STAT 209: SQL Part I

Code

Basic query components in SQL

Goal

Learn the SQL equivalents of the basic "five verbs" from `dplyr` , and practice using them to pull data from large databases stored on a remote server.

Setting up the connection

Before we can interact with the data, we need to set up a connection to the server that hosts the database. This is similar to what you do when you set up your RStudio account to talk to the GitHub servers: you need to supply the address where the data is, and a set of credentials to log in to the remote server.

The database we'll work with is hosted at Smith College where the first author of your textbook teaches; and the authors have provided a convenience function with a general use set of credentials to make connecting with that database quick and easy.

Code:

Code

We can see what data tables are available with `dbListTables()` .

Code:

Code

```
## [1] "airports" "carriers" "flights"  "planes"
```

Interacting with arbitrary databases

(Skip this section for now; come back to this later if you want to use SQL with data other than `scidb`)

For more general usage (that is, to interact with databases other than `scidb` at Smith), we can use the generic `dbConnect()` function. You can see how this is done by peeking at the source code or `dbConnect_scidb()` :

Code

Code

```
## function (dbname, ...)
## {
##     DBI::dbConnect(RMySQL::MySQL(), dbname = dbname, host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com",
##         user = "mdsr_public", password = "ImhsmflMDSwR")
## }
## <bytecode: 0x55d72591c660>
## <environment: namespace:mdsr>
```

So using this function with the argument "airlines" is equivalent to typing

```
dbConnect(RMySQL::MySQL(),
    dbname = "airlines",
    host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com",
    user = "mdsr_public",
    password = "ImhsmflMDSwR")
```

This is fine in this case since `mdsr_public` is a read-only account that has been set up for anyone to use, and so privacy of credentials is not a big deal. However, for more general usage, it's a good idea to store your credentials in a configuration file that you keep locally, instead of typing out your password in your source code.

The config file should be called `.my.cnf` (note the leading `.`, which is a convention for this sort of file; note that this makes it hidden if using a standard file browser), placed in your home directory, and be formatted as follows

```
[scidbAirlines]
dbname = "airlines"
host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com"
user = "mdsr_public"
password = "ImhsmflMDSwR"
```

where the part in square brackets can be any shorthand you want to use for this database. Then you can open the connection by typing

Code:

Code

The resulting R object (called `con`) is equivalent to the object `db` we created above using the helper function that hardcoded the access credentials for us.

Constructing a `tbl` view of the dataset

Sometimes you can avoid having to write much SQL code by creating a "view" of the dataset that you can interact with as though it were an R-style data frame (technically an instance of the `tbl` class)

Here's how to create a `tbl` view of the `flights` data table **Code:**

Code

(You could do the same with other tables from the list you printed out with `dbListTables()` above)

The basic SQL verbs

You can do a lot of data-wrangling by interacting with this `tbl` view, without ever writing a single line of SQL code. However, for the cases when that doesn't work, let's dive into writing basic SQL queries.

Here's a summary list of the basic verbs and what they're used for (reproduced from MDSR):

`SELECT` allows you to list the columns, or functions operating on columns, that you want to retrieve. This is an analogous operation to the `select()` verb in `dplyr`, potentially combined with `mutate()`.

`FROM` specifies the table where the data are.

`JOIN` allows you to stitch together two or more tables using a key. This is analogous to the `join()` commands in `dplyr`.

`WHERE` allows you to filter the records according to some criteria. This is an analogous operation to the `filter()` verb in `dplyr`.

`GROUP BY` allows you to aggregate the records according to some shared value. This is an analogous operation to the `group_by()` verb in `dplyr`.

`HAVING` is like a `WHERE` clause that operates on the result set—not the records themselves. This is analogous to applying a second `filter()` command in `dplyr`, after the rows have already been aggregated.

`ORDER BY` is exactly what it sounds like—it specifies a condition for ordering the rows of the result set. This is analogous to the `arrange()` verb in `dplyr`.

`LIMIT` restricts the number of rows in the output. This is similar to the R command `head()`, but somewhat more versatile.

Image Source: Baumer et al. *Modern Data Science with R*.

