SQL for Data Science Interviews

Alex Simonoff

Senior Data Scientist @ Spotify

Agenda

- Background
- SQL Overview
  - Basic clauses
  - Joins
  - Sub-queries and Common Table Expressions (CTEs)
- Common Mistakes and ‘Gotchas’
- Practice Questions
- Bonus Section: Window Functions

Background

Background - SQL Usage on the Job

I use SQL every day at my job. I don’t use tensorflow or build production ML models- I write SQL pipelines to power dashboards, pull A/B test metrics, deep dive on specific datasets to learn more about our products, and answer ad hoc questions posed by engineers and PMs (product managers).

If you’re looking for a job in ML engineering or AI research, it’s entirely possible you don’t need much SQL knowledge.

Background - SQL Interviews

I’ve done a few SQL interviews and none of them touched on more advanced topics beyond what is covered in today’s workshop.

The best way to practice on your own would be to look through SQL questions various companies have asked and explore the various functions at your disposal in SQL.

I’ve added a section on window functions which I’ve never actually been tested on in a SQL interview but I use often on the job.

I’ve also never been tested on things like query optimization or DML (data manipulation language) so I won’t cover that material but will provide resources!

Background - Resources

- Example SQL Interview Questions
- Harder (but still useful) SQL Interview Questions
- W3 School SQL Tutorial ⭐ (My all time favorite)
  - MySQL functions section is a great way to learn SQL functions
- Interview Query
  - Paid resource
- Introduction to Query Optimization
- Some Query Optimization Tips
- Introduction to DDL/DML/DCL

SQL Overview

For the purpose of this section, let’s assume we have three tables of sales data for a grocery store:

SALES columns: **date (mm-dd-yyyy format)**, **order_id**, **item_id**, **customer_id**, **quantity**, **revenue**

ITEMS columns: **item_id**, **item_name**, **price**, **department**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Basic Clauses

Let’s start with the basic SQL syntax:

SELECT columns

FROM table

Every non-trivial query has at least these elements

If we want to see all columns, we can use the * wildcard

We can also add ***LIMIT <n>*** to the end of our query to limit our output to the first n rows processed

ex:
Pull a sample of 10 sales

```
SELECT *
FROM sales
LIMIT 10
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Let’s start with the basic SQL syntax:

SELECT columns

FROM table

WHERE condition

We can also select a subset of the rows that we are interested in by using a WHERE clause which acts as a row filter.

We can use several operators: *LIKE, =, <, >, IS NULL*, BETWEEN, IN, etc. and have multiple conditions applied via *AND* and *OR*

ex:
Pull a sample of 10 sales from September 3rd, 2022

```
SELECT *
FROM sales
WHERE date = ‘09-03-2022’
LIMIT 10
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Common Aggregate Functions:

COUNT(column)

Counts all non null rows in a column (COUNT(*) will count all rows in a table)

COUNT(DISTINCT column)

Counts all non duplicate, non null row values in a column

SUM(column) and AVG(column)

Calculates the sum or average all values of a column

MIN(column) and MAX(column)

Calculates the minimum or maximum value of a column

SQL Aggregate Syntax:

```
SELECT columns,
    aggregate_fn(column)
FROM table
WHERE condition
GROUP BY columns
```

We **must** group by all non aggregate columns, but we also can have aggregate functions alone without additional columns to group by

ex:
For each day in September 2022, how much revenue did we generate and how many sales did we have?

```
SELECT date,
SUM(revenue) as rev,
COUNT(distinct order_id) as
sales
FROM sales
WHERE date between
‘09-01-2022’ and ‘09-30-2022’
GROUP BY date
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

SQL Aggregate Syntax:

```
SELECT columns,
    aggregate_fn(column)
FROM table
WHERE condition
GROUP BY columns
ORDER BY column ASC/DESC
```

We can also sort by any number of columns. In the group by and sort by, columns can be referenced by name OR index in the select statement

ex:
How many items do we have in each department sorted by most to fewest items?

