Introduction to SQL

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

Objectives

- Describe why we use Relational Database Management Systems (RDBMS)
- Understand primary keys, foreign keys, and table relationships
- Write simple SQL queries on a single table using SELECT, FROM, WHERE, GROUP BY, ORDER BY clauses as well as aggregation functions (COUNT, AVG, etc.)
- Write more complex SQL queries using joins, subqueries, and temporary tables.
- Discuss SQL best practices
- Interact with a Postgres database from the command line
- Learn how to say PostgreSQL (link)

Motivation

O'REILLY®

2016 Data Science Salary Survey

Tools, Trends, What Pays (and What Doesn't) for Data Professionals

Key findings include:

- Python and Spark are among the tools that contribute most to salary.
- Among those who code, the highest earners are the ones who code the most.
- **SQL**, Excel, R and Python are the most commonly used tools.

An inefficient way to store data...

A single table with records of customer purchases at an outdoor sports store.

id	cust_name	cust_state	item_purchased	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

Relational Database Management Systems

A RDBMS is a type of database where **data is stored in multiple related tables**.
The tables are related through **primary** and **foreign keys**.
The same information as shown before in an RDBMS:

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

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

Primary Keys

- Every table in a RDBMS has a **primary key** that uniquely identifies that row
- Each entry must have a primary key, and primary keys cannot repeat within a table
- Primary keys are usually integers but can take other forms

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

Foreign Keys and Table Relationships

- A **foreign key** is a column that uniquely identifies a column in another table
- Often, a foreign key in one table is a primary key in another table
- We can use foreign keys to join tables

customers

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

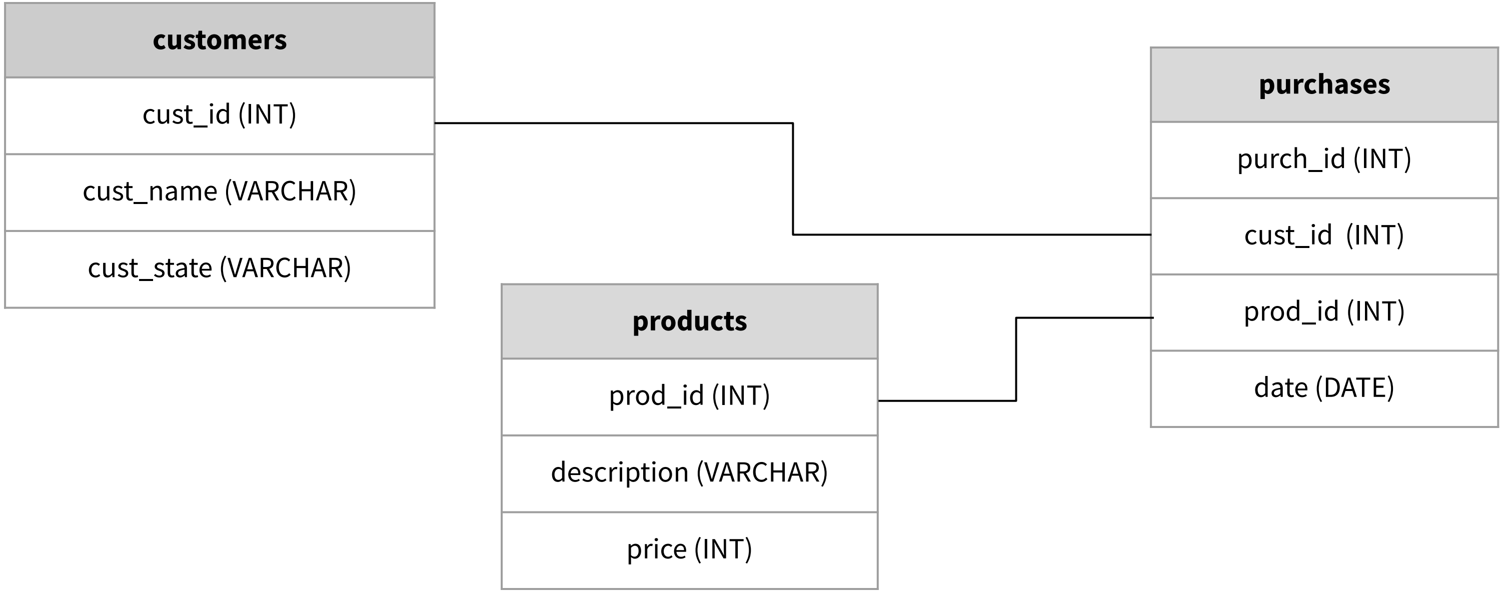
products

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

Entity Relationship Diagram (ERD)



