How to Simplify Complex SQL Queries (With Examples)

In this article, we'll discuss various ways to simplify complex SQL queries and apply these tips to answer difficult SQL interview questions.

As a data scientist, you should always aim to write simple and efficient SQL queries that perfectly address the problem and handle edge cases. From time to time, however, you'll have to write complex SQL queries to handle difficult tasks.

The problem is that sometimes queries are more complex than they need to be.

As you accumulate experience writing SQL queries, you'll be able to simplify them using the right SQL features. This article will explain how to simplify complex SQL queries to help you learn faster.

What Is a Complex SQL query?

They are difficult to characterize, but you'll know one when you see one.

Complex SQL queries typically involve a fair bit of logic and advanced SQL features like transforming data, working with subqueries, aggregation, analytics, filtering, joining tables, or all of these features combined. Usually, the code looks unintelligible, and its logic is difficult to follow.

For example, nested subqueries can be an indicator of a complex query. These are subqueries that have more subqueries in them. CTEs allow you to store the result of each subquery. So instead of having subqueries in the main query, you can simply reference their result.

CTEs are more readable than subqueries and also easier to test and build upon. Our blog post about SQL query interview questions contains plenty of examples where we use CTEs to simplify the query.

How to Simplify a Complex SQL Query

Here are a few ways to simplify a complex SQL query:

1. Replace complex subqueries with CTEs.
2. Find a better SQL feature (or combination of features) to accomplish the same in fewer lines of code. For this, you need to have a deep knowledge of SQL features and their use cases. Later we'll show an example of how you can eliminate a nested subquery by using DISTINCT instead of GROUP BY.
3. Use more readable syntax. For example, shorthand for casting values or using numbers to specify columns for GROUP BY and ORDER BY statements. These are small improvements but can sometimes make a difference.
4. Last but not least, formatting, indentation, and comments. These are not technically part of the query itself but can make a difference.

Let's take a closer look at each of these features and how you can use them to simplify SQL queries.

SELECT and FROM

For starters, make sure only to SELECT the values you need. This will add clarity to your SQL query. It'll also improve performance by reducing the volume of data you're working on.

If your SELECT statement contains an aggregate function, use the AS keyword to give it a descriptive name.

The FROM clause specifies the dataset for selecting values. If there are multiple tables, try to establish relationships between them and decide which ones you'll need to JOIN.

Creating a subquery

Subqueries are useful when you want to work with the result of a query. If the subquery is simple and easy to understand, it may be better to leave it as it is. If it's complex, use CTEs instead.

Subqueries tend to become unintelligible when nested or contain too many calculation steps. Later in the article, we'll replace difficult subqueries with CTEs.

CTEs also allow you to save the result of a subquery and reference the result as many times as you want. This is useful if your main query references the same subquery multiple times.

Common Table Expressions (CTEs)

As we already mentioned, we can use CTEs to work with the result of a query without having it embedded in the main query.

Storing the result of a subquery in a CTE also makes it easy to debug. You can run the subquery, preview its output, fix possible errors and make necessary changes to achieve the desired result.

We use the WITH clause and AS command to create a common table expression:

WITH dataset_name AS (subquery)

Choose a descriptive name for the output of the subquery, so the dataset and its values are easy to understand.

Aliases – Using the AS Keyword

You can use the **AS** command to name values, tables, and datasets. This will help you understand datasets, values, and how they fit in your query.

For example, if you create a value using three different functions, giving it a descriptive name can save you and your collaborators some time in understanding this value and its purpose.

If tables in the FROM clause have long names, use the AS keyword to give them aliases and avoid writing long table names.

JOINs

JOINs are a ubiquitous SQL feature. Understanding different types of JOINs and their use cases can help you simplify a complex SQL query. Sometimes you can use the ON clause to do all the filtering and eliminate the need for an additional WHERE clause.

Depending on the specifics of the task, using the right type of JOIN can save you multiple lines of code.

Setting conditions

Complex SQL queries tend to have very specific and complex conditions.

In addition to commonly used AND and OR logical operators, SQL has other useful operators like NOT, IN, and EXISTS.

You can use the NOT operator to specify the list of unacceptable values. IN is the opposite. It allows you to specify a list of acceptable options, so you won't have to use the OR logical operator to chain multiple conditions.

