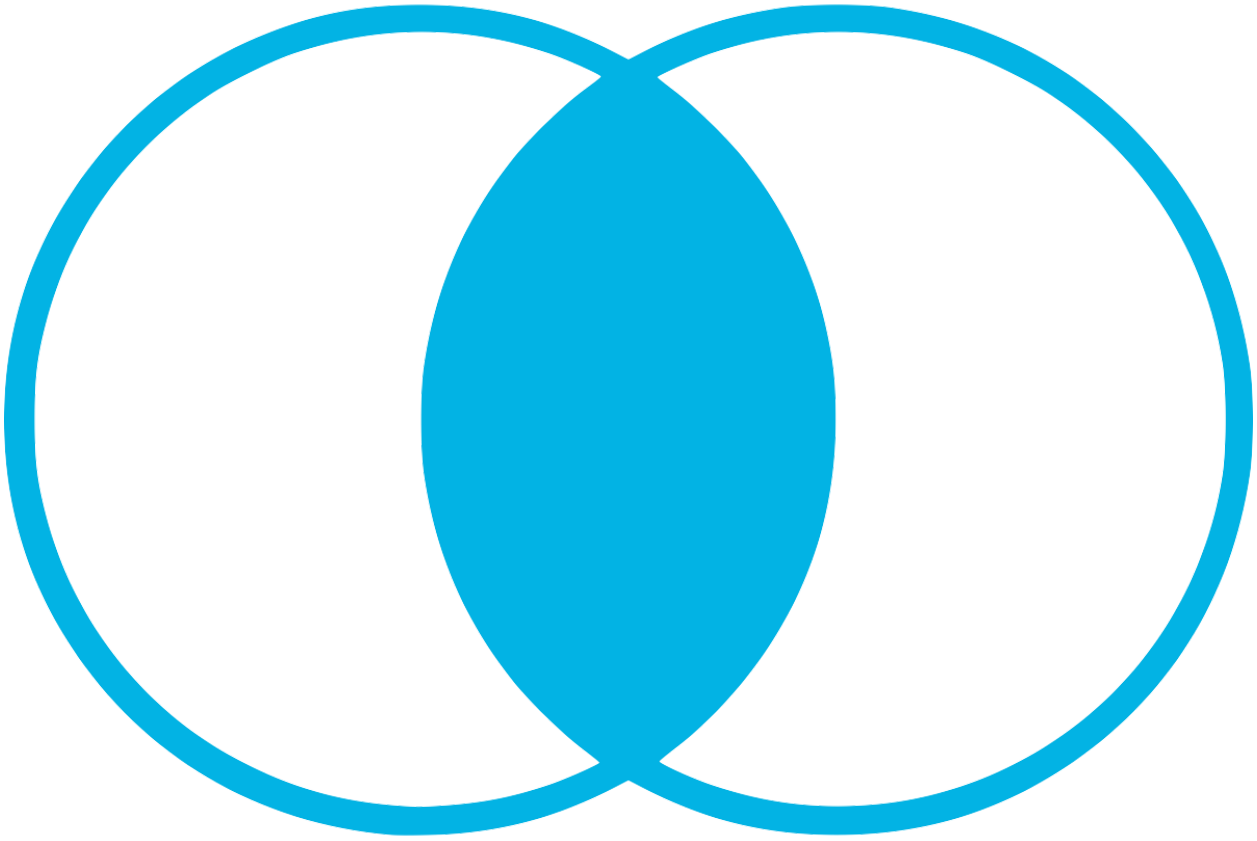
FULL OUTER JOIN

In earlier lessons, we covered inner joins, which produce results for which the join condition is matched in both tables.

Venn diagrams, which are helpful for visualizing table joins, are provided below along with sample queries. Consider the circle on the left Table A and the circle on the right Table B.