RDBMS Terminology

- **Schema** defines the structure of a table or a database
- Database is composed of a number of user-defined **tables**
- Each table has **columns** (or fields) and **rows** (or records)
- A column is of a certain **data type** such as an *integer*, *text*, or *date*

With a new data source, your first task is typically to understand the schema.

This will likely take time and conversations with those that gave you access to the database or its data.

Structured Query Language (SQL)

SQL is the tool we use to interact with RDBMS. We can use SQL commands to:

- Create tables
- Alter tables
- Insert records
- Update records
- Delete records
- **Query (SELECT) records within or across tables**

The most critical skill for a Data Scientist--as opposed to a Data Engineer or Database Administrator--is to extract information from databases.

We will focus on writing queries in PostgreSQL, but all of the commands use similar vocabulary and syntax. (PostgreSQL syntax)

SQL Query Basics

SQL queries have three main components:

```
SELECT    # What data (columns) do you want?

FROM      # From what location (table) you want it?

WHERE     # What data (rows) do you want?
```

Note: SQL queries always return tables.

Note: SQL is a *declarative* language, unlike Python, which is *imperative*. With a declarative language, you tell the machine *what* you want, instead of *how*, and it figures out the best way to do it for you.

Query Components vs. Order of Evaluation

Order of Components	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

Formatting SQL statements

Unlike Python, whitespace and capitalization do not matter (except for strings)

```
select column1, column2 from my_table;
```

Convention is to use ALL CAPS for keywords

Line breaks and indentation help make queries more readable (especially complex ones)

```
SELECT
    column1,
    column2
FROM
    my_table;
```

Punctuation such as commas (between items under each clause) and semicolons (after each statement) are required for proper evaluation

SELECT *

TABLE(S)

customers

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

SELECT *

TABLE(S)

QUERY

customers

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

```
SELECT
    *
FROM
    customers;
```

The asterisk means “everything.”

SELECT *

TABLE(S)

customers

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

QUERY

```
SELECT
    *
FROM
    customers;
```

OUTPUT

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

The asterisk means “everything.”

Aliases

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_name AS name,
    cust_state state
FROM
    customers;
```

name	state
Kayla	CO
Erich	CO
Adam	NY
Frank	AZ

- Aliasing can be used to rename columns and even tables (more on this later).
- "AS" makes code clearer but is not necessary.
- Be careful not to use keywords (e.g. count) as aliases!

WHERE

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_name AS name,
    cust_state AS state
FROM
    customers
WHERE
    cust_state = ‘CO’ ;
```

name	state
Kayla	CO
Erich	CO

- WHERE specifies criterion for selecting specific rows (row filter)
- Note that the WHERE statement must reference the original column name, not the alias
- However, WHERE can reference a table column that is not in SELECT (e.g. cust_id)

WHERE (Multiple Criteria)

TABLE(S)

customers

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

Think about AND first, then OR
(even though the actual execution order isn’t done this way)

QUERY

```
SELECT
    cust_name AS name,
    cust_state AS state
FROM
    customers
WHERE
    (cust_state = ‘CO’
    AND cust_name = ‘Kayla’)
    OR cust_state = ‘NY’;
```

OUTPUT

name	state
Kayla	CO
Adam	NY

The order of AND and OR can get confusing with multiple conditions: Use parentheses to clarify desired order

- We can specify multiple conditions on the “WHERE” clause by using AND/OR
- Note that comparison operator uses a single equal sign (= instead of ==)

LIMIT and ORDER BY

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    *
FROM
    customers
ORDER BY
    cust_name DESC
LIMIT 3;
```

cust_id	cust_name	cust_state
1	Kayla	CO
4	Frank	AZ
2	Erich	CO

- ORDER BY is ascending by default; specify DESC for reverse sorting
- LIMIT specifies the number of records returned

SELECT DISTINCT

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT DISTINCT
    cust_state
FROM
    customers;
```

cust_state
CO
NY
AZ

- SELECT DISTINCT grabs all the unique records.
- If multiple columns are selected, then all unique combinations are returned.