GROUP BY and ORDER BY:

You can simplify your SQL query by using numbers to specify columns for GROUP BY and ORDER BY statements.

This tip is really useful if GROUP BY or ORDER BY statements specify multiple columns or if column names are too long or difficult to write.

Aggregate functions

Aggregate functions can be used to eliminate a lot of unnecessary code.

You are probably aware of the five most popular aggregate functions - **AVG()**, **SUM()**, **COUNT()**, **MIN()**, and **MAX()**, but there are many more.

Being aware of 'unpopular' aggregate functions can help you write a simple query to solve a difficult challenge. Sometimes interviewers ask you these questions to measure your ability to choose the right SQL feature for any given task. Read our SQL Advanced Interview Questions blog for more examples.

Casting values

You can simplify casting operations by using a shorthand syntax (::) in or using the implicit conversions. Later we'll show you an example where, instead of explicitly casting an integer value to a float, we multiply it by 1.0 to achieve the same result.

Better formatting

SQL does not provide guidelines on formatting. Technically you can write complex queries on a single line, but it will be a nightmare to understand.

Poorly formatted SQL queries tend to look more complex than they actually are. Using white spaces, indentation, and new lines can improve the readability of the code.

It's also important to implement consistent casing. You can write SQL features like the SELECT statement in uppercase or lowercase. Usually, most data scientists use upper case, but you can also do lower case and even title case.

Whatever you decide, be consistent throughout the query. For example, don't write a SELECT statement in uppercase and the 'from' clause in lowercase.

Comments

Technically, comments do not simplify complex SQL queries, but they help you, and your collaborators better understand the query.

It's best to keep comments short. One or two sentences are usually enough to explain confusing parts of code.

In SQL, comments are marked using two hyphens '--'.

Real-World Complex SQL Queries Examples

Let's try to simplify answers to five difficult questions on the StrataScratch platform. Once you're finished reading, go through our list of SQL Interview Questions and try to simplify them yourself.

Complex SQL Queries Example #1 - Distance Per Dollar

Distance Per Dollar

Interview Question Date: November 2020

Uber Hard Interview Questions ID 10302

You're given a dataset of uber rides with the traveling distance ('distance_to_travel') and cost ('monetary_cost') for each ride. For each date, find the difference between the distance-per-dollar for that date and the average distance-per-dollar for that year-month. Distance-per-dollar is defined as the distance traveled divided by the cost of the ride.

The output should include the year-month (YYYY-MM) and the absolute average difference in distance-per-dollar (Absolute value to be rounded to the 2nd decimal).
You should also count both success and failed request_status as the distance and cost values are populated for all ride requests. Also, assume that all dates are unique in the dataset. Order your results by earliest request date first.

Table: uber_request_logs

Link to the question: https://platform.stratascratch.com/coding/10302-distance-per-dollar

Available data:
To solve this question, candidates have to work with the **uber_request_logs** table, which contains information about all requests.

request_id	request_date	request_status	distance_to_travel	monetary_cost	driver_to_client_distance
1	2020-01-09 00:00:00	success	70.59	6.56	14.36
2	2020-01-24 00:00:00	success	93.36	22.68	19.9
3	2020-02-08 00:00:00	fail	51.24	11.39	21.32
4	2020-02-23 00:00:00	success	61.58	8.04	44.26

request_id:	int64
request_date:	datetime64[ns]
request_status:	object
distance_to_travel:	float64
monetary_cost:	float64
driver_to_client_distance:	float64

- We will need to work with **request_date** values to find the absolute average distance per dollar for each month.
- The question itself specifies that we need to use **distance_to_travel** and **monetary_cost** values to calculate the mean deviation.

How to Simplify This Complex SQL Query

First, let's look at a correct but not simple solution and try to simplify it.

The most complex part is the query that contains another subquery. It divides the result of one window function by another to calculate the average distance per dollar for each month. Also, we use the **TO_CHAR()** function to convert date values, which we cast to date. Instead of using the double-semicolon shorthand syntax, this solution uses the **CAST()** function.