```
SELECT department,
    COUNT(*) as items
FROM items
GROUP BY 1 [or department]
ORDER BY 2 [or items] desc
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SQL Aggregate Syntax:

```
SELECT columns,
    aggregate_fn(column)
FROM table
WHERE condition
GROUP BY columns
HAVING condition
```

The ‘Having‘ clause acts as a ‘Where’ clause for your aggregate columns

ex:
Pull any order that cost at least $1000 sorted by order revenue descending.

```
SELECT order_id,
     SUM(revenue) as rev
FROM sales
GROUP BY 1
HAVING SUM(revenue)>=1000
     or HAVING rev>=1000
ORDER BY 2 desc
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

SQL Column Functions:

CASE WHEN * THEN * [WHEN * THEN * ELSE *] END

An IF/THEN statement for SQL

CAST(column AS dtype)

Changes a column’s datatype (int64, string, float64 are the most common dtypes)

UPPER() and LOWER()

Adjusts the case of a string field for easier string matching

LIKE ‘%string%’

To match on ‘string’ with % acting as a wildcard (this is actually a conditional, not a function)

Ex: [REVISITED]
How much revenue did we generate in September 2022 and how many sales did we have?

```
SELECT SUM(revenue) as rev,
COUNT(distinct order_id) as
sales
FROM sales
WHERE CAST(date as string)
LIKE ‘09-%-2022’
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

SQL Column Functions:

CASE WHEN * THEN *
[WHEN * THEN * ELSE *] END

An IF/THEN statement for SQL

CAST(column AS dtype)

Changes a column’s datatype (int64, string, float64 are the most common dtypes)

UPPER() and LOWER()

Adjusts the case of a string field for easier string matching

LIKE ‘%string%’

To match on ‘string’ with % acting as a wildcard (this is actually a conditional, not a function)

Ex: [NEW]
What was our average order value in 2021?

```
SELECT SUM(revenue)/
COUNT(distinct order_id) as
rev_per_order
FROM sales
WHERE CAST(date as string)
LIKE ‘%-2021’
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Joins

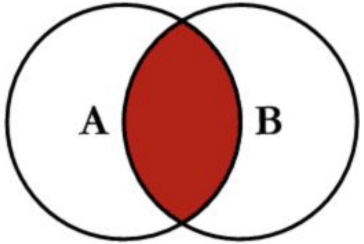
Joins:

```
SELECT columns
FROM table1 AS A
[JOIN] table2 AS B
ON A.key = B.key
```

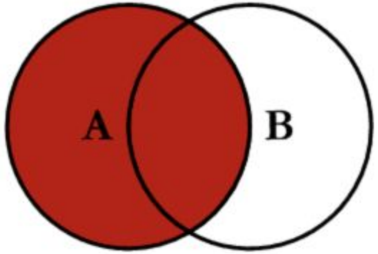
By joining on a key in common across two tables we can get information from multiple sources to tell a better story of our data.

There are many types of joins but the four fundamental ones are shown to the right.

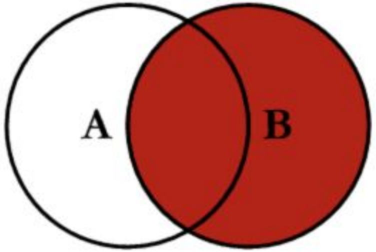
INNER JOIN



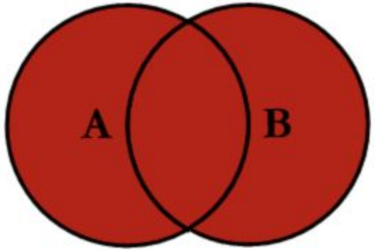
LEFT JOIN



RIGHT JOIN

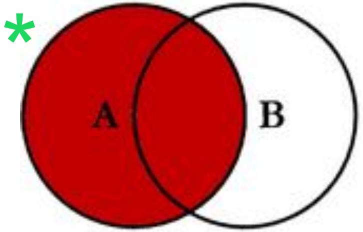


FULL JOIN

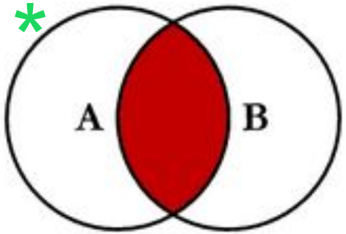


SQL JOINS

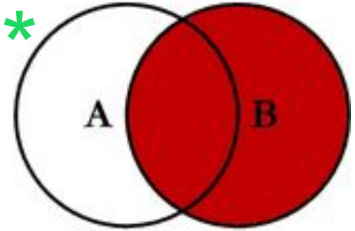
and when to use them!



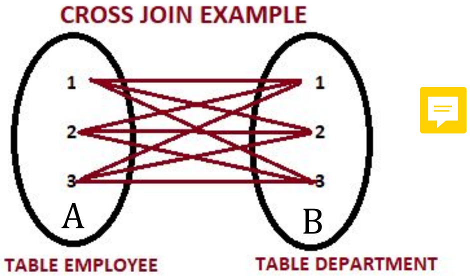
Used where you want all the entries in A and their corresponding data from B.



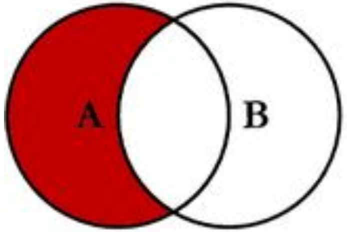
Used when you want all the entries in A that also appear in B.



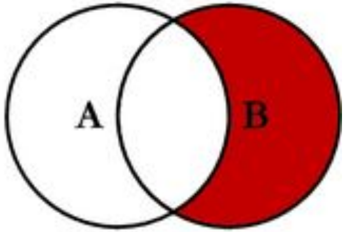
Used when you want all the entries in B and their corresponding data from A.



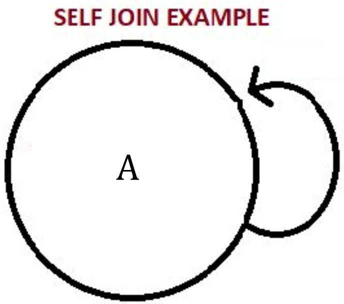
Used when you need to create a combination of every row from two tables. A common use for a cross join is to create obtain all combinations of items, such as colors and sizes. [RARELY USED]



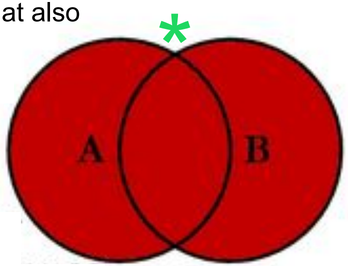
Used where you want all the entries in A except for any entries that also appear in B.



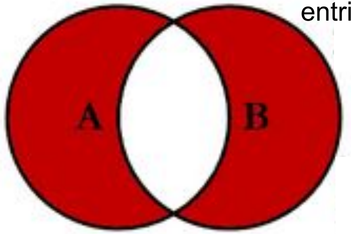
Used when you want all the entries in B except for any entries that also appear in A.



Used where there is any relationship between rows stored in the same table you’d like to understand. An Employee table may have a ManagerID column that points to the employee that is the boss of that employee. Self joins get information for both people in one row.

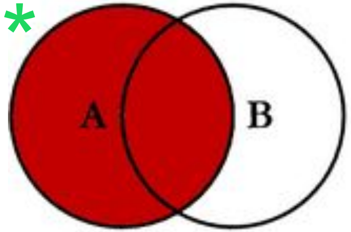


Used when you want all the entries in A AND all the entries in B with shared data appearing in the same row.

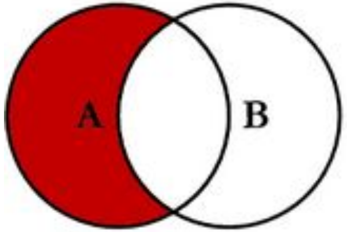


Used when you want all the entries in A AND all the entries in B except for data that appear in both.

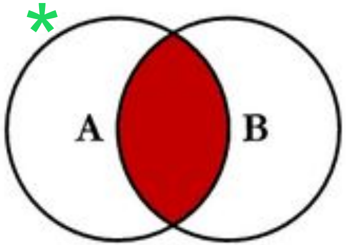
SQL JOINS



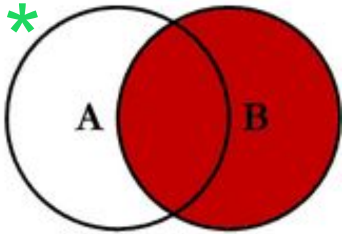
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



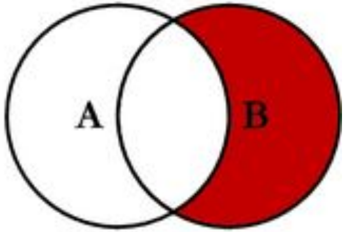
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



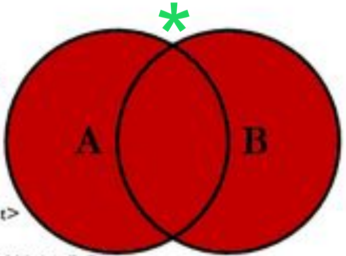
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



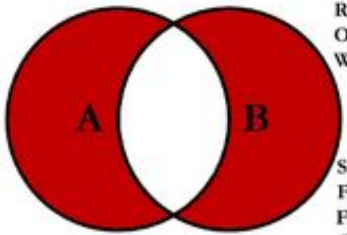
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



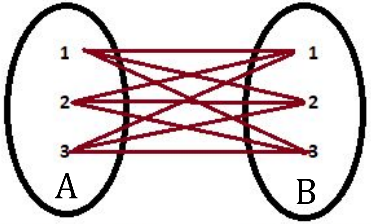
```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

