More SQL

brought to you by:
Frank Burkholder (Neil Bezdek, Taryn Heilman, Jordan Hagan, Ed White) DSI

Objectives

- Debrief from morning
  - You realize that you made your own PostgreSQL database this morning, right? Given .csv files, and a psql script where you defined the schema associated with those csv files, you made a queryable PostgreSQL database.
- Clarify two things: SELECT DISTINCT and Aliases
- Build on understanding of JOINs
  - Joining more than two tables in one query
  - Joining to the same table multiple times (hint: aliases are key) <- self join
  - Joining to subqueries
- Learn to use temporary tables
- SQL Best Practices

SELECT DISTINCT

TABLE(S) → **QUERY** → **OUTPUT**

cars

make	model	category
Ford	Explorer	SUV
Ford	Focus	Sedan
Ford	Taurus	Sedan
Ford	Excursion	SUV
Ford	Expedition	SUV
Toyota	4Runner	SUV
Toyota	Highlander	SUV
Toyota	Camry	Sedan

```
SELECT DISTINCT
    make
FROM
    cars;
```

make
Ford
Toyota

```
SELECT DISTINCT
    make,
    category
FROM
    cars;
```

make	category
Ford	SUV
Ford	Sedan
Toyota	SUV
Toyota	Sedan

Aliases

- In Postgres, Aliases can NOT be used in WHERE or HAVING clauses
- Aliases can be used in GROUP BY clauses
- This is an artifact of the order of operations (thanks Ed)

```
SELECT
    type AS meal_type,
    AVG(price) AS avg_price
FROM
    meals
WHERE
    type != 'french'
GROUP BY
    meal_type
HAVING
    AVG(price) > 2
```

Recall order of syntax & evaluation

Order of Syntax	Order of Evaluation
`SELECT`	5 - Targeted list of columns evaluated and returned
`FROM`	1 - Product of all tables is formed
`JOIN / ON`	
`WHERE`	2 - Rows filtered out that do not meet condition
`GROUP BY`	3 - Rows combined according to GROUP BY clause and aggregations applied
`HAVING`	4 - Aggregations that do not meet that HAVING criteria are removed
`ORDER BY`	6 - Rows sorted by column(s)
`LIMIT`	7 - Final table truncated based on limit size
`;`	8 - Semicolon included as reminder

Queries with Multiple JOIN Clauses

Recall the original hypothetical table that we used as the basis for a 3-table database:

purch_id	cust_name	cust_state	description	price	date
1	Kayla	CO	skis	$300	10/30
2	Kayla	CO	goggles	$75	11/14
3	Erich	CO	snowboard	$400	11/18
4	Adam	NY	skis	$300	12/11
5	Frank	AZ	skis	$300	12/19
6	Adam	NY	goggles	$75	12/24

Queries with Multiple JOIN Clauses (cont'd)

How would we combine the tables below in *a single query* to re-form the original table?

customers

cust_id	cust_name	cust_state
1	Kayla	CO
2	Erich	CO
3	Adam	NY
4	Frank	AZ

products

prod_id	description	price
1	skis	300
2	goggles	75
3	snowboard	400

purchases

purch_id	cust_id	prod_id	date
1	1	1	10/30
2	1	2	11/14
3	2	3	11/18
4	3	1	12/11
5	4	1	12/19
6	3	2	12/24

Queries with Multiple JOIN Clauses (cont'd)

Recall that the first part evaluating of any query is to form a product of all tables based on the FROM and JOIN clauses.

```
SELECT …
FROM purchases AS p
```

p.purch_id	p.cust_id	p.prod_id	p.date
1	1	1	10/30
2	1	2	11/14
3	2	3	11/18
4	3	1	12/11
5	4	1	12/19
6	3	2	12/24

Queries with Multiple JOIN Clauses (cont'd)

Recall that the first part evaluating of any query is to form a product of all tables based on the FROM and JOIN clauses.

```
SELECT …
FROM purchases AS p
LEFT OUTER JOIN customers AS c  ON p.cust_id = c.cust_id
```

p.purch_id	p.cust_id	p.prod_id	p.date	c.cust_id	c.cust_name	c.cust_state
1	1	1	10/30	1	Kayla	CO
2	1	2	11/14	1	Kayla	CO
3	2	3	11/18	2	Erich	CO
4	3	1	12/11	3	Adam	NY
5	4	1	12/19	4	Frank	AZ
6	3	2	12/24	3	Adam	NY