The query uses the AS command, but the specified names are too vague. For example, we divide the results of analytical functions, but the result is named **division_result_one**, not a descriptive name.

```
SELECT request_mnth,
       round(avg(result), 2) AS number
FROM
  (SELECT request_mnth,
          CAST(abs(division_result_one-division_result_two) AS decimal) AS result
   FROM
```

```
     (SELECT to_char(CAST(request_date AS date), 'YYYY-MM') AS request_mnth,
             distance_to_travel/monetary_cost AS division_result_one,
             sum(distance_to_travel) OVER (PARTITION BY to_char(CAST(request_date AS date),
'YYYY-MM')) / sum(monetary_cost) OVER (PARTITION BY to_char(CAST(request_date AS date),
'YYYY-MM')) AS division_result_two
      FROM uber_request_logs) a) b
GROUP BY request_mnth
```

This is definitely a complex query. The logic is difficult to follow, but nested queries add to the complexity of the query. To simplify the query, we need to save the results of subqueries as CTEs. So instead of embedding them in the main query, we'll be able to reference their result.

This doesn't always reduce the amount of code, but it will make the query much easier to follow.

This solution from the 'Solution Discussions' thread utilizes a combination of comments, and the AS keyword, replacing subqueries with CTEs to simplify this query and giving more descriptive aliases to the column. Additionally, we changed the capitalization so the SQL keywords are written in uppercase.

```
-- Create distance-per-dollar column
WITH added_distance_per_dollar AS
  (SELECT request_id,
          request_date,
          request_status,
          (distance_to_travel/monetary_cost) AS distance_per_dollar
   FROM uber_request_logs) -- Create year-month column
,
     added_year_month AS
  (SELECT *,
          TO_CHAR(request_date, 'YYYY-MM') AS year_month
   FROM added_distance_per_dollar) -- Adding average distance-per-dollar per year_month
,
     added_year_month_avg AS
  (SELECT *,
          AVG(distance_per_dollar) OVER(PARTITION BY year_month) AS year_month_avg
   FROM added_year_month) -- Adding difference between distance_per_dollar and
year_month_avg
,
     added_difference AS
  (SELECT *,
          (distance_per_dollar-year_month_avg) AS difference
   FROM added_year_month_avg) -- Adding absolute_average_difference
,
     absolute_average_difference AS
  (SELECT request_id,
          request_date,
```

```
          request_status,
          year_month,
          ROUND(ABS(difference)::NUMERIC, 2) AS absolute_average_difference
   FROM added_difference) -- Final solution

SELECT DISTINCT(year_month),
       absolute_average_difference
FROM absolute_average_difference
```

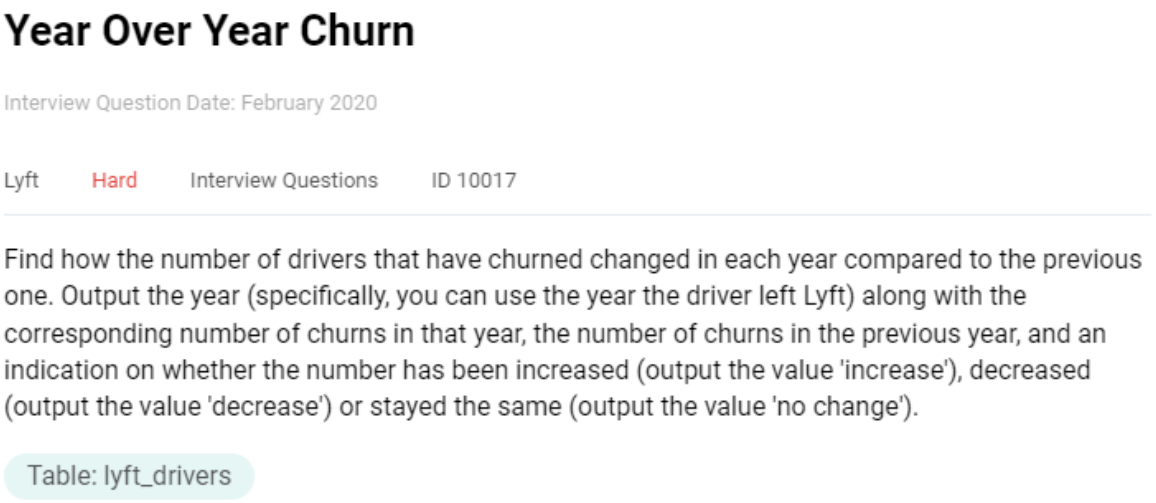
This way, the entire logic of the query is spread across five CTEs. The code may be longer, but it's much simpler and easy to understand.

The main query is reduced to just three lines. There may be more overall lines of code, but CTEs are easier to understand.