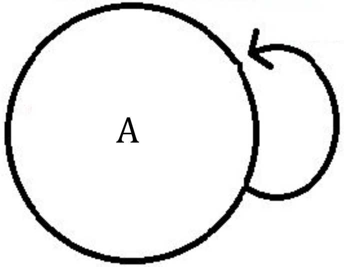
© C.L. Moffatt, 2008

CROSS JOIN EXAMPLE



```
Select <select list>
FROM TableA A
CROSS JOIN TableB B
```

SELF JOIN EXAMPLE



```
Select <select list>
FROM TableA A1
INNER JOIN TableA A2
ON A1.Key = A2.Key
```

Joins:

SELECT columns
FROM table1 **AS** A
JOIN* table2 **AS** B
ON A.key = B.key

The join key should be specified using its column name in each table.

You can join on several keys by using AND A.key2=B.key2, etc.

Ex:
How much revenue has every item we sell generated?

```
SELECT i.item_id,
SUM(s.revenue) as revenue
FROM items as i
LEFT JOIN sales as s
ON i.item_id = s.item_id
GROUP BY 1
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Another very important column function:

COALESCE(column, new_value [, ...])

Convert a null to a new value (of the same dtype) if ‘column’ is null

Ex:
How much revenue has every item we sell generated?

```
SELECT i.item_id,
COALESCE(SUM(s.revenue), 0)
as revenue
FROM items as i
LEFT JOIN sales as s
ON i.item_id = s.item_id
GROUP BY 1
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Subqueries and CTEs

Subqueries:

```
SELECT columns
FROM table
WHERE column_val [<,>,IN, etc.]
    (SELECT …)
```

Subqueries are SQL queries that are nested inside a larger query. They can be used in the SELECT, FROM, WHERE and/or HAVING statements.

Typically when using a subquery in the SELECT, WHERE or HAVING statements, the subquery must only return one value.

Ex: [HAVING/WHERE]
Pull the sales that generated more revenue than order ‘1234’.

```
SELECT order_id,
     SUM(revenue) as rev
FROM sales
GROUP BY 1
HAVING rev > (
     SELECT SUM(revenue)
     FROM sales
     WHERE order_id = ‘1234’)
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Subqueries:

SELECT columns, **(SELECT …)**
as new_column
FROM table

Subqueries are SQL queries that are nested inside a larger query. They can be used in the SELECT, FROM, WHERE and/or HAVING statements.

Typically when using a subquery in the SELECT, WHERE or HAVING statements, the subquery must only return one value.

Ex: [SELECT- Least Common]
Pull the sales that generated more revenue than order ‘1234’ including how much additional revenue they generated.

```
SELECT order_id,
  SUM(revenue) as rev,
  SUM(revenue) - (
     SELECT SUM(revenue)
     FROM sales
     WHERE order_id = ‘1234’)
     as additional_rev
FROM sales
GROUP BY 1
HAVING rev > (
     SELECT SUM(revenue)
     FROM sales
     WHERE order_id = ‘1234’)
```

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Subqueries:

SELECT columns
FROM (SELECT ...) as table1

Subqueries are SQL queries that are nested inside a larger query. They can be used in the SELECT, FROM, WHERE and/or HAVING statements.

Ex: [FROM]
Pull the sales that generated more than $100 and the last name of their respective customers.