CASE WHEN

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_name AS name,
    CASE WHEN cust_state = ‘CO’ THEN 1
        ELSE 0 END AS in_state
FROM
    customers;
```

When __ THEn __ Else __ END

name	in_state
Kayla	1
Erich	1
Adam	0
Frank	0

- CASE WHEN statement is the SQL version of an if-then-else statement
- Used in the SELECT clause
- Can combine multiple WHEN statements and/or multiple conditionals

Something about Coalesce

Aggregators

TABLE(S)

QUERY

OUTPUT

products

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

```
SELECT
    COUNT(*),
    MAX(price)
FROM
    products;
```

COUNT	MAX
3	400

- Aggregators combine information from multiple rows into a single row.
- Other aggregators include MIN, MAX, SUM, COUNT, STDDEV, etc.

BREAKOUT!

pets

id	name	species	age	gender	owner
1	Max	cat	8	M	Taryn
2	Belle	cat	10	F	Taryn
3	Bailey	dog	11	F	Kyrie
4	Daisy	cat	5	F	Kyrie
5	Kahlua	dog	7	F	Blair
6	Henley	dog	9	F	Megan
7	Salem	cat	1	F	Megan
8	Teeny	cat	1	F	Megan

Write queries that would return:

1) Owner(s) of Male pet(s)
2) Names of dogs
3) Names and ages of oldest two pets
4) The species of the youngest pet
5) Names of cats that are 8 years old or younger
6) Pets that are babies (<= 1 year) are expensive. Senior pets (>=8) can also be expensive. Who owns one or more expensive pets?

GROUP BY

TABLE(S)

customers

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

QUERY

```
SELECT
    cust_state as state,
    count(*)
FROM
    customers
GROUP BY
    cust_state;
```

OUTPUT

state	count(*)
CO	2
NY	1
AZ	1

- The GROUP BY clause calculates aggregate statistics for groups of data
- **Any column that is not an aggregator *must* be in the GROUP BY clause** (for example, if we added `cust_name` to the SELECT clause only, SQL would not know whether to return Kayla or Erich in the CO row)
- Any column in the GROUP BY by clause must also appear in the SELECT clause (true of Postgres but not MySQL)

GROUP BY and WHERE

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_state AS state,
    COUNT(*) AS total
FROM
    customers
WHERE
    cust_name != ‘Adam’
GROUP BY
    cust_state;
```

state	total
CO	2
AZ	1

GROUP BY and WHERE (cont'd)

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_state AS state,
    COUNT(*) AS total
FROM
    customers
WHERE
    COUNT(*) >= 2
GROUP BY
    cust_state;
```

~~state~~	~~total~~
~~CO~~	~~2~~

ERROR

- Why does the query above not work?

GROUP BY and HAVING

TABLE(S)

QUERY

OUTPUT

customers

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

```
SELECT
    cust_state AS state,
    COUNT(*) AS total
FROM
    customers
~~WHERE~~
    ~~count(*) >= 2~~
GROUP BY
    cust_state
HAVING
    COUNT(*) >= 2;
```

state	total
CO	2

- Use HAVING instead of WHERE when filtering rows *after* aggregation
- WHERE clause filters rows in the root table *before* aggregation
- Like WHERE clause, HAVING clause cannot reference an alias (in Postgres, at least)

BREAKOUT!

pets

id	name	species	age	gender	owner
1	Max	cat	8	M	Taryn
2	Belle	cat	10	F	Taryn
3	Bailey	dog	11	F	Kyrie
4	Daisy	cat	5	F	Kyrie
5	Kahlua	dog	7	F	Blair
6	Henley	dog	9	F	Megan
7	Salem	cat	1	F	Megan
8	Teeny	cat	1	F	Megan

Write queries that would return:

1) Owners and the number of pets they own
2) Owners and the count of each species of pet
3) Count of pets by gender and species
4) All owners who own two or more cats

JOINs

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

customers

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

QUERY