Note: SQL is less flexible than `dplyr` about what order the verbs show up in. The order in the above table is the canonical one, and verbs lower in the list must appear after verbs higher in the list. We won't always use every verb, but if we use one, it can't occur after verbs lower in the list. And we must always include at least a `SELECT` and a `FROM` clause to specify the fields (variables/"columns") we want to return and the table from which we want to get them. Thus the simplest query which is equivalent to just printing out a data frame called `my_data` in R is

```
SELECT *
    FROM my_data
```

where the `*` is a “wildcard” that means “everything”.

Here’s a table summarizing how to translate between `dplyr` verbs and SQL verbs (also reproduced from MDSR):

Concept	SQL	R

Filter by rows & columns	`SELECT col1, col2`
	`FROM a`
	`WHERE col3 = 'x'`
		`a %>%`
		`filter(col3 == "x")`
		`%>%`
		`select(col1, col2)`

Aggregate by rows	`SELECT id, sum(col1)`
	`FROM a`
	`GROUP BY id`
		`a %>%`
		`group_by(id) %>%`
		`summarize(sum(col1))`

Combine two tables	`SELECT *`
	`FROM a`
	`JOIN b ON a.id = b.id`
		`a %>%`
		`inner_join(b, by =`
		`c("id" = "id"))`

Table 12.1: Equivalent commands in SQL and R, where *a* and *b* are SQL tables and R `data.frames`.

Image Source: Baumer et al. *Modern Data Science with R*

Running SQL queries in Markdown

In a Markdown document, you can create an executable raw SQL query by creating a code chunk that opens with `{sql connection=db}` (where `db` is whatever you named your connection in a previous R code chunk) in place of the `r` that is usually there. This tells RStudio that this chunk is to be interpreted as SQL code, and that the database we are querying is accessed through the connection called `db` in our environment.

Before we do any actual queries, let’s get a feel for the structure of the database.

We used `dbListTables()` to list the tables in a database using R code; the SQL equivalent of this is `SHOW TABLES`. Put the following in a code chunk but use the `{sql connection=db}` specification in the chunk options so that it is treated as SQL code accessing the database through the connection called `db`:

Code:

Code

4 records

Tables_in_airlines
airports
carriers
flights
planes

To see what *variables* (“fields” in database lingo) are in a particular table, we can use `DESCRIBE` (similar to `glimpse()` in R).

Code

Displaying records 1 - 10

Field	Type	Null	Key	Default	Extra
year	smallint(4)	YES	MUL	NA	
month	smallint(2)	YES		NA	
day	smallint(2)	YES		NA	
dep_time	smallint(4)	YES		NA	
sched_dep_time	smallint(4)	YES		NA	
dep_delay	smallint(4)	YES		NA	
arr_time	smallint(4)	YES		NA	
sched_arr_time	smallint(4)	YES		NA	
arr_delay	smallint(4)	YES		NA	
carrier	varchar(2)	NO	MUL		

Your first query: `SELECT * FROM <table> LIMIT 0,<n>`

To view the first few rows of the `flights` data without creating a `tbl` view first, we can use a `SELECT * FROM <table> LIMIT 0,<n>` construction (where `n` is the number of rows we want to view)

Caution: Be very careful never to run a command like the above without the `LIMIT` component unless you know for sure that the table you're accessing is relatively small. Omitting this will cause your computer to try to retrieve and print the entirety of the database, which in this case is over 100 million records. This will likely crash your computer and also slow the server way down for everyone else.

Code:

Code

Displaying records 1 - 10

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	ca
2010	10	1	1	2100	181	159	2320	159	XE	N11137	2558	EWR	OMA	162	1133	
2010	10	1	1	1920	281	230	2214	256	B6	N659JB	562	FLL	SWF	131	1119	
2010	10	1	3	2355	8	339	334	5	B6	N563JB	701	JFK	SJU	196	1597	
2010	10	1	5	2200	125	41	2249	112	XE	N16559	5982	IAD	BNA	82	542	
2010	10	1	7	2245	82	104	2347	77	OO	N908SW	6433	LAX	FAT	37	209	
2010	10	1	7	10	-3	451	500	-9	AA	N3FRAA	700	LAX	DFW	150	1235	
2010	10	1	7	2150	137	139	2337	122	DL	N347NW	1752	ATL	IAD	70	533	
2010	10	1	8	15	-7	538	537	1	CO	N73283	1740	SMF	IAH	193	1609	
2010	10	1	8	10	-2	643	645	-2	DL	N333NW	2344	LAS	CVG	196	1678	
2010	10	1	10	2225	105	831	642	109	B6	N585JB	174	SJC	JFK	293	2570	

Note that in the above code, `flights` is referring to the table in the database, not the `tbl` variable we created above.

