**How To Write Better SQL Queries: The Definitive Guide – Part 1**

*Most forget that SQL isn’t just about writing queries, which is just the first step down the road. Ensuring that queries are performant or that they fit the context that you’re working in is a whole other thing. This SQL tutorial will provide you with a small peek at some steps that you can go through to evaluate your query.*

**By Karlijn Willems, Data Science Journalist & [DataCamp](https://www.datacamp.com/) Contributor**.

Structured Query Language (SQL) is an indispensable skill in the data science industry and generally speaking, learning this skill is fairly easy. However, most forget that SQL isn’t just about writing queries, which is just the first step down the road. Ensuring that queries are performant or that they fit the context that you’re working in is a whole other thing.

That’s why this SQL tutorial will provide you with a small peek at some steps that you can go through to evaluate your query:

* First off, you’ll start with a short overview of the [importance of learning SQL](https://www.datacamp.com/community/tutorials/sql-tutorial-query#importance) for jobs in data science;
* Next, you’ll first learn more about how [SQL query processing and execution](https://www.datacamp.com/community/tutorials/sql-tutorial-query#execution) so that you can properly understand the importance of writing qualitative queries: more specifically, you’ll see that the query is parsed, rewritten, optimized and finally evaluated;
* With that in mind, you’ll not only go over some [query anti-patterns](https://www.datacamp.com/community/tutorials/sql-tutorial-query#antipattern) that beginners make when writing queries, but you’ll also learn more about alternatives and solutions to those possible mistakes; You’ll also learn more about the [set-based versus the procedural approach](https://www.datacamp.com/community/tutorials/sql-tutorial-query#setbased) to querying.
* You’ll also see that these anti-patterns stem from performance concerns and that, besides the “manual” approach to improving SQL queries, you can [analyze your queries](https://www.datacamp.com/community/tutorials/sql-tutorial-query#queryplan) also in a more structured, in-depth way by making use of some other tools that help you to see the query plan; And,
* You’ll briefly go more into [time complexity and the big O notation](https://www.datacamp.com/community/tutorials/sql-tutorial-query#bigo) to get an idea about the time complexity of an execution plan before you execute your query; Lastly,
* You’ll briefly get some pointers on how you can [tune your query](https://www.datacamp.com/community/tutorials/sql-tutorial-query#tune) further.



Are you interested in an SQL course? Take DataCamp’s [Intro to SQL for Data Science](https://www.datacamp.com/courses/intro-to-sql-for-data-science) course!

**Why Should I Learn SQL For Data Science?**

SQL is far from dead: it’s one of the most in-demand skills that you find in job descriptions from the data science industry, whether you’re applying for a data analyst, a data engineer, a data scientist or [any other roles](https://www.datacamp.com/community/tutorials/data-science-industry-infographic). This is confirmed by 70% of the respondents of the 2016 O’Reilly Data Science Salary Survey, who indicate that they use SQL in their professional context. What’s more, in this survey, SQL stands out way above the R (57%) and Python (54%) programming languages.

You get the picture: SQL is a must-have skill when you’re working towards getting a job in the data science industry.

Not bad for a language that was developed in the early 1970s, right?

But why exactly is it that it is so frequently used? And why isn’t it dead even though it has been around for such a long time?

There are several reasons: one of the first reasons would be that companies mostly store data in Relational Database Management Systems (RDBMS) or in Relational Data Stream Management Systems (RDSMS) and you need SQL to access that data. SQL is the lingua franca of data: it gives you the ability to interact with almost any database or to even build your own locally!

As if this wasn’t enough yet, keep in mind that there are quite a few SQL implementations that are incompatible between vendors and do not necessarily follow standards. Knowing the standard SQL is thus a requirement for you to find your way around in the (data science) industry.

On top of that, it’s safe to say that SQL has also been embraced by newer technologies, such as Hive, a SQL-like query language interface to query and manage large datasets, or Spark SQL, which you can use to execute SQL queries. Once again, the SQL that you find there will differ from the standard that you might have learned, but the learning curve will be considerably easier.