```
SELECT
    purchases.purch_id,
    customers.cust_id,
    customers.cust_state
FROM
    purchases
JOIN
    customers
ON
    purchases.cust_id =
    customers.cust_id;
```

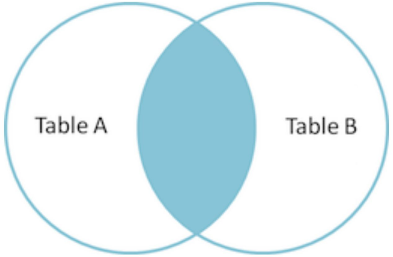
OUTPUT

purch_id	cust_id	cust_state
1	1	CO
2	1	CO
3	2	CO
4	3	NY
5	4	AZ
6	3	NY

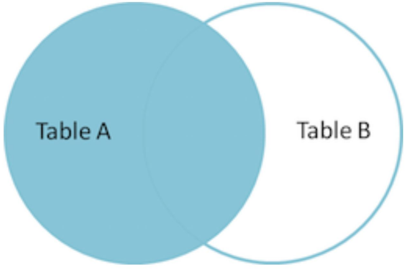
JOIN Types

```
SELECT … FROM TableA _____ JOIN TableB ON…
```

- **(INNER) JOIN:** Discards any entries that do not have match between the keys specified in the ON clause. No null/nan values.

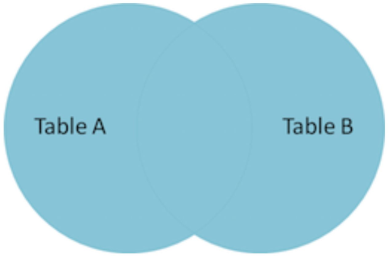


- **LEFT (OUTER) JOIN:** Keeps all entries in the left (FROM) table, regardless of whether any matches are found in the right (JOIN) tables. Some null/nan values.



  - **RIGHT (OUTER) JOIN:** Is the same, except keeps all entries in the right (JOIN) table instead of the left (FROM) table); usually avoided because it does the same thing as a LEFT join

- **FULL (OUTER) JOIN:** Keeps the rows in both tables no matter what. More null/nan values.



(INNER) JOIN

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~~	~~NULL~~	~~1~~	~~12/19~~
~~6~~	~~NULL~~	~~2~~	~~12/24~~

customers

cust_id	cust_name	cust_state
1	Kayla	CO
2	Erich	CO
3	Adam	NY
~~4~~	~~Frank~~	~~AZ~~
~~5~~	~~Neil~~	~~NY~~

QUERY

```
SELECT
    purchases.purch_id,
    customers.cust_id,
    customers.cust_state
FROM
    purchases
INNER JOIN
    customers
ON
    purchases.cust_id =
    customers.cust_id;
```

OUTPUT

purch_id	cust_id	cust_state
1	1	CO
2	1	CO
3	2	CO
4	3	NY

INNER JOIN discards records that do not have a match in both tables

LEFT (OUTER) JOIN

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	NULL	1	12/19
6	NULL	2	12/24

customers

cust_id	cust_name	cust_state
1	Kayla	CO
2	Erich	CO
3	Adam	NY
~~4~~	~~Frank~~	~~AZ~~
~~5~~	~~Neil~~	~~NY~~

QUERY

```
SELECT
    purchases.purch_id,
    customers.cust_id,
    customers.cust_state
FROM
    purchases
LEFT OUTER JOIN
    customers
ON
    purchases.cust_id =
    customers.cust_id;
```

OUTPUT

purch_id	cust_id	cust_state
1	1	CO
2	1	CO
3	2	CO
4	3	NY
5	NULL	NULL
6	NULL	NULL

LEFT OUTER JOIN retains all records from the left (FROM) tables and includes records from the right (JOIN) table if they are available

FULL (OUTER) JOIN

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	NULL	1	12/19
6	NULL	2	12/24

customers

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

QUERY