This is a lot of information even for just 10 cases. We can restrict the output to just the variables we care about by just listing their names separated by commas in place of the wildcard `*`. This is equivalent to the `select()` verb in `dplyr`. For example:

Code:

Code

Displaying records 1 - 10

year	month	day	carrier	flight	origin	dest
2010	10	1	XE	2558	EWR	OMA
2010	10	1	B6	562	FLL	SWF
2010	10	1	B6	701	JFK	SJU
2010	10	1	XE	5982	IAD	BNA
2010	10	1	OO	6433	LAX	FAT
2010	10	1	AA	700	LAX	DFW

year	month	day	carrier	flight	origin	dest
2010	10	1	DL	1752	ATL	IAD
2010	10	1	CO	1740	SMF	IAH
2010	10	1	DL	2344	LAS	CVG
2010	10	1	B6	174	SJC	JFK

Using `WHERE` to filter data

To restrict the output to certain cases, we use the `WHERE` verb (roughly equivalent to `filter()` in `dplyr`). As with `filter()` we can create conjunctions of filtering conditions; in SQL we just use the `AND` keyword. For example, to get only United flights on June 29, 2012, we can do

Code

Code

Displaying records 1 - 10

year	month	day	carrier	flight	origin	dest
2013	6	29	UA	1294	ONT	IAH
2013	6	29	UA	368	PDX	IAH
2013	6	29	UA	1481	SEA	ORD
2013	6	29	UA	1202	LAX	ORD
2013	6	29	UA	249	LAX	IAH
2013	6	29	UA	1104	ANC	DEN
2013	6	29	UA	369	SEA	IAH
2013	6	29	UA	1238	SMF	IAH
2013	6	29	UA	1197	SFO	IAH
2013	6	29	UA	455	SFO	LAS

Note the single `=` signs here, unlike in `dplyr` where we would use `==` in this context.

Filtering on variables not in the output

In `dplyr` if we want to use a variable as part of a filtering condition, it has to be part of the dataset at the time the `filter` occurs. For example, if I want to omit the `year`, `month`, `day` and `carrier` columns from the above dataset since I am only looking at data from one specific day and carrier, I would need to do the `filter()` *before* doing the `select()`; otherwise at the point when the `filter()` occurs, those variables are not present.

Code:

Code

In SQL, on the other hand, `SELECT` must always occur before `WHERE` in a query. However, we are allowed to refer to variables in a `WHERE` statement that are not in the output. In fact `WHERE` can only refer to variables in the original data, and *cannot* refer to variables calculated elsewhere in the query.

Code:

Code

Displaying records 1 - 10

flight	origin	dest
262	ORD	DEN
1741	SEA	IAH
580	SFO	BWI
1611	EWR	IAH
710	ORD	MSP
587	SFO	SNA
1443	SFO	IAH
1737	LAX	EWR
522	SFO	PDX
1173	IAD	TPA

`BETWEEN`

To get flights from a particular date range, say June 25th through 30th, 2012, we can use `BETWEEN` with `WHERE` :

Code

Code

Displaying records 1 - 10

carrier	flight
B6	580
EV	5730
UA	1482
B6	165
EV	4696
AA	1866
AA	700
FL	372
DL	1769
UA	1237

Creating new variables (SQL analog of `mutate()`)

If, however, we wanted to specify a date range that spanned parts of two different months (say, June 15th through July 14th), this would be cumbersome to write using `WHERE` statements alone. We could say

Code:

Code

but this is a bit awkward. Instead, we may want to create a new column that represents the date as a single number that we can reference.

There isn't actually a verb in SQL that directly corresponds to `mutate()` in `dplyr` ; it turns out we do this as part of the `SELECT` step, with the help of the keyword `as` which creates an "alias" for an expression.

The example below uses the `str_to_date()` function to translate year, month and date into a single value with which ordinal comparisons can be made.

Code:

Code

This produces an error! Why?

Filtering on calculated variables: `HAVING`

Remember we said above that `WHERE` only works with variables that exist in the original dataset? That means we can't use `date` with `WHERE` , since `date` was calculated in our query.

