Best Practices to Write SQL Queries: How To Structure Your Code

This article focuses on how to write SQL queries correctly and how it can still be improved, in particular when it comes to performance and readability.

Structured Query Language is an absolutely necessary skill in the data science industry. SQL is not just about writing queries, you also need to make sure that your queries are performant, fast, and readable. So, it's also necessary that you know how to write SQL queries efficiently.

This article will show you the best practices for structuring SQL queries. Even when your SQL code works correctly, it can still be improved, in particular when it comes to performance and readability. This is important because, at technical interviews, the goal is not only to check if a candidate is able to produce a working solution to a problem but also if they can prepare a solution that is efficient and clear. The same in a work environment, making the queries fast and easily understandable by others is equally important as making them correct.

Let's use a real-life example of data science interview questions that can be solved with an SQL query. We will have a solution that produces a correct output but it's very inefficient and extremely hard to read. We'll then go through several key best practices to write SQL queries and

apply them to the code to improve it so that it can be used as a response to an interview question.

SQL Interview Question Example to Practice Writing Efficient SQL Queries

Premium vs Freemium

"Find the total number of downloads for paying and non-paying users by date. Include only records where non-paying customers have more downloads than paying customers. The output should be sorted by earliest date first and contain 3 columns date, non-paying downloads, paying downloads."

Link to the question: https://platform.stratascratch.com/coding/10300-premium-vs-freemium

We'll use this SQL interview question as an example comes from technical job interviews at Microsoft and is titled "Premium vs Freemium". The task is to find the total number of downloads for paying and non-paying users by date and to include only records where non-paying customers have more downloads than paying customers. What's more, this question comes with a dataset split across three tables that will need to be merged together.

Initial Solution

Let's have the initial solution to this SQL problem that we will use as a starting point. We will not find how this solution works or how it can be obtained - it's not the point of this article. Instead, we will use it only as an illustration to syntax practices that are universal for any SQL query.

This is a solution that may be used to produce the correct output for this problem:

```
SELECT date, "NonPaying",
            paying
FROM
  (SELECT p.date,
          p.sum AS paying,
          n.nonpaying AS "NonPaying"
   FROM
     (SELECT date, sum(downloads)
```

```
      FROM ms_user_dimension a
      INNER JOIN ms_acc_dimension b ON a.acc_id = b.acc_id
      INNER JOIN ms_download_facts c ON a.user_id=c.user_id
      WHERE paying_customer = 'yes'
      GROUP BY date
      ORDER BY date) p
   JOIN
     (SELECT date, sum(downloads) AS nonpaying
      FROM ms_user_dimension a
      INNER JOIN ms_acc_dimension b ON a.acc_id = b.acc_id
      INNER JOIN ms_download_facts c ON a.user_id=c.user_id
      WHERE paying_customer = 'no'
      GROUP BY date
      ORDER BY date) n ON p.date = n.date
   ORDER BY p.date) s
GROUP BY date, "NonPaying",
              paying
HAVING ("NonPaying" - paying) >0
ORDER BY date ASC
```

Don’t worry if you don’t get right away what’s going on here. This is actually a thing, this solution is long, convoluted and confusing on purpose and the task will be to clean it up using the SQL syntax best practices. But what’s most interesting is that this solution works - we can execute it and see how many downloads from paying and non-paying customers there were on each date. But even though this solution is correct, an interviewer probably wouldn’t be happy with an answer like that. So let’s see what best practices to write SQL queries are we missing here and how to improve our chances of impressing the interviewer.

Best Practices to Write SQL Queries: How To Structure Your Code

1. Remove multiple nested queries

Even without understanding exactly what the code is doing, we can see that it has several nested queries. There is the main query in which three columns are selected, then in its FROM clause, there is another, long query, called an inner query. It has an alias ‘s’. But then this inner query ‘s’ itself also has two additional and nearly identical inner queries, ‘p’ and ‘n’, that are merged together using a JOIN statement. While it is absolutely alright to have one outer query and one inner query, more than two queries nested in each other are considered not very readable and should be avoided.