Queries with Multiple JOIN Clauses (cont'd)

Recall that the first part evaluating of any query is to form a product of all tables based on the FROM and JOIN clauses.

```
SELECT …
FROM purchases AS p
LEFT OUTER JOIN customers AS c  ON p.cust_id = c.cust_id
LEFT OUTER JOIN products AS pr  ON p.prod_id = pr.prod_id
```

p.purch_id	p.cust_id	p.prod_id	p.date	c.cust_id	c.cust_name	c.cust_state	pr.prod_id	pr.description	pr.price
1	1	1	10/30	1	Kayla	CO	1	skis	300
2	1	2	11/14	1	Kayla	CO	2	goggles	75
3	2	3	11/18	2	Erich	CO	3	snowboard	400
4	3	1	12/11	3	Adam	NY	1	skis	300
5	4	1	12/19	4	Frank	AZ	1	skis	300
6	3	2	12/24	3	Adam	NY	2	goggles	75

Queries with Multiple JOIN Clauses (cont'd)

Then we specify which columns we want to keep,
and we have our answer.

```
SELECT
      p.purch_id,
      c.cust_name,
      c.cust_state,
      pr.description,
      pr.price,
      p.date
FROM
      purchases AS p
LEFT OUTER JOIN
      customers AS c
            ON p.cust_id = c.cust_id
LEFT OUTER JOIN
      products AS pr
            ON p.prod_id = pr.prod_id;
```

p.purch_id	c.cust_name	c.cust_state	pr.description	pr.price	p.date
1	Kayla	CO	skis	300	10/30
2	Kayla	CO	goggles	75	11/14
3	Erich	CO	snowboard	400	11/18
4	Adam	NY	skis	300	12/11
5	Frank	AZ	skis	300	12/19
6	Adam	NY	goggles	75	12/24

Joining to the Same Table Twice
(a.k.a. Self Join)

call_history

caller_id	receiver_id	date
3	4	10/30
2	4	11/14
3	2	11/18
4	1	12/11
2	3	12/19

customers

id	name
1	Kayla
2	Erich
3	Adam
4	Frank

QUERY

```
SELECT
  caller.name AS caller_name,
  receiver.name AS receiver_name,
  ch.date
FROM
  call_history AS ch
LEFT OUTER JOIN
  customers AS caller
ON
  ch.caller_id = caller.id
LEFT OUTER JOIN
  customers AS receiver
ON
  ch.receiver_id = receiver.id;
```

OUTPUT

Query: Who called whom?

caller_name	receiver_name	date
Adam	Frank	10/30
Erich	Frank	11/14
Adam	Erich	11/18
Frank	Adam	12/11
5	NULL	12/19

Using different aliases for the same table allows us to JOIN to that table multiple times ON different fields.

Joining to the Same Table Twice

call_history

caller_id	receiver_id	date
3	4	10/30
2	4	11/14
3	2	11/18
4	1	12/11
2	3	12/19

customers

id	name
1	Kayla
2	Erich
3	Adam
4	Frank

QUERY

```
SELECT
  customers.name,
  calls_made.total_calls
FROM
  customers
LEFT OUTER JOIN
  (SELECT
     caller_id,
     count(*) AS total_calls
   FROM call_history
   GROUP BY caller_id
  ) AS calls_made
ON
  customers.id = calls_made.caller_id;
```

OUTPUT

How many calls did each person make?

name	total_calls
Kayla	NULL
Erich	2
Adam	2
Frank	1

Again, aliasing a subquery allows us to refer to it after creation (in ON clause).

Another way: Using Temp. Tables

call_history

caller_id	receiver_id	date
3	4	10/30
2	4	11/14
3	2	11/18
4	1	12/11
2	3	12/19

customers

id	name
1	Kayla
2	Erich
3	Adam
4	Frank

QUERY

```
WITH calls_made AS
  (SELECT
     caller_id,
     count(*) AS total_calls
   FROM call_history
   GROUP BY caller_id)

SELECT
  customers.name,
  calls_made.total_calls
FROM
  customers
LEFT OUTER JOIN
  calls_made
ON
  customers.id = calls_made.caller_id;
```

OUTPUT

How many calls did each person make?

name	total_calls
Kayla	NULL
Erich	2
Adam	2
Frank	1

A single temporary table can be used in place of multiple identical subqueries.