Also, instead of using the CAST() functions, we used shorthand to convert data types.

You can run both codes and see they return the same output.

Complex SQL Queries Example #2 - Year Over Year Churn



Link to the question: https://platform.stratascratch.com/coding/10017-year-over-year-churn

Available data:
We are given the **lyft_drivers** table, which contains information about Lyft drivers.

index	start_date	end_date	yearly_salary
0	2018-04-02 00:00:00		48303
1	2018-05-30 00:00:00		67973
2	2015-04-05 00:00:00		56685
3	2015-01-08 00:00:00		51320
4	2017-03-09 00:00:00		67507

index:	int64
start_date:	datetime64[ns]
end_date:	datetime64[ns]
yearly_salary:	int64

- We'll have to work with **end_date** values to find the number of drivers who left the company in a specific year and compare that number with the previous year.

How to Simplify This Complex SQL Query

Like the previous example, we could solve this question by using a complex subquery with a nested subquery of its own.

```
SELECT *,
       CASE
           WHEN n_churned > n_churned_prev THEN 'increase'
           WHEN n_churned < n_churned_prev THEN 'decrease'
           ELSE 'no change'
       END
FROM
  (SELECT year_driver_churned,
          COUNT(*) AS n_churned,
          LAG(COUNT(*), 1, '0') OVER (ORDER BY year_driver_churned) AS n_churned_prev
   FROM
     (SELECT date_part('year', CAST (end_date AS date)) AS year_driver_churned
      FROM lyft_drivers
      WHERE end_date IS NOT NULL) base
   GROUP BY year_driver_churned
   ORDER BY year_driver_churned ASC) calc
```

To simplify this query, we're going to move subqueries out of the main query and write them as CTEs. We will also separate logic into two different CTEs, making both datasets easier to understand.

First, we'll define a CTE that extracts the year from every **end_date** value. If the driver is still working with Lyft, the **end_date** column will be empty. So we need to SELECT years from **end_date** values that are not NULL.

As usual, we use WITH and AS keywords to create a common table expression (CTE). We also use the AS keyword to give the year value a descriptive name - **year_driver_churned**. This creates a list of all drivers who quit Lyft.

The second CTE is based on the result of the first one. In it, we use **COUNT()** and **GROUP BY** to find the number of churned drivers for each year. Then we use the **LAG()** window function to generate the number of churned drivers in the previous year. Finally, we order rows by year. Everything is simple.

To make the query look cleaner, we use numbers to specify columns for GROUP BY and ORDER BY statements.

Once subqueries are out of the main query, it becomes a basic SELECT statement with a CASE expression and a simple ORDER BY statement.

In the previous code, we used the CAST() function to convert one column into a DATE data type. It's much easier to use '::' as its shorthand, shown in this example.

Run the codes in the widgets to see their output. Also, feel free to play around with it and find some other ways to simplify these codes.

```
WITH drivers_churned AS
  (SELECT date_part('year', end_date::DATE) AS year_driver_churned
   FROM lyft_drivers
   WHERE end_date IS NOT NULL),

churned_current_and_previous_year AS
  (SELECT year_driver_churned,
          COUNT(*) AS n_churned,
          LAG(COUNT(*), 1, '0') OVER (ORDER BY year_driver_churned) AS n_churned_prev
   FROM drivers_churned
   GROUP BY 1
   ORDER BY 1 ASC)

SELECT *,
       CASE
           WHEN n_churned > n_churned_prev THEN 'increase'
           WHEN n_churned < n_churned_prev THEN 'decrease'
```

```
            ELSE 'no change'
        END
FROM churned_current_and_previous_year
ORDER BY 1
```

Also, you can go an entirely different route and use the **COALESCE()** function to write a solution without any subqueries or even CTEs.