One approach to getting rid of so many nested queries is to define some or all of them in the form of the Common Table Expressions, or CTEs - the constructions that use the WITH keyword and allow to reuse one query multiple times. So let it be our first step - we can turn all the three nested queries, ‘s’, ‘p’ and ‘n’ into the CTEs.

```
WITH p AS
  (SELECT date, sum(downloads)
   FROM ms_user_dimension a
   INNER JOIN ms_acc_dimension b ON a.acc_id = b.acc_id
   INNER JOIN ms_download_facts c ON a.user_id=c.user_id
   WHERE paying_customer = 'yes'
   GROUP BY date
   ORDER BY date),

     n AS
  (SELECT date, sum(downloads) AS nonpaying
   FROM ms_user_dimension a
   INNER JOIN ms_acc_dimension b ON a.acc_id = b.acc_id
   INNER JOIN ms_download_facts c ON a.user_id=c.user_id
   WHERE paying_customer = 'no'
   GROUP BY date
   ORDER BY date),

     s AS
  (SELECT p.date,
          p.sum AS paying,
          n.nonpaying AS "NonPaying"
   FROM p
   JOIN n ON p.date = n.date
   ORDER BY p.date)

SELECT date, "NonPaying",
             paying
FROM s
GROUP BY date, "NonPaying",
               paying
HAVING ("NonPaying" - paying) >0
ORDER BY date ASC
```

Now, this query has already become a bit easier to read. Even without understanding exactly what is happening, we can see that there are two very similar steps, ‘p’ and ‘n’ that should be performed first to enable ‘s’, and then the result of ‘s’ can be used in the main query. But what exactly is this ‘p’, ‘n’, ‘s’ and other aliases? This brings us to the second point.

2. Ensure consistent aliases

Aliases in SQL can be assigned to both columns, queries and tables to change their initial names. They need to be used when merging tables with the same column names to avoid ambiguities in column names. Aliases are also useful for making the code easier to be understood by others and to replace default column names when using analytical functions (for example SUM() or COUNT()).

There are also several unwritten rules regarding aliases that should be followed because an incorrectly used alias may be more confusing than helpful. Starting with the table and query aliases, it's good when these are a bit more than just a single letter and allow us to understand what is in the table or what is produced by a query. In our case, the first CTE, currently called 'p' is used to count the number of downloads made by paying customers, so a more informative name would be for example 'paying'. It's worth noting that aliases, while informative, shouldn't also be too long, for instance, 'paying_customers' could be a bit long. Then the second CTE, the 'n' one is the same but for non-paying customers so, following the schema, it can be named 'nonpaying'.

Finally, the CTE 's' is combining the two values: numbers of downloads of paying and nonpaying customers but it's not filtering them yet because this happens in the main query. So its name can be, for example, 'all_downloads'. Now, note that these aren't yet all the tables that are given aliases. That's because in the first two CTEs we're merging three tables with each other and since they share some column names, they need to be given aliases. Currently, it's simply 'a', 'b' and 'c' but the more informative names would be 'users', 'accounts' and 'downlds' - the abbreviation is here because this table already has a column called 'downloads'.

The final thing related to table aliases is the consistency in using them. Usually it's better to either use them with all the column names, or only in the absolutely necessary places (e.g. only when defining JOINs) or not at all. Let's decide to use the table aliases in all the cases when several tables are merged, so in all the CTEs and not use them when all columns come from only one table like in the main query.

```
WITH paying AS
  (SELECT downlds.date, sum(downlds.downloads)
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'yes'
   GROUP BY downlds.date
   ORDER BY downlds.date),
```

```
    nonpaying AS
  (SELECT downlds.date, sum(downlds.downloads) AS nonpaying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'no'
   GROUP BY downlds.date
   ORDER BY downlds.date),

    all_downloads AS
  (SELECT paying.date,
          paying.sum AS paying,
          nonpaying.nonpaying AS "NonPaying"
   FROM paying
   JOIN nonpaying ON paying.date = nonpaying.date
   ORDER BY paying.date)

SELECT date, "NonPaying",
            paying
FROM all_downloads
GROUP BY date, "NonPaying",
              paying
HAVING ("NonPaying" - paying) >0
ORDER BY date ASC
```

These were the table and query aliases, now let's take a look at the aliases given to columns. The first thing is to give aliases to results of analytical functions. Take a look at the first query. There is the function SUM() but there is no alias added to it even though this column is later reused. That's why in the CTE all_downloads, we need to write paying.sum to select it. Let's add an alias, for example n_paying.