Instead of `WHERE` , we need to use the verb `HAVING` , which works much the same way, but allows us to use calculated variables. The reason these are two different verbs is similar to why statically typed programming languages require you to specify what data type you will pass to an argument: if the SQL engine knows what type of variable you are passing in, it allows the query to be run more efficiently, which is increasingly important as datasets get larger.

It is generally slower to operate on calculated variables than on the original variables, so if possible, it is a good idea to do any filtering that you can using a `WHERE` clause so that the number of cases that `HAVING` has to look through is reduced. For example, in the following query, the year restriction in `WHERE` is redundant with the `date` restriction in `HAVING` , but by trimming the number of cases first, the query will strain computing resources much less.

Code:

Code

Displaying records 1 - 10

date	origin	dest	flight	carrier
2012-06-16	SFO	ORD	236	UA
2012-06-16	SEA	IAH	1741	UA
2012-06-16	LAX	EWR	1000	UA
2012-06-16	SFO	EWR	1175	UA
2012-06-16	ANC	DEN	1104	UA

date	origin	dest	flight	carrier
2012-06-16	ORD	IAD	1251	UA
2012-06-16	SEA	ORD	512	UA
2012-06-16	SFO	IAH	1184	UA
2012-06-16	PDX	IAH	1719	UA
2012-06-16	ORD	CMH	1228	UA

Sorting with `ORDER BY` (cf. `dplyr::arrange()`)

To sort the output, we can use `ORDER BY`, which works like `arrange()` in `dplyr`. It has `asc` and `desc` options to control the sorting direction, and you can specify more than one clause to create nested sorts.

For example, to see all flights into JFK in the date range specified operated by United Airlines, sorted first by date and then by flight number within dates:

Code:

Code

Aggregation (equivalent of `summarize()`)

SQL doesn't have a verb equivalent to `summarize()`; just like with `mutate()` this gets handled by `SELECT` as well. We can ask for aggregated variables (which in `dplyr` is the job of `summarize()`) just as we can ask for elementwise transformations (the job of `mutate()`), using exactly the same syntax. For example, to calculate the average departure delay for all flights on June 29th, 2012, we can do

Code:

Code

1 records

avg_delay
15.8233

Note that we don't need a `LIMIT` here, since we're aggregating the dataset to a single number.

(If you forgot that the SQL function for the average is `avg()` instead of `mean()` you can do

Code:

Code

```
## Warning: `overscope_eval_next()` is deprecated as of rlang 0.2.0.
## Please use `eval_tidy()` with a data mask instead.
## This warning is displayed once per session.
```

```
## Warning: `overscope_clean()` is deprecated as of rlang 0.2.0.
## This warning is displayed once per session.
```

```
## <SQL> AVG(`dep_delay`) OVER ()
```

though it turns out we don't need the quotes, and we can leave out the `OVER()` clause since it's empty anyway.

There are two ways to get the number of records being aggregated over (for which we would use `n()` in `dplyr`): either `sum(1)` or `count(*)`:

Code:

Code

1 records

N1	N2	avg_delay
18413	18413	15.8233

`GROUP BY`

Conveniently, the SQL verb equivalent to `dplyr`'s `group_by()` is also called `GROUP BY`. Except now it goes toward the end of the query, after the aggregations we want are specified, and we need to explicitly indicate that we want the grouping variable included in the output (this happened automatically in `dplyr`). To compute average departure delay on a specific day by carrier, and sort carriers in ascending order of mean delay:

Code

Displaying records 1 - 10

carrier	num_flights	avg_delay
HA	226	1.5708
AS	457	2.1072
FL	663	3.7496
US	1192	3.9010
DL	2224	8.0692
F9	247	9.4372
YV	414	11.3237
OO	1846	12.1056
AA	1480	14.9311
EV	2327	17.1104

Notice that, unlike `WHERE` , the `ORDER BY` component here is sorting the output based on what shows up in the results, not what was in the original data.

Exercise 1 Suppose we want to restrict our results to bigger airlines; namely those with over 1000 flights that day. Modify the above query to achieve this. (Hint: you won't need to modify the actual grouping and summarization, but you'll need to "filter" using your summary variable.)
Be cognizant of the "canonical order" of the verbs!