```
SELECT r.order_id, r.rev,
c.last_name
FROM (SELECT order_id,
     customer_id,
     SUM(revenue) as rev
     FROM sales
     GROUP BY 1, 2
     HAVING rev > 100) as r
INNER JOIN customers as c
ON c.customer_id =
     r.customer_id
```

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Common Table Expressions:

```
WITH table1 as (
    SELECT columns
    FROM table) [, table2 as ()]
SELECT columns
FROM table1
```

CTEs and subqueries do essentially the same thing but CTEs are contained outside your “main” query and can be used several times throughout your query.

When you need to use more than one subquery (two nested subqueries or more), CTEs are a much cleaner approach.

Ex: [REVISITED]
Return the sales that generated more than $100 and the last name of their respective customers.

```
WITH order_rev as (
    SELECT order_id,
           customer_id
           SUM(revenue) as rev
    FROM sales
    GROUP BY 1, 2
    HAVING SUM(revenue)>100)
SELECT r.order_id, r.rev,
c.last_name
FROM order_rev r
INNER JOIN customers c
ON c.customer_id =
    r.customer_id
```

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Common Mistakes

Syntax order

SQL cares the order in which you write clauses

SELECT

FROM

JOIN [+ON]

WHERE

GROUP BY

HAVING

ORDER BY

LIMIT

Group By aggregations

Remember to always group by every column you aren’t aggregating

Ex:
How much revenue has every item we sell generated by day?

```
SELECT s.date, i.item_id,
COALESCE(SUM(s.revenue), 0)
as revenue
FROM items as i
LEFT JOIN sales as s
ON i.item_id = s.item_id
GROUP BY 1, 2
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Be careful with null values

It isn’t always appropriate to coalesce NULL values, so be careful with the problem you’ve been given

Clarification:
We typically only coalesce numerical values when we want to capture all entries including those with no ‘value’.

This is very handy when we want to calculate the average number of songs streamed in a week by all users with a Spotify account.

Users without any streams will have no rows in the streams table but we want to consider their 0 values. We probably wouldn’t coalesce their registration date or country.

Be careful with null values

When using a conditional for null values, you cannot use ‘=’ and MUST use ‘IS NULL’

Ex:
How many of our customers are missing addresses?

```
SELECT COUNT(customer_id) as
customer_count
FROM customers
WHERE address IS NULL
```

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Use distinct at the right times

Use distinct when values might be duplicated across multiple rows.

Don’t use distinct on a table’s key- it isn’t necessary to dedupe a key that is unique.

Ex: [distinct]
How many orders have been placed?

```
SELECT COUNT(distinct
order_id) as orders
FROM sales
```

Ex: [no distinct]
How many items do we stock?

```
SELECT COUNT(item_id) as
items
FROM items
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Complicated joins

Know which join to use

Often you will have to join tables on multiple keys

Clarification:
We have to join tables on multiple keys where there isn't a unique identifier for every row in both tables.

In our tables, the sales table has a 'composite primary key'; order_id and item_id. Let's say we had another table called 'INVENTORY' with inventory_id, order_id and item_id to identify when each individual item had sold.

We would have to join these tables with:

```
ON t1.order_id = t2.order_id
AND t1.item_id = t2.item_id
```

The easiest one to fix

Make sure you’re talking about your code while you write it!

Interviewers want to know that you understand what you’re doing and this often helps show that.

Practice Questions

Question 1:

Pull the total number of orders completed on 09-15-2022 (just the value)

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 1:

```
SELECT COUNT(distinct order_id) as orders
FROM sales
WHERE date = ‘09-15-2022’
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 2:

Pull the total number of orders completed on 09-15-2022 to customers with the first name ‘Lucy’ or ‘John’

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 2:

```
SELECT COUNT(distinct s.order_id) as orders
FROM sales s
INNER JOIN customers c
ON c.customer_id = s.customer_id
WHERE s.date = ‘09-15-2022’
AND c.first_name IN (‘Lucy’, ‘John’)
```

Pro Tip!:

Text matching is best when the capitalization is controlled for!
Using the lower() function, a better last line would be:

```
AND lower(c.first_name) IN (‘john’, ‘lucy’)
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 3:

Pull the total number of customers that purchased items in September 2022 and the average amount spent (in September) per customer

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 3:

```
SELECT COUNT(distinct customer_id) as customers,
   SUM(revenue)/COUNT(distinct customer_id) as avg_spend
FROM sales
WHERE CAST(s.date as string) LIKE ‘09-%-2022’
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 4:

Which departments generated less than $500 revenue in September 2022?