Another thing is to keep the column aliases consistent in names but also to avoid clashes with other aliases. Like in the CTE nonpaying, there is a SUM() function that, correctly, as an alias assigned but this alias is the same as the CTE's alias which can be confusing. Let's stick to the same naming convention as earlier and change this alias to n_nonpaying.

Now, in the all_downloads CTE, there is a lot going on. First of all, the alias 'paying' assigned to the second column is the same as the alias of one of the CTEs. And right after that, the alias of the third column is "NonPaying" in quotation marks. While SQL allows us to assign aliases like that, it's dangerous to use such aliases because everytime we want to reuse it, we need to use

the quotation marks again and match all the capital and small letters in the alias. We could change these two aliases to something else and without the quotation marks. But actually, it's not even necessary to use these aliases, after all, there are no analytical functions here, so the column names from earlier, 'n_paying' and n_nonpaying' stay the same and can be referenced in the main query without causing issues.

```
WITH paying AS
  (SELECT downlds.date, sum(downlds.downloads) AS n_paying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'yes'
   GROUP BY downlds.date
   ORDER BY downlds.date),

     nonpaying AS
  (SELECT downlds.date, sum(downlds.downloads) AS n_nonpaying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'no'
   GROUP BY downlds.date
   ORDER BY downlds.date),

     all_downloads AS
  (SELECT paying.date,
          paying.n_paying,
          nonpaying.n_nonpaying
   FROM paying
   JOIN nonpaying ON paying.date = nonpaying.date
   ORDER BY paying.date)

SELECT date, n_nonpaying,
             n_paying
FROM all_downloads
GROUP BY date, n_nonpaying,
               n_paying
HAVING (n_nonpaying - n_paying) >0
ORDER BY date ASC
```

There are no specific rules as to how aliases should be formatted but most people use small letters only and the underscore if an alias has multiple words.

3. Remove unnecessary ORDER BY clauses

Now that the aliases have been taken care of, let's start reducing the amount of code in our solution. The first thing is rather minor but it still contributes to the readability of the query. It's about ORDER BY clauses. These are obviously used to sort the data in a table and are often useful or even required. After all, it's sometimes necessary to use the ORDER BY clause in combination with a window function or when selecting top rows of a table using a LIMIT keyword. We may also want to arrange the final results in a certain order, at interviews, it may even be a requirement at times.

But if we have several queries, it's usually not necessary to add the same ORDER BY clause in each one of them. Look at our query, for example, we sort the results by date but we do it in all possible queries and subqueries. It's not only useless but also not efficient because each ORDER BY clause adds a little bit of complexity and hence time it takes to execute the query, especially when dealing with large datasets. So if each query has its own sorting clause, it's good to think if it's really needed. In our case, it's fine to leave it only in the last query.

```
WITH paying AS
  (SELECT downlds.date, sum(downlds.downloads) AS n_paying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'yes'
   GROUP BY downlds.date),

     nonpaying AS
  (SELECT downlds.date, sum(downlds.downloads) AS n_nonpaying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   WHERE accounts.paying_customer = 'no'
   GROUP BY downlds.date),

     all_downloads AS
  (SELECT paying.date,
          paying.n_paying,
```

```
          nonpaying.n_nonpaying
   FROM paying
   JOIN nonpaying ON paying.date = nonpaying.date)

SELECT date, n_nonpaying,
            n_paying
FROM all_downloads
GROUP BY date, n_nonpaying,
              n_paying
HAVING (n_nonpaying - n_paying) >0
ORDER BY date ASC
```

4. Remove unnecessary subqueries and CTEs

Then, there is a much more serious topic than removing unnecessary ORDER BY clauses - removing unnecessary subqueries and CTEs. We talked about them before when splitting the nested queries into CTEs but back then we just left them as they were without analysing if we really need them.