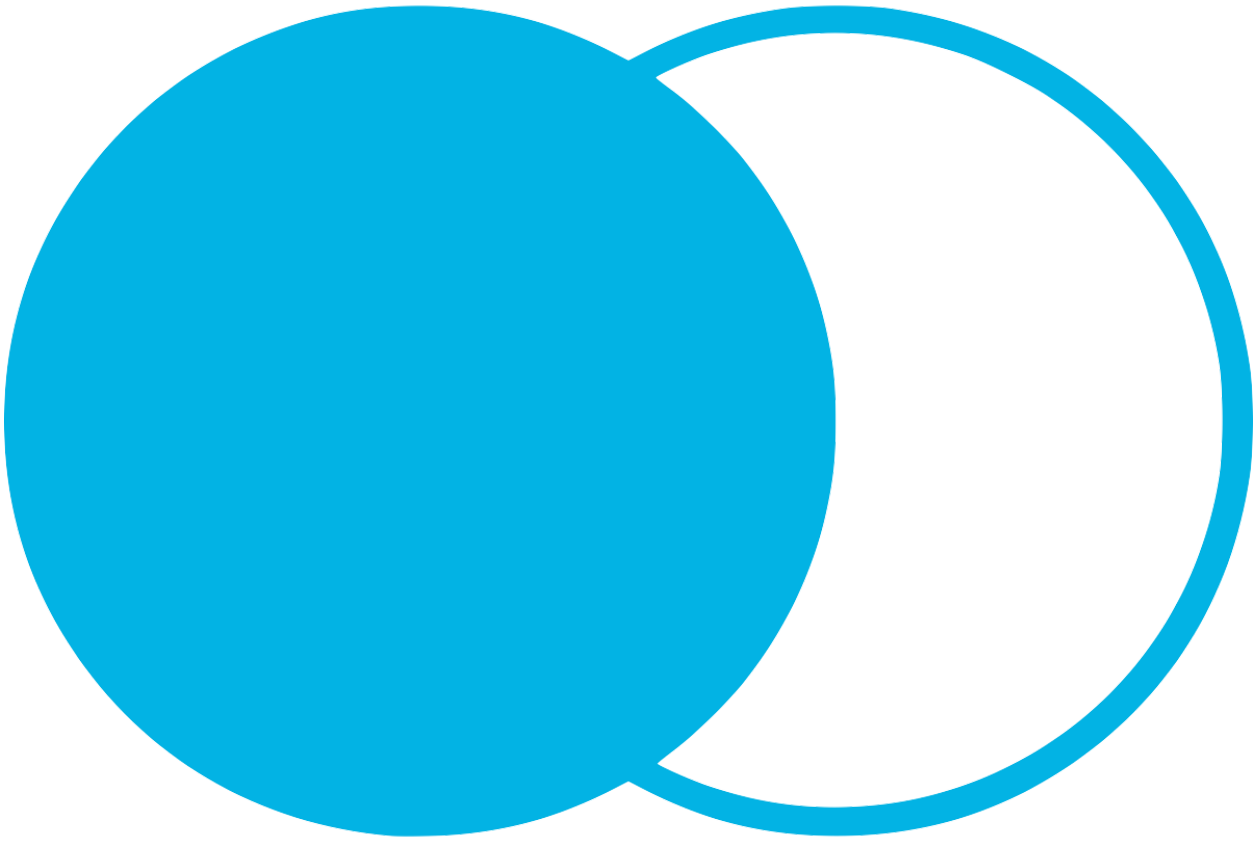


`INNER JOIN` Venn Diagram

```
SELECT column_name(s)
```

```
FROM Table_A
INNER JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

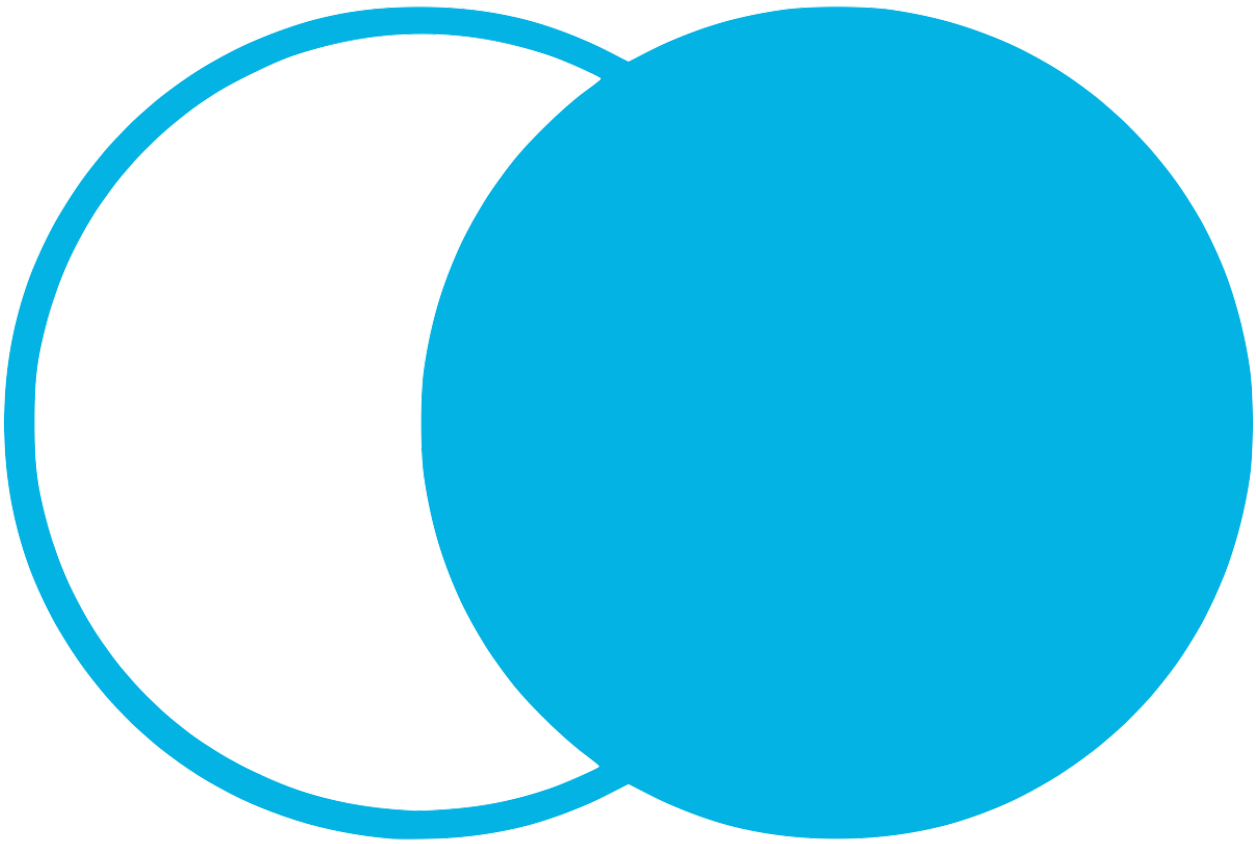
