a flight from Austin to London the command would give us `Austin` and `London`. So to combine these to create a unique ID, we could type `CONCAT(GREATEST(destination_location, source_location),'-',LEAST(destination_location, source_location))` and that would give us `Austin-Dallas` whenever the these two cities appeared in destination location and source location.

Percentiles

As you could see from above, extracting the median is difficult. If you want a bunch of percentiles, the problem is even worse.

```
SELECT
UNNEST(array[0, 0.25, 0.5, 0.75, 1]) AS percentile,
UNNEST(PERCENTILE_CONT(array[0, 0.25, 0.5, 0.75, 1]) within group (order by var1)) ---- No
FROM ...
```

You have two options for calculating percentiles `PERCENTILE_CONT` and `PERCENTILE_DISC`. `PERCENTILE_CONT` will interpolate values while `PERCENTILE_DISC` will give you values only from your data. Suppose you have 2,3,4,5. PERCENTILE_CONT would give you 3.5, PERCENTILE_DISC gives you 3.

Merging

There are four types of joins:

- INNER JOIN/JOIN
  - Joins all common records between tables
- LEFT JOIN
  - Joins the matching records in the right frame (i.e. the one after the LEFT JOIN clause) with all the records in the left frame (i.e. the one after the FROM clause)
- RIGHT JOIN
  - Joins all of the records in the right frame (i.e. the one after the RIGHT JOIN clause) with all the matching records in the left frame (i.e. the one after the FROM clause)
- FULL JOIN

  - All the records in both frames

These joins are all on some variable.

```
SELECT a.var1, a.var2, a.id, b.var3, b.var4, b.id1 ---- note the aliases up here.  This is
FROM df1 a ---- alias dataframe 1 as a
INNER JOIN df2 b ----- alias dataframe 2 as b
ON a.id = b.id
```

You can use `OR` in the join to join on either value. `AND` works too.

```
SELECT a.var1, a.var2, a.id, a.id2, b.var3, b.var4, b.id1, b.alt_id <---- note the aliases
FROM df1 a <---- alias dataframe 1 as a
INNER JOIN df2 b <----- alias dataframe 2 as b
ON a.id = b.id OR a.var1 = b.alt_id <---- this joins if EITHER is true
```

- UNION
  - Concatenates queries. Does not allow duplicates

```
SELECT ...
FROM ...
UNION
SELECT ...
FROM ...
```

- UNION ALL
  - Concatenates queries. Allows duplicates

Window Functions

Suppose you need to do something within a window like find all flights within 3 days. Here you need a window function. The basic syntax is as follows:

$\underbrace{\ldots}_{\text{Some fcn}}$ OVER(

$\underbrace{\text{PARTITION BY var1}}_{\text{group by var1}}$

$\underbrace{\text{ORDER BY var2}}_{\text{order by var2}}$) AS newvar

In addition to the functions in Section **??**, here are a bunch of useful functions. For a more comprehensive list, go to https://www.postgresql.org/docs/current/functions-window.html

Table 3: Window Functions

Window Function	What does it do?
`lag()`	lags the data 1, lag(2) would lag 2 rows
`lead()`	opposite of lag()
`rank()`	ranks rows. Suppose you have two rows tied for first, the rankings would go 1,1,3
`dense_rank()`	ranks rows without skipping.Suppose you have two rows tied for first, the rankings would go 1,1,2
`ntile()`	splits the data into n groups, indexed by an integer, as equally as possible. `ntile(4)` for example, gives us quartiles if ordered properly
`cume_dist()`	cumulative distribution

Bounding window functions

Sometimes you need to search within a certain **window** in a group as opposed to within an entire group. Suppose we wanted a moving average within the last three years. We could do that by properly bounding our query. The bounds come after the `ORDER BY` clause.

```
SELECT AVG(var1) OVER  <---- give us the mean of variable 1
 (
 PARTITION BY country <---- group by country
 ORDER BY year desc <---- order by year descending
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW <---- gives us the last 3 years.  includes the c
)
FROM df1
```

We can be fairly creative with the bounds using the following building blocks:

Bounds	What does it do?
`n PRECEDING`	2 Preceding gives us the 2 prior rows not including the current row
`UNBOUND PRECEDING`	All rows up to but not including the current row
`CURRENT ROW`	Just the current row
`n PRECEDING AND CURRENT ROW`	n preceding and the current row
`n FOLLOWING`	n following but not including the current row
`UNBOUND FOLLOWING`	n preceding and the current row

Conditionals

Sometimes you don't just need an average, you need a conditional average or sum. This can be done with the `FILTER` command

```
sum(var) FILTER(WHERE ...) AS ...
```

You can create a new variable based on the values of other variables. You do that with `CASE WHEN`

```
CASE
  WHEN [condition] THEN [result]
  WHEN [condition2] THEN [result2]
  ELSE [result3] END AS new_conditional_variable
```

Note that the `CASE WHEN ...` can be used in the `GROUP BY` command to create groups.

Dates

Dates are difficult to deal with. You just have to memorize these commands.

Table 5: Datetime Functions

Function	What does it do?
`MAKE_DATE(year,month,day)`	makes dates
`MAKE_TIMESTAMP(year,month,day, hour, minute, second)`	makes timestamps
`MAKE_INTERVAL(year,month,day, hour, minute, second)`	makes intervals
`DATE(datetime_var)`	extracts dates from datetimes

There are variations. Suppose you wanted to find all processes that lasted less than 10 days. You could use the command `MAKE_INTERVAL(days < 10)`

Common Table Expressions

Sometimes you need to create a separate table that you can then extract data from to avoid conflicts like using a window function in the `WHERE` clause. You do this with a common table expression (CTE).

The basic syntax is as follows:

```
WITH cte1 AS (
SELECT ...
FROM db1
)

SELECT ...
FROM ... (likely db1 or db2)
```

You're not limited to one CTE, you can have multiple if you want:

```
WITH RECURSIVE cte1 AS (
SELECT ...
FROM db1
), ---- need the parenthesis and comma here otherwise you get an error!
cte2 AS (
SELECT ...
FROM db2
)

SELECT ...
FROM ... (likely db1 or db2)
```

Below is an an example of using CTEs to avoid a window function in the `WHERE` clause.[3] The query is from a problem that asks you to find the second longest flight between two cities. This is a bit of a tricky problem because it requires a window function, concatentaion, ordering text strings, and a common table expression.

```
WITH tmp AS (SELECT
id,
destination_location,
source_location,
flight_start,
flight_end,
dense_rank() OVER (
    PARTITION BY CONCAT(GREATEST(destination_location, source_location),'.',LEAST(destinat
    ORDER BY extract(epoch from flight_end - flight_start) desc
) AS flight_duration
FROM flights)
```

[3]See https://www.interviewquery.com/questions/second-longest-flight for full problem details