It turns out that sometimes two queries do exactly the same thing, or can be combined into one using another clauses or statements. After all, each query adds on to the complexity and time it takes to execute a query. In our case, the four different queries result in four different times the engine needs to access a table to select data from it. What's more, in three of these queries we merge multiple tables using JOINs - operations that may take a long time especially when the tables are large.

To reduce the number of queries in our case, there are two directions to go. One possibility would be to combine the query all_downloads with the main query. After all, these two queries are nearly identical and if we only added the filter, so the expression saying that the difference between n_nonpaying and n_paying should be larger than 0, to the query in all_downloads, it would produce the same results. We could safely get rid of the all_downloads CTE and instead merge the paying and nonpaying CTEs in the main query. This way we can reduce the number of queries to 3. But can we do better?

We can because it turns out that the first two CTEs, namely 'paying' and 'nonpaying' can be executed within the 'all_downloads' CTE. That's because these first two CTEs are nearly identical with the main difference being the WHERE clause. We select the same types of data from the same tables but under different conditions. But in SQL, there is another way to select and even perform arithmetic operations on data using different conditions in only one query: we need to use CASES.

We can use them to convert the strings ‘yes’ and ‘no’ from the ‘paying_customer’ column to number of downloads and then sum them to get a total. This means that the whole first CTE can be replaced with a following piece of code:

```
sum(CASE
    WHEN paying_customer = 'yes' THEN downloads
    END)
```

It will be very similar for non paying customers. Both of these statements can be executed right in the all_downloads CTE, as long as we merge the three tables, users, accounts and downlds and include the GROUP BY data statement also in this CTE.

```
WITH all_downloads AS
  (SELECT downlds.date,
          sum(CASE
                WHEN accounts.paying_customer = 'yes' THEN
downlds.downloads
            END) AS n_paying,
          sum(CASE
                WHEN accounts.paying_customer = 'no' THEN downlds.downloads
            END) AS n_nonpaying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   GROUP BY downlds.date)

SELECT date, n_nonpaying,
             n_paying
FROM all_downloads
GROUP BY date, n_nonpaying,
               n_paying
HAVING (n_nonpaying - n_paying) >0
ORDER BY date ASC
```

With these changes, we are down to only two queries, one set of multiple JOINs and two cases when data needs to be selected from a table instead of four. At the same time, we can’t solve

this question using only one query because the columns ‘n_paying’ and ‘n_nonpaying’ need to be defined in one query before being used in a filter in another query.

5. HAVING vs WHERE

Speaking of performance and efficiency, there is one more detail in our query that unnecessarily makes it slower. This detail is the HAVING clause in the main query. But why is it HAVING and not WHERE? The two clauses are very similar to each other and allow to filter the data based on some conditions. Like in this case, where the difference between n_nonpaying and n_paying values should be larger than 0. It is possible to define this condition both with WHERE and HAVING keywords but one of them is far more appropriate in this situation. The key difference is that HAVING clause can include aggregate functions e.g. SUM() or COUNT(). It enables the creation of conditions based on the sum, average, minimum or maximum values, or a number of elements in the dataset or in partitions defined using a GROUP BY clause. That’s exactly the reason why the HAVING clause must always be accompanied by a GROUP BY statement.

What many SQL users don’t know is that the HAVING clause should only be used when there’s a need to create a condition using an aggregate function. In all other cases, the WHERE clause is a better choice? Why? It all comes down to efficiency. The WHERE clause is executed together with the rest of the query so if it’s more efficient, the SQL engine may decide to restrict the number of instances in the dataset using the condition from WHERE before conducting other operations. On the other hand, the HAVING statement is always executed after the query, even though in code it’s a part of it. This almost always results in a slightly longer computing time.