Left joins also include unmatched rows from the left table, which is indicated in the “FROM” clause.



`LEFT JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
LEFT JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

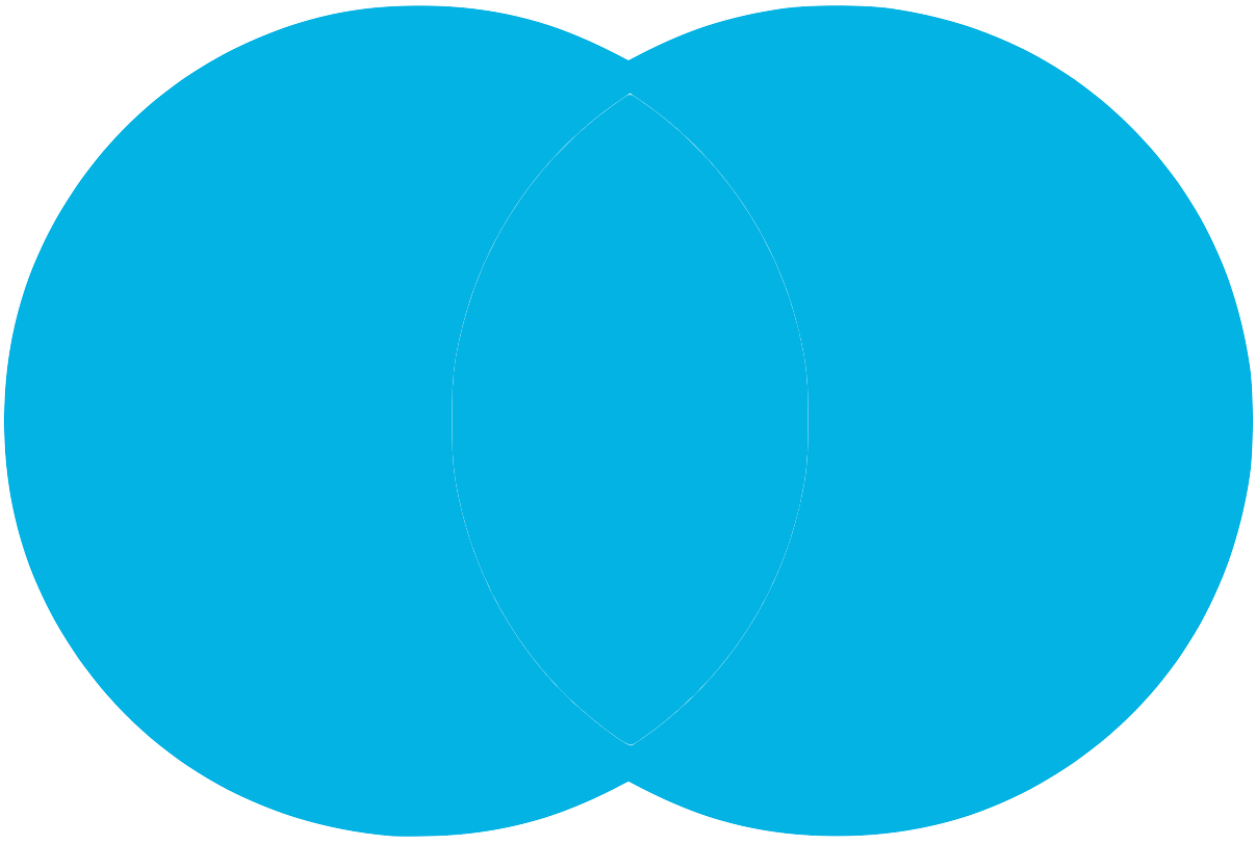
Right joins are similar to left joins, but include unmatched data from the right table -- the one that's indicated in the JOIN clause.



`RIGHT JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
RIGHT JOIN Table_B ON Table_A.column_name = Table_B.column_name;
```