Subquery vs Temp Table vs Create/Drop Table

All three approaches yield the same results. The best one might depend on how many times you will reference newTable. And which are the most readable?

```
SELECT
  newTable.col1,
  newTable.col2
FROM
  (SELECT
     col1,
     col2,
     col3
   FROM
     anotherTable
  ) AS newTable;
```

```
WITH newTable AS
  (SELECT
     col1,
     col2,
     col3
   FROM
     anotherTable)

SELECT
  newTable.col1,
  newTable.col2
FROM
  newTable;
```

```
CREATE TABLE newTable AS
  (SELECT
     col1,
     col2,
     col3
   FROM
     anotherTable);

SELECT
  newTable.col1,
  newTable.col2
FROM
  newTable;

DROP TABLE newTable;
```

SQL Best Practices

“I spent 2 years refactoring poorly running SQL queries for a major healthcare company with 18,000+ tables in its database (some with over a billion rows in it).

Here is what I learned.”

-- Jordan Hagan, DSI alumna

SQL Best Practices

Don't use SELECT * unless you are learning about the data and trying to see what is in a table.

- People used to cite performance issues as a main reason for this. With today's technology that is not totally true any more.
- But what is true is that SQL is already pretty slow, and no reason to make it pull in every column if you don't need them all.
- It has "code smell" which means it's not wrong, it's just not a best practice.
- It makes your code unreadable to anyone else skimming it (i.e... on GitHub).

SQL Best Practices

The most important line of any SQL query you will ever write is your "FROM" statement.

- Your FROM statement dictates how the rest of the code is going to be written.
  - Joins that link back to the FROM table instead of other join tables run are much less computationally intensive because SQL is not running through all of FROM and all of the other tables to finally get the records it needs.
- I [Jordan] have never once had to write a RIGHT JOIN. If you have to, you can likely move that table to be your FROM table, and LEFT JOIN to the table you need to.
  - Not that this really matters, it's just easier to read.
- Your FROM table should be a small-medium concise table. (i.e... a site directory).

SQL Best Practices

Do not make your joins in your WHERE statement.

```
SELECT
    table1.this,
    table2.that,
    table2.somethingelse
FROM
    table1, table2
WHERE
    table1.foreignkey = table2.primarykey
AND (some other conditions)
```

It's a personal pet peeve [Jordan] but it also way more computationally intensive, and much harder to read.

SQL Best Practices

Don't use subselects (subqueries) if you can avoid it.

- Again, there are computational and readability reasons.
- Sometimes it’s necessary - but most of the time you can make it a temp table!
- Faster!
- Prettier!
- Easier to read!

Check out Jordan’s SQL code on Github.

SQL Best Practices

SQL isn't case sensitive - so make your code pretty.

- This is different for everyone!
- I [Jordan] have strong opinions on how "SELECT, CASE, WHEN, END, FROM, WHERE, ORDER BY, HAVING, and GROUP BY" should all be all capitalized. But that's just a personal preference.
- Some people like commas in the SELECT before the columns, I prefer them to all be after the column.
- Some people are crazy and like all their columns on one line, I like each one on it's own line.

Whatever you do, just be consistent!

SQL Best Practices

SQL isn't case sensitive - so make your code pretty.

```
Select
table1.this
,table2.that
,table2.somethingelse
From table1
Inner Join table2 on table1.foreignkey = table2.primarykey
Where table1.name like '%smith%'
and table2.city = 'Denver'
```

OR

```
SELECT
    tb1.this,
    tb2.that,
    tb2.somethingelse
FROM table1 as tb1
INNER JOIN table2 as tb2 on tb1.foreignkey = tb2.primarykey
WHERE tb1.name like '%smith%'
AND tb2.city = 'Denver'
```

BUT MAYBE NOT

```
select b.this, a.that, a.somethingelse
from table1 as b, table2 as a
where table1.foreignkey = a.primarykey
and b.name like '%smith%'
and a.city = 'Denver'
```

Objectives

- Debrief from morning
  - You realize that you made your own PostgreSQL database this morning, right? Given .csv files, and a psql script where you defined the schema associated with those csv files, you made a queryable PostgreSQL database.
- Clarify two things: SELECT DISTINCT and Aliases
- Build on understanding of JOINs
  - Joining more than two tables in one query
  - Joining to the same table multiple times (hint: aliases are key) <- self join
  - Joining to subqueries
- Learn to use temporary tables
- SQL Best Practices