```
SELECT
     purchases.purch_id,
     customers.cust_id,
     customers.cust_state
FROM
     purchases
FULL OUTER JOIN
     customers
ON
     purchases.cust_id =
     customers.cust_id;
```

OUTPUT

purch_id	cust_id	cust_state
1	1	CO
2	1	CO
3	2	CO
4	3	NY
5	NULL	NULL
6	NULL	NULL
NULL	4	AZ
NULL	5	NY

FULL OUTER JOIN retains all records from both tables regardless of matches

Subqueries

- In general, you can replace any table name with a subquery:

```
SELECT … FROM (SELECT …)
```

- If a query returns a single value, you can use it as such:

```
...WHERE column1 = (SELECT …)
```

- If a query returns a single column, you can treat it like a vector:

```
....WHERE column1 IN (SELECT …)
```

Subquery motivation

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

OUTPUT

How many calls did each person make?

name	total_calls
Kayla	NULL
Erich	2
Adam	2
Frank	1

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

Using a subquery

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).

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

Another way: a temporary table

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;
```

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;
```

In memory (if small enough), dropped at end-of-session

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

SELECT
  newTable.col1,
  newTable.col2
FROM
  newTable;

DROP TABLE newTable;
```

written to hard disk

And you can have more than 1 Temp Table

```
WITH newTable1 AS
  (SELECT
     col1
   FROM
     anotherTable),

newTable2 AS
(SELECT
     col1
   FROM
     anotherTable2)

SELECT
   newTable1.col1 AS alias1
   newTable2.col1 AS alias2
```

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

```
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
      table2.city = ‘Denver’
```

BUT MAYBE NOT

```
SELECT b.this, a.that, a.somethingelse
FROM table1 AS b, table2 AS a
WHERE b.foreignkey = a.primarykey
AND b.name LIKE ‘%smith%’
AND a.city = ‘Denver’
```

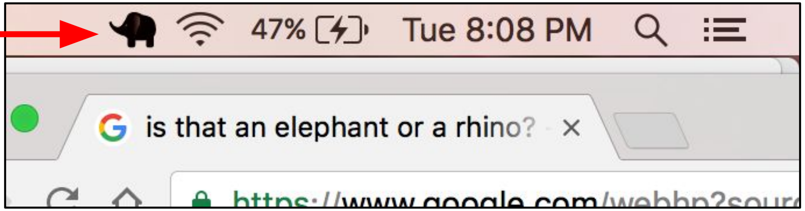
Objectives

- Describe why we use Relational Database Management Systems (RDBMS)
- Understand primary keys, foreign keys, and table relationships
- Write simple SQL queries on a single table using SELECT, FROM, WHERE, GROUP BY, ORDER BY clauses as well as aggregation functions (COUNT, AVG, etc.)
- Write more complex SQL queries using joins, subqueries, and temporary tables.
- Discuss SQL best practices
- Interact with a Postgres database from the command line
- Learn how to say PostgreSQL (link)

Appendix

WHERE IS THIS

- Instructions on Postgres installation and set-up are in the *individual.md* file
- Postgres must be running in order to use it from the command line:



Look for ambiguous ungulate icon

is that an elephant or a rhino?

- Instructions on loading the database and entering postgres prompt from the command line are also in the *individual.md* file

Load .sql file into a DB and run queries

One-time step to create a database and load .sql file. From the command line:

```
$ psql
# CREATE DATABASE MyDatabase;
# \q
$ psql MyDatabase < file.sql
```

Now you can access this database any time:

```
psql MyDataBase
```

Some Postgres commands

Useful commands from the psql interactive shell prompt:

- **\l** - list all databases
- **\d** - list all tables
- **\d <table name>** - describe a table’s schema
- **\h <clause>** - Help for SQL clause help
- **q** - exit current view and return to command line
- **\q** - quit psql
- **\i** script.sql - run script (or query)