In some cases, you might want to include unmatched rows from *both* tables being joined. You can do this with a full outer join.



`FULL OUTER JOIN` Venn Diagram

```
SELECT column_name(s)
FROM Table_A
FULL OUTER JOIN Table_B ON Table_A.column_name =
Table_B.column_name;
```

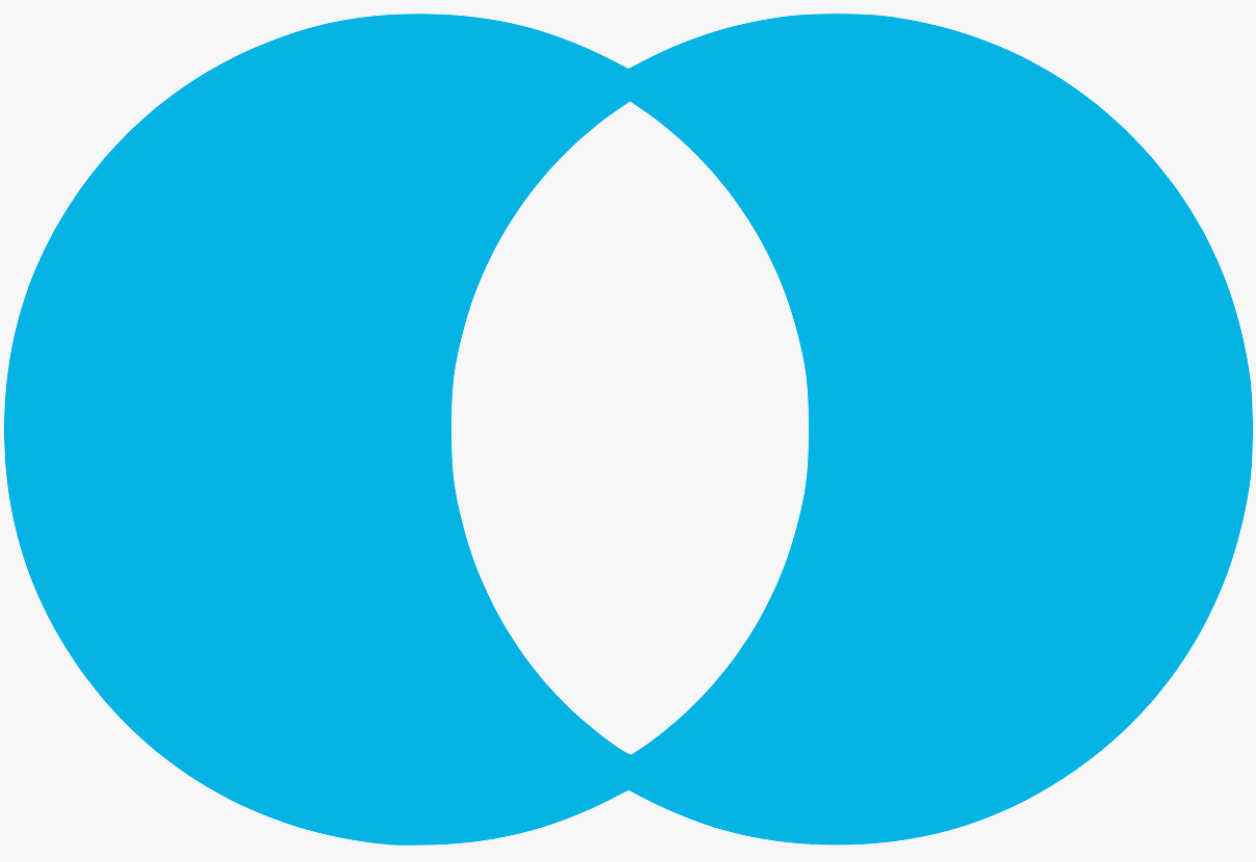
A common application of this is when joining two tables on a timestamp. Let's say you've got one table containing the number of *item 1* sold each day, and another containing the number of *item 2* sold. If a certain date, like January 1, 2018, exists in the left table but not the right, while another date, like January 2, 2018, exists in the right table but not the left:

- a left join would drop the row with January 2, 2018 from the result set
- a right join would drop January 1, 2018 from the result set

The only way to make sure both January 1, 2018 and January 2, 2018 make it into the results is to do a full outer join. A full outer join returns unmatched records in each table with null values for the columns that came from the opposite table.