With a deep knowledge of SQL features, you can come up with various solutions to the same problem. Then you can choose the simplest one.

```
SELECT DATE_PART('year', end_date) AS year,
       COUNT(end_date) AS current,
       COALESCE(LAG(COUNT(end_date)) OVER (ORDER BY DATE_PART('year', end_date)),0) AS
previous,
       CASE
           WHEN COUNT(end_date) > COALESCE(LAG(COUNT(end_date)) OVER (ORDER BY
DATE_PART('year', end_date)),0) THEN 'Increase'
           WHEN COUNT(end_date) < COALESCE(LAG(COUNT(end_date)) OVER (ORDER BY
DATE_PART('year', end_date)),0) THEN 'Decrease'
           ELSE 'No Change'
       END AS change
FROM lyft_drivers
WHERE DATE_PART('year', end_date) IS NOT NULL
GROUP BY year
```

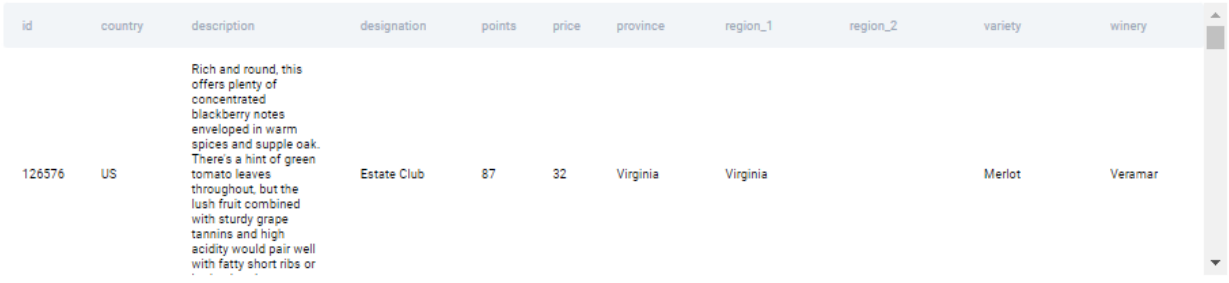
Complex SQL Queries Example #3 - Median Price of Wines

Link to the question: https://platform.stratascratch.com/coding/10043-median-price-of-wines

Available data

To answer this question, we must work with two tables: **winemag_p1** and **winemag_p2**.

winemag_p1

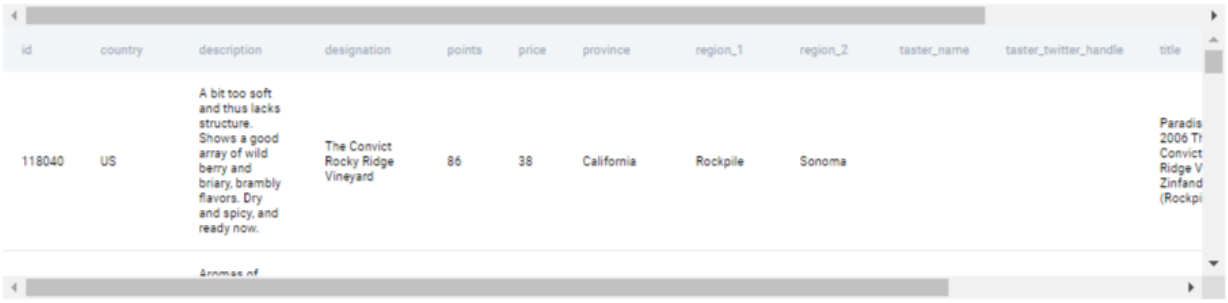


id	country	description	designation	points	price	province	region_1	region_2	variety	winery
126576	US	Rich and round, this offers plenty of concentrated blackberry notes enveloped in warm spices and supple oak. There's a hint of green tomato leaves throughout, but the lush fruit combined with sturdy grape tannins and high acidity would pair well with fatty short ribs or	Estate Club	87	32	Virginia	Virginia		Merlot	Veramar

id:	int64
country:	object
description:	object
designation:	object
points:	int64
price:	float64
province:	object
region_1:	object
region_2:	object
variety:	object
winery:	object

- We'll need to work with **price** and **variety** values to find the median price for each variety of wine.

winemag_p2



id	country	description	designation	points	price	province	region_1	region_2	taster_name	taster_twitter_handle	title
118040	US	A bit too soft and thus lacks structure. Shows a good array of wild berry and briary, brambly flavors. Dry and spicy, and ready now.	The Convict Rocky Ridge Vineyard	86	38	California	Rockpile	Sonoma			Paradis 2006 T Convict Ridge V Zinfand (Rockpi

id:	int64
country:	object
description:	object
designation:	object
points:	int64
price:	float64
province:	object
region_1:	object
region_2:	object
taster_name:	object
taster_twitter_handle:	object
title:	object
variety:	object
winery:	object

- The question tells us to find the median price for each variety of wine across both datasets. We'll work with **price** and **variety** values in this table as well.