Print the departments and their September revenue

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 4:

```
SELECT department, coalesce(SUM(revenue), 0) as rev
FROM sales s
RIGHT JOIN items i
ON i.item_id = s.item_id
WHERE CAST(date as string) LIKE ‘09-%-2022’
GROUP BY 1
HAVING rev<500
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 5:

What is the most revenue we have generated from a single order?

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 5:

```
WITH order_rev as (
        SELECT order_id, SUM(revenue) as rev
        FROM sales
        GROUP BY 1)
SELECT MAX(rev) as max_rev
FROM order_rev
```

OR:

```
SELECT order_id, SUM(revenue) as rev
FROM sales
GROUP BY 1
ORDER BY 2 desc
LIMIT 1
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 6:

What items were purchased in our most lucrative (highest revenue) order?

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 6:

```
WITH order_rev as (
        SELECT order_id, SUM(revenue) as rev
        FROM sales
        GROUP BY 1)
SELECT s.item_id
FROM sales s
JOIN order_rev o
ON o.order_id = s.order_id
WHERE o.rev = (
        SELECT MAX(rev) as max_rev
        FROM order_rev)
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 6 - alternative:

```
WITH order_rev as (
        SELECT order_id, SUM(revenue) as rev
        FROM sales
        GROUP BY 1
        ORDER BY 2 desc
        LIMIT 1)
SELECT s.item_id
FROM sales s
JOIN order_rev o
ON o.order_id = s.order_id
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 7:

What items are commonly purchased by our biggest spenders? Pull the top 15 items (by quantity) purchased by our 10 biggest spenders.

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 7:

```
WITH big_spenders as (
    SELECT customer_id, SUM(revenue) as rev
    FROM sales
    GROUP BY 1
    ORDER BY 2 desc
    LIMIT 10)
SELECT s.item_id, SUM(quantity) as total_orders
FROM sales s
JOIN big_spenders b
ON b.customer_id = s.customer_id
GROUP BY 1
ORDER BY 2 desc
LIMIT 15
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 8:

Make a list of 10 of our top seasonal items for December

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

```
WITH december as (
        SELECT item_id, COUNT(distinct order_id) as orders
        FROM sales
        WHERE CAST(date as string) like ‘12-%’
        GROUP BY 1
        ORDER BY 2 desc
        LIMIT 100),
rest_of_year as (
        SELECT item_id, COUNT(distinct order_id) as orders
        FROM sales
        WHERE CAST(date as string) not like ‘12-%’
        GROUP BY 1
        ORDER BY 2 desc
        LIMIT 100),
SELECT item_id
FROM december
WHERE item_id NOT IN (SELECT item_id FROM year_round)
ORDER BY orders desc
LIMIT 10
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 9:

What items are often bought together? Return the top 20 pairs.

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 9:

```
WITH item_pairs as (
      SELECT s1.order_id, s1.item_id as item1, s2.item_id as item2
      FROM sales s1
      JOIN sales s2
      ON s2.order_id = s1.order_id
      WHERE s1.item_id != s2.item_id)
SELECT item1, item2, COUNT(distinct order_id) as orders
FROM item_pairs
WHERE                 #to remove duplicated pairs
 CONCATENATE(item1, item2) != CONCATENATE(item2, item1)
GROUP BY 1, 2
ORDER BY 3 desc
LIMIT 20
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Question 10:

Recommend customer ‘1234’ a set of 10 items they might be interested in.

Don’t recommend any items they have already purchased.

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

```
WITH cust_items as (
        SELECT distinct item_id
        FROM sales
        WHERE customer_id = ‘1234’
), all_orders as (
        SELECT distinct order_id
        FROM sales
        WHERE item_id IN (SELECT * FROM cust_items) )
SELECT s.item, COUNT(distinct s.order_id) as orders
FROM sales s
JOIN all_orders o
ON s.order_id = o.order_id
WHERE s.item NOT IN (SELECT * FROM cust_items)
GROUP BY 1
ORDER BY 2 desc
LIMIT 10
```

Tables:

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

CUSTOMERS columns: **customer_id**, **first_name**, **last_name**, **address**

Bonus Section:
Window Functions

Window Functions Overview

A window function performs a calculation across a set of table rows. This is comparable to the type of calculation that can be done with an aggregate function, but unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row.

Clauses

Window Function Syntax:

```
SELECT columns,
    window_fn(args) OVER()
FROM table ...
```

This is the basic window function that windows over all rows and returns a function value across all rows.

Here, the AVG() function is essentially running an AVG() aggregate function on the entire column of ‘price’ and adding that result to every row.

ex:
Return all items that cost at least $50 and the average price for that set of items.

```
SELECT item_id, price,
    AVG(price) OVER() as
avg_price
FROM items
WHERE price >= 50
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

Partition By Clause:

```
SELECT columns,
    window_fn(args) OVER(
    [PARTITION BY col1, col2, ...]
    )
FROM table ...
```

PARTITION BY acts as the ‘grouped by’ element in a window function. When we partition by a column we get the aggregated value applied to all related fields in the grouped by element.

ex:
Return all items that cost at least $50 and the number of items in that item’s department that are also $50+.

```
SELECT department, item_id,
price, COUNT() OVER(PARTITION
BY department) as
items_in_dept
FROM items
WHERE price >= 50
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

Order By Clause:

```
SELECT columns,
    window_fn(args) OVER(
    [PARTITION BY col1, col2, ...]
    [ORDER BY col1, col2, ...]
    )
FROM table ...
```

The ORDER BY clause is used in ranking-based window functions to establish the column by which to rank

ex:
Return any item that costs at least $50 and the rank of that item's value compared to all other items that cost at least $50.

```
SELECT item_id, price,
RANK(price) OVER(ORDER BY
price desc) as price_rank
FROM items
WHERE price >= 50
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

Functions

Window Aggregate Functions:

COUNT(column) OVER()

Counts all non null rows in a column

SUM(column) OVER() and AVG(column) OVER()

Calculates the sum or average all values of a column

MIN(column) OVER() and MAX(column) OVER()

Calculates the minimum or maximum value of a column

Window Aggregate Functions:

```
SELECT columns,
    window_fn(args) OVER(
    [PARTITION BY col1, col2, ...]
    )
FROM table ...
```

The window aggregate functions are used when you need aggregate values across all related rows.

ex:
Return the total revenue generated by items in each department as well as the percent of all that departments revenue generated by each item.

```
SELECT department, item_id,
rev, rev/SUM(rev)
OVER(PARTITION BY department)
as pct_dept_rev
FROM items i
JOIN (SELECT item_id,
       SUM(revenue) as rev
     FROM sales GROUP BY 1) s
ON i.item_id = s.item_id
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**

Window Ordering Functions:

RANK() OVER(order by column**)**

Ranks all non null rows in a column from 1 to the total number of items

PERCENT_RANK() OVER(order by column**)**

Returns the percentile rank (rank/max rank) for each row (value ranges from 0 to 1)

ROW_NUMBER() OVER(order by column**)**

Assign a sequential number to each row and rank rows with no ties

NTILE(#**) OVER(**order by column**)**

Breaks rows into a specified number of buckets (if #=10 you’re looking at the deciles of your column)

Ranking Window Functions:

```
SELECT columns,
    window_fn(args) OVER(
    [PARTITION BY col1, col2, ...]
    ORDER BY col1, col2, ...)
FROM table ...
```

Here the ORDER BY is required, and it dictates the direction and the column to sort by (default is ascending).

For our example we use a new function, ROUND() which takes a number and a number of decimal places to round the number to.

ex:
Return our median order revenue.

```
WITH orders as (
     SELECT order_id,
            SUM(revenue) as rev
     FROM sales
     GROUP BY 1),
ranked_orders as (
     SELECT order_id, rev,
     ROW_NUMBER() OVER(ORDER BY
     rev) as rownum
     FROM orders)
SELECT rev
FROM ranked_orders
WHERE rownum = ROUND((SELECT
     MAX(rownum) FROM
     ranked_orders)/2, 0)
```

ITEMS columns: **item_id**, **item_name**, **price**, **department**

SALES columns: **date**, **order_id** , **item_id**, **customer_id**, **quantity**, **revenue**