If you wanted to return unmatched rows only, which is useful for some cases of data assessment, you can isolate them by adding the following line to the end of the query:

```
WHERE Table_A.column_name IS NULL OR Table_B.column_name IS NULL
```



`FULL OUTER JOIN` with `WHERE A.Key IS NULL OR B.Key IS NULL` Venn Diagram

Finding Matched and Unmatched Rows with FULL OUTER JOIN

You're not likely to use `FULL JOIN` (which can also be written as `FULL OUTER JOIN`) too often, but the syntax is worth practicing anyway. `LEFT JOIN` and `RIGHT JOIN` each return unmatched rows from one of the tables—`FULL JOIN` returns unmatched rows from both tables. `FULL JOIN` is commonly used in conjunction with aggregations to understand the amount of overlap between two tables.

Say you're an analyst at Parch & Posey and you want to see:

- each account who has a sales rep and each sales rep that has an account (all of the columns in these returned rows will be full)
- but also each account that does not have a sales rep and each sales rep that does not have an account (some of the columns in these returned rows will be empty)

This type of question is rare, but `FULL OUTER JOIN` is perfect for it. In the following SQL Explorer, write a query with `FULL OUTER JOIN` to fit the above described Parch & Posey scenario (selecting all of the columns in both of the relevant tables, `accounts` and `sales_reps`) then answer the subsequent multiple choice quiz.

Finding Matched and Unmatched Rows with FULL OUTER JOIN

```
SELECT *
  FROM accounts
 FULL JOIN sales_reps ON accounts.sales_rep_id = sales_reps.id
```

If unmatched rows existed (they don't for this query), you could isolate them by adding the following line to the end of the query:

```
WHERE accounts.sales_rep_id IS NULL OR sales_reps.id IS NULL
```

To elaborate on the rarity of `FULL OUTER JOINS` in practice, this Stack Overflow answer is helpful: When is a good situation to use a full outer join?

Joining without an Equals Sign

Inequality JOINs

Expert Tip

If you recall from earlier lessons on joins, the join clause is evaluated before the where clause -- filtering in the join clause will eliminate rows before they are joined, while filtering in the WHERE clause will leave those rows in and produce some nulls.

Inequality JOINs

The query in Derek's video was pretty long. Let's now use a shorter query to showcase the power of joining with comparison operators.

Inequality operators (a.k.a. comparison operators) don't only need to be date times or numbers, they also work on strings! You'll see how this works by completing the

following quiz, which will also reinforce the concept of joining with comparison operators.

In the following SQL Explorer, write a query that left joins the `accounts` table and the `sales_reps` tables on each sale rep's ID number *and* joins it using the `<` comparison operator on `accounts.primary_poc` and `sales_reps.name`, like so:

```
accounts.primary_poc < sales_reps.name
```

The query results should be a table with three columns: the account name (e.g. Johnson Controls), the primary contact name (e.g. Cammy Sosnowski), and the sales representative's name (e.g. Samuel Racine). Then answer the subsequent multiple choice question.

Inequality JOINs

```
SELECT accounts.name as account_name,
       accounts.primary_poc as poc_name,
       sales_reps.name as sales_rep_name
  FROM accounts
  LEFT JOIN sales_reps
    ON accounts.sales_rep_id = sales_reps.id
   AND accounts.primary_poc < sales_reps.name
```

For more details on how string comparison with `<` and `>` work in SQL, check out this excellent answer on Stack Overflow: SQL string comparison, greater than and less than operators

Expert Tip

This comes up pretty commonly in job interviews. Self JOIN logic can be pretty tricky -- you can see here that our join has three conditional statements. It is important to pause and think through each step when joining a table to itself.

Self JOINs

One of the most common use cases for self JOINs is in cases where two events occurred, one after another. As you may have noticed in the previous video, using inequalities in conjunction with self JOINs is common.