How to Simplify This Complex SQL Query

This is a difficult question. It might seem like it requires a complex answer, but that's not the case. There's one SQL function that can help you write a simple answer.

As an inexperienced data scientist, you might not be aware of this function and take a different approach:

1. Use **UNION** to vertically combine variety and price values from two tables
2. Group wines by **'variety'**
3. Use **COUNT()** to find the number of wines in each group
4. Use **ROW_NUMBER()** to number wines in each partition. Also, order wines by their price in ascending order.
5. Somehow calculate the median for each group.

Doing all of these operations will result in a lot of code.

The median is calculated by using two SELECT statements to query the second CTE and unionize the two statements' results.

The query is correct and follows the logic of calculating the median. For wines with an even number of rows, the median is calculated by finding the average values from the two middle rows. If there's an odd number of rows, the median simply takes the value from the middle row.

```
WITH winemag AS
  (
    SELECT
        variety,
        price
    FROM
        winemag_p1
    UNION ALL
    SELECT
        variety,
        price
    FROM
        winemag_p2
    ),

ranked_wine AS
    (
    SELECT
        *,
```

```
        ROW_NUMBER() OVER(PARTITION BY variety ORDER BY price) AS n,
        COUNT(*) OVER(PARTITION BY variety) AS cnt
    FROM
        winemag
    WHERE
        price IS NOT NULL
        AND variety IS NOT NULL
        AND variety != ''
        AND variety != ''

    )

SELECT variety,
       AVG(price) AS median
FROM ranked_wine
WHERE cnt%2 = 0 AND n IN (cnt/2, cnt/2+1)
GROUP BY 1
UNION ALL
SELECT variety,
       price AS median
FROM ranked_wine
WHERE cnt%2 <> 0 AND n = cnt/2+1
GROUP BY variety, median
ORDER BY variety
```

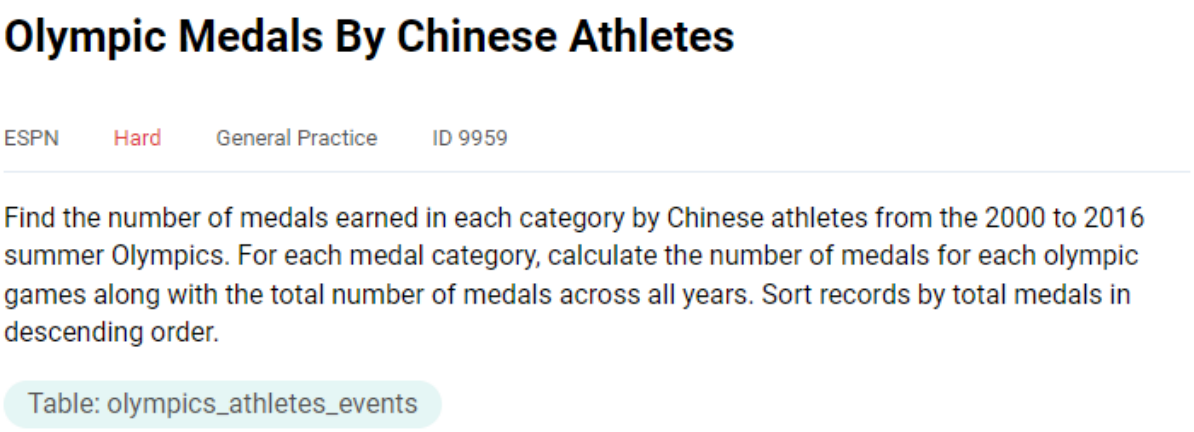
This could work, but it's a very roundabout way of calculating the median. There's a function for this task - **PERCENTILE_CONT()**. It takes one argument - the percentile to compute (a decimal between 0 and 1). Passing 0.5 as an argument will return a median price for each variety of wine in the list.

```
SELECT DISTINCT variety,
                PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY price) AS median_price
FROM
  (SELECT variety,
          price::FLOAT
   FROM winemag_p1
   UNION ALL SELECT variety,
                    price::FLOAT
   FROM winemag_p2) a
GROUP BY 1
ORDER BY 1
```

As you can see, using the **PERCENTILE_CONT()** function significantly reduces the amount of SQL code. You can read documentation to learn about useful SQL functions like this one.