In our example, the condition is based on an arithmetic operation involving two columns but it’s not the same as an aggregate function. For this reason, this condition can well be defined in a WHERE clause instead. When doing it, apart from improving the efficiency, we also get rid of the redundant GROUP BY clause.

```
WITH all_downloads AS
  (SELECT downlds.date,
          sum(CASE
                WHEN accounts.paying_customer = 'yes' THEN
downlds.downloads
            END) AS n_paying,
          sum(CASE
                WHEN accounts.paying_customer = 'no' THEN downlds.downloads
            END) AS n_nonpaying
   FROM ms_user_dimension users
```

```
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   GROUP BY downlds.date)

SELECT date, n_nonpaying,
            n_paying
FROM all_downloads
WHERE (n_nonpaying - n_paying) > 0
ORDER BY date ASC
```

6. Text formatting

The last thing that adds to the readability of queries is the formatting of the code. This, compared to other particles we have mentioned, has nothing to do with efficiency and everything to do with the code being easy to understand by everyone. This is also this detail that everyone forgets when writing an SQL query, especially in a stressful environment of a job interview. Or everyone is used to another style of text formatting and gets confused seeing other approaches.

In SQL there are no official rules on how to format text in queries but there are some unofficial guidelines that many SQL users follow. Probably the most common and well-known is that all keywords such as SELECT, FROM, WHERE, GROUP BY, ORDER BY etc. should be written in capital letters. This also applies to other built-in keywords that appear inside the clauses like JOIN, AS, IN, ON or ASC and DESC. When it comes to the names of the functions such as SUM() or COUNT(), there is no general consensus if these should be written in all capital or all small letters but it's probably better to capitalize them as well to better distinguish from column names that should be written with small letters.

Another important rule is that while it is not necessary for a code to work, each clause such as SELECT, FROM, WHERE, GROUP BY etc. should be in a new line. To further improve readability it is also a good practice to have a new line for each column name in a SELECT clause. Moreover, if we use subqueries or CTEs, a good approach is using tabulations to visually distinguish the inside of a parenthesis from the rest of the query.

The code from our example is mostly well formatted already. But we can still add new lines in the SELECT clause of the main query and capitalize names of the SUM() functions.

```
WITH all_downloads AS
  (SELECT downlds.date,
          SUM(CASE
                WHEN accounts.paying_customer = 'yes' THEN
downlds.downloads
            END) AS n_paying,
          SUM(CASE
                WHEN accounts.paying_customer = 'no' THEN downlds.downloads
            END) AS n_nonpaying
   FROM ms_user_dimension users
   INNER JOIN ms_acc_dimension accounts ON users.acc_id = accounts.acc_id
   INNER JOIN ms_download_facts downlds ON users.user_id=downlds.user_id
   GROUP BY downlds.date)

SELECT date,
       n_nonpaying,
       n_paying
FROM all_downloads
WHERE (n_nonpaying - n_paying) > 0
ORDER BY date ASC
```

We know that sometimes it's hard to remember all the text formatting rules or adjust everything manually, for example capitalize the keywords or add tabulations. So, it's good to get in the habit of using these formatting rules when writing SQL queries because it makes it easier to understand the code for ourselves and to write more readable code at job interviews.

Conclusion

To conclude, here is once again the list of how to write SQL queries best practices:

1. Remove multiple nested queries
2. Ensure consistent aliases
3. Remove unnecessary ORDER BY clauses
4. Remove unnecessary subqueries
5. If possible, use WHERE and not HAVING
6. Format your code according to best practices

What's more, when removing multiple nested queries, you can turn them into CTEs - a good rule of thumb is that one subquery is fine but multiple sub-subqueries or a subquery repeated multiple times should become a CTE.

Consistency in aliases includes making them informative, longer than 1 letter, but also not too long. Be consistent whether you use aliases or not. Add aliases to columns produced by analytical functions. Stick to some naming convention and avoid columns or tables sharing the same alias. Use small letters for aliases and underscores if it has multiple words. Don't use quotation marks to define an alias.

To remove an ORDER BY clause, remember it often doesn't need to be repeated in multiple queries. If it is, see if it's possible to only have it in the last one.

Sometimes subqueries or CTEs may be combined together. Look for subqueries that look similar or result in the same columns - these are often easiest to combine. One trick is to replace a WHERE clause with CASEs.

Use the HAVING clause only in combination with aggregate functions, in nearly all other cases the choice should be WHERE.

When formatting the text of the query, remember to write keywords and possibly functions in capital letters, aliases, table and column names in small letters. Each clause and possibly each column in the SELECT clause should be in a new line. Use tabulation to show subqueries and CTEs.