Modify the query from the previous video, which is pre-populated in the SQL Explorer below, to perform the same interval analysis except for the `web_events` table. Also:

- change the interval to 1 day to find those web events that occurred after, but not more than 1 day after, another web event
- add a column for the `channel` variable in both instances of the table in your query

You can find more on the types of INTERVALS (and other date related functionality) in the Postgres documentation here.

```
SELECT o1.id AS o1_id,
       o1.account_id AS o1_account_id,
       o1.occurred_at AS o1_occurred_at,
       o2.id AS o2_id,
       o2.account_id AS o2_account_id,
       o2.occurred_at AS o2_occurred_at
  FROM orders o1
 LEFT JOIN orders o2
   ON o1.account_id = o2.account_id
  AND o2.occurred_at > o1.occurred_at
  AND o2.occurred_at <= o1.occurred_at + INTERVAL '28 days'
ORDER BY o1.account_id, o1.occurred_at
```

Self JOINs

```
SELECT we1.id AS we_id,
       we1.account_id AS we1_account_id,
       we1.occurred_at AS we1_occurred_at,
       we1.channel AS we1_channel,
       we2.id AS we2_id,
       we2.account_id AS we2_account_id,
       we2.occurred_at AS we2_occurred_at,
       we2.channel AS we2_channel
  FROM web_events we1
 LEFT JOIN web_events we2
   ON we1.account_id = we2.account_id
  AND we1.occurred_at > we2.occurred_at
  AND we1.occurred_at <= we2.occurred_at + INTERVAL '1 day'
ORDER BY we1.account_id, we2.occurred_at
```

Appending Data via UNION

Appending Data via UNION Demonstration

SQL's two strict rules for appending data:

1. Both tables must have the same number of columns.
2. Those columns must have the same data types in the same order as the first table.

A common misconception is that column names have to be the same. Column names, in fact, **don't** need to be the same to append two tables but you will find that they typically are.

Pretreating Tables before doing a UNION

Performing Operations on a Combined Dataset

Appending Data via UNION

Write a query that uses `UNION ALL` on two instances (and selecting all columns) of the `accounts` table. Then inspect the results and answer the subsequent quiz.

Quiz 1:

```
SELECT *
    FROM accounts

UNION ALL

SELECT *
  FROM accounts
```

Quiz 2:

```
SELECT *
    FROM accounts
    WHERE name = 'Walmart'

UNION ALL

SELECT *
  FROM accounts
  WHERE name = 'Disney'
```

Quiz 3:

```
WITH double_accounts AS (
    SELECT *
      FROM accounts

    UNION ALL

    SELECT *
      FROM accounts
)

SELECT name,
       COUNT(*) AS name_count
 FROM double_accounts
GROUP BY 1
ORDER BY 2 DESC
```

Menu

Expand

NEXT

How You Can and Can't Control Performance

One way to make a query run faster is to reduce the number of calculations that need to be performed. Some of the high-level things that will affect the number of calculations a given query will make include:

- Table size
- Joins
- Aggregations

Query runtime is also dependent on some things that you can't really control related to the database itself:

- Other users running queries concurrently on the database
- Database software and optimization (e.g., Postgres is optimized differently than Redshift)

QUIZ QUESTION

Select all of the following statements that are true about tuning performance with LIMIT.

- If you have time series data, limiting to a small time window can make your queries run more quickly.
- Testing your queries on a subset of data, finalizing your query, then removing the subset limitation is a sound strategy.

- Applying `LIMIT 10` when aggregating data to one row (i.e. with a `GROUP BY`) will speed up your queries.
- When working with subqueries, limiting the amount of data you're working with in the place where it will be executed first will have the maximum impact on query run time.

JOINing Subqueries to Improve Performance

Code along with Derek in the SQL Explorer below this next video. At the end, run each of the subqueries independently to get a better understanding of how they work.

```
Menu

Expand
```

Expert Tip

If you'd like to understand this a little better, you can do some extra research on cartesian products. It's also worth noting that the FULL JOIN and COUNT above actually runs pretty fast—it's the COUNT(DISTINCT) that takes forever.

NEXT