This SQL query also uses shorthand syntax for converting data types. It also names results of aggregate functions and uses numbers to specify columns for GROUP BY and ORDER BY statements.

Complex SQL Queries Example 4: Olympic Medals By Chinese Athletes



Link to the question:
https://platform.stratascratch.com/coding/9959-olympic-medals-by-chinese-athletes

Available data
We are given access to the **olympics_athletes_events**, which contains information about athletes' performance on each Olympic event.

id	name	sex	age	height	weight	team	noc	games	year	season
3520	Guillermo J. Amparan	M				Mexico	MEX	1924 Summer	1924	Summer
35394	Henry John Finchett	M				Great Britain	GBR	1924 Summer	1924	Summer
21918	Georg Frederik Ahrensborg Clausen	M	28			Denmark	DEN	1924 Summer	1924	Summer

id:	int64
name:	object
sex:	object
age:	float64
height:	float64
weight:	datetime64[ns]
team:	object
noc:	object
games:	object
year:	int64
season:	object
city:	object
sport:	object
event:	object
medal:	object

- General overview of most important values in the dataset

How to Simplify This Complex SQL Query

Our initial solution is technically correct, but there's a lot of room for simplification.

```
SELECT
    base.medal,
    sum(CASE WHEN year = 2000 THEN n_medals ELSE 0 END),
    sum(CASE WHEN year = 2004 THEN n_medals ELSE 0 END),
    sum(CASE WHEN year = 2008 THEN n_medals ELSE 0 END),
    sum(CASE WHEN year = 2012 THEN n_medals ELSE 0 END),
    sum(CASE WHEN year = 2016 THEN n_medals ELSE 0 END),
    sum(n_medals)
FROM
    (SELECT
        year,
        medal,
```

```
        count(*) AS n_medals
    FROM
        olympics_athletes_events
    WHERE
        team = 'China' AND
        (year = 2000 OR year = 2004 OR year = 2008 OR year = 2012 OR year = 2016) AND
        medal IS NOT null
    GROUP BY
        year,
        medal) base
 GROUP BY
    base.medal
ORDER BY
    sum(n_medals) DESC
```

In the SELECT statement, we use the **SUM()** aggregate function to create six values. We do not use the AS command to give values a custom name, so the result of the **SUM()** function will be **sum**. To know which column shows what data, you would have to read the CASE statement each time. Previewing the table isn't going to help either because all columns will have the same name - **sum**.

Also, to know which column shows what data, you would have to read the CASE statement each time.

We can use the AS command to give columns a descriptive name. For example, the **SUM()** function that finds the number of medals won in the year 2000 should be named **medals_2000**.

The condition for the WHERE clause can be improved as well. We want to find Olympic records where the **year** value is either 2000, 2004, 2008, 2012, or 2016. Currently, we use the OR logical operator to set this condition. We can use the IN operator to make this expression much shorter.

We could further simplify the query by using numbers to specify columns for GROUP BY and ORDER BY statements. Since we didn't give an alias to a column, we don't have a choice but to use the SUM() function to calculate the number of medals. Redundancy in SQL should be avoided whenever possible.

Another issue is the poorly named subquery('base'). This can be resolved by moving the subquery outside the main query and writing it as CTE with a more descriptive name.

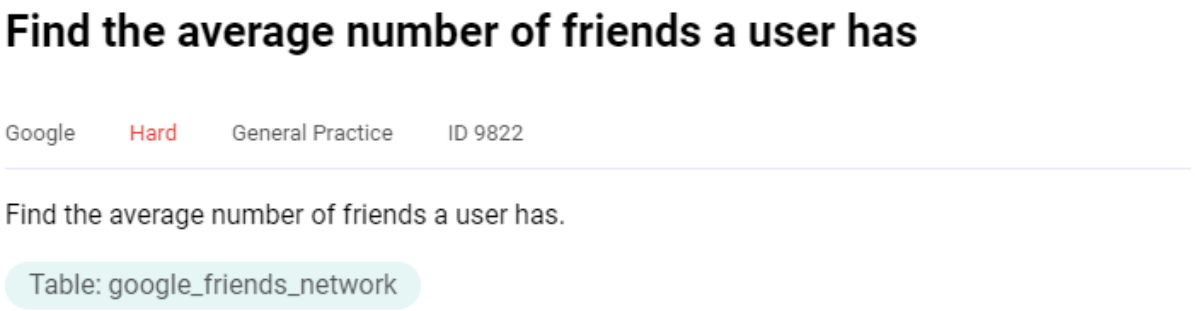
Also, use all uppercase letters for the SQL keywords to improve readability.

```
WITH types_of_medals_in_specific_years AS
  (SELECT YEAR,
          medal,
          COUNT(*) AS n_medals
```

```
   FROM olympics_athletes_events
   WHERE team = 'China'
     AND YEAR IN (2000, 2004, 2008, 2012, 2016)
     AND medal IS NOT NULL
   GROUP BY 1, 2)

SELECT medal,
       SUM(CASE WHEN YEAR = 2000 THEN n_medals ELSE 0 END) AS medals_2000,
       SUM(CASE WHEN YEAR = 2004 THEN n_medals ELSE 0 END) AS medals_2004,
       SUM(CASE WHEN YEAR = 2008 THEN n_medals ELSE 0 END) AS medals_2008,
       SUM(CASE WHEN YEAR = 2012 THEN n_medals ELSE 0 END) AS medals_2012,
       SUM(CASE WHEN YEAR = 2016 THEN n_medals ELSE 0 END) AS medals_2016,
       SUM(n_medals) AS total_medals
FROM types_of_medals_in_specific_years
GROUP BY 1
ORDER BY 7 DESC
```

Complex SQL Queries Example 5: Find the Average Number of Friends a User Has



Link to the question:
https://platform.stratascratch.com/coding/9822-find-the-average-number-of-friends-a-user-has

Available data
To answer this question, we'll have to work with the **google_friends_network** table:

user_id	friend_id
0	1
0	2
0	3
0	4
1	5

user_id:	int64
friend_id:	int64

- In this case, both **user_id** and **friend_id** values are essential for finding an answer.

How to Simplify This Complex SQL Query

Let's look at two solutions. The first is simple and effective, but it can be improved. For example, using different SQL features will allow us to remove an unnecessary nested subquery.

```
SELECT sum(cnt)::float / count(user_id) AS avg_fr
FROM
  (SELECT user_id,
          count(friend_id) AS cnt
   FROM
     (SELECT user_id,
             friend_id
      FROM google_friends_network
      UNION
      SELECT friend_id,
                  user_id
      FROM google_friends_network
      ORDER BY 1,
               2) a1
   GROUP BY 1) a2
```

This query contains a nested subquery that generates a list of all friend connections, and another that finds the number of friends for each **user_id**. In the main query, we use **SUM()** to find the total number of friends, **COUNT()** to find the number of unique users, and divide two numbers to find the average.

This is the case of using incorrect SQL features to accomplish something.

If your goal is to write SQL queries as simple as they can be, the importance of knowing SQL features can't be stressed enough.

We can simply use the **COUNT()** aggregate function to find the total number of friendships on the website.

Currently, we find the total number of users by grouping friendships by **user_id** and using **COUNT()** to find the number of users. We can instead use **COUNT()** and **DISTINCT** to find the total number of unique users.

We also don't need a nested subquery. Only a simple subquery that creates a list of all friend connections.

```
-- x * 1.0 converts values to a float. This is necessary to make sure numbers can be divided
SELECT
  COUNT(*) * 1.0 / COUNT(DISTINCT users) AS avg_num_of_friends
FROM
  (
    SELECT
      user_id AS users,
      friend_id AS friends
    FROM
      google_friends_network
    UNION
    SELECT
      friend_id AS users,
      user_id AS friends
    FROM
      google_friends_network
  ) AS t
```

As a cherry on top, we use the AS command to give values a descriptive name.

Also, we multiply one of the numbers by 1.0 to convert the integer value to a float and add a comment explaining what we did.

To improve the readability, we used uppercase for the SQL keywords.

Summary

SQL complex queries are usually born of necessity, but more often than not, they can be simplified. In this article, we explained different ways to simplify complex SQL queries while maintaining their core features.

Later we walked you through examples of simplifying answers to difficult SQL interview questions.

You should practice if you want to get into the habit of writing simple and efficient queries. The StrataScratch platform is a safe environment where you can practice writing queries to answer actual interview questions. More importantly, the platform has a community of data scientists who discuss queries and ways to simplify them.