If you do want to make a comparison, consider it as learning linear algebra: by putting all that effort into this one subject, you know that you will be able to use it to master machine learning as well!

In short, this is why you should learn this query language:

* It’s is fairly easy to learn, even for total newbies. The learning curve is quite easy and gradual, so you’ll be writing queries in no time.
* It follows the “learn once, use anywhere” principle, so it’s a great investment of your time!
* It’s an excellent addition to programming languages; In some cases, writing a query is even preferred over writing code because it’s more performant!
* …

What are you still waiting for? :)

**SQL Processing & Query Execution**

To improve the performance of your SQL query, you first have to know what happens internally when you press the shortcut to run the query.

First, the query is parsed into a “parse tree”; The query is analyzed to see if it satisfies the syntactical and semantical requirements. The parser creates an internal representation of the input query. This output is then passed on to the rewrite engine.

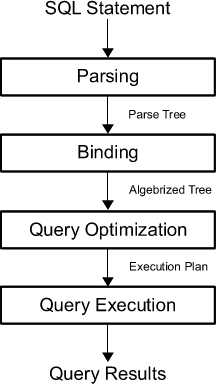
It is then the task of the optimizer to find the optimal execution or query plan for the given query. The execution plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

To indeed find the most optimal execution plan, the optimizer enumerates all possible execution plans, determines the quality or cost of each plan, takes information about the current database state and then chooses the best one as the final execution plan. Because query optimizers can be imperfect, database users and administrators sometimes need to manually examine and tune the plans produced by the optimizer to get better performance.

Now you probably wonder what is considered to be a “good query plan”.

As you already read, the quality of cost of a plan plays a huge role. More specifically, things such as the number of disk I/Os that are required to evaluate the plan, the plan’s CPU cost and the overall response time that can be observed by the database client and the total execution time are essential. That is where the notion of time complexity will come in. You’ll read more about this later on.

Next, the chosen query plan is executed, evaluated by the system’s execution engine and the results of your query are returned.



**Writing SQL Queries**

What might not have become clear from the previous section is that the Garbage In, Garbage Out (GIGO) principle naturally surfaces within the query processing and execution: the one who formulates the query also holds the keys to the performance of your SQL queries. If the optimizer gets a badly formulated query, it will only be able to do as much…

That means that there are some things that *you* can do when you’re writing a query. As you already saw in the introduction, the responsibility is two-fold: it’s not only about writing queries that live up to a certain standard, but also about gathering an idea of where performance problems might be lurking within your query.

An ideal starting point is to think of “spots” within your queries where issues might sneak in. And, in general, there are four clauses and keywords where newbies can expect performance issues to occur:

* The WHERE clause;
* Any INNER JOIN or LEFT JOIN keywords; And,
* The HAVING clause;

Granted, this approach is simple and naive, but as a beginner, these clauses and statements are nice pointers and it’s safe to say that when you’re just starting out, these spots are the ones where mistakes happen and, ironically enough, where they’re also hard to spot.

However, you should also realize that performance is something that needs a context to become meaningful: simply saying that these clauses and keywords are bad isn’t the way to go when you’re thinking about SQL performance. Having a WHERE or HAVING clause in your query doesn’t necessarily mean that it’s a bad query…

Take a look at the following section to learn more about anti-patterns and alternative approaches to building up your query. These tips and tricks are meant as a guide. How and if you actually need to rewrite your query depends on the amount of data, the database and the number of times you need to execute the query, among other things. It entirely depends on the goal of your query and having some prior knowledge about the database that you want to query is crucial!

**1. Only Retrieve The Data You Need**

The mindset of “the more data, the better” isn’t one that you should necessarily live by when you’re writing SQL queries: not only do you risk obscuring your insights by getting more than what you actually need, but also your performance might suffer from the fact that your query pulls up too much data.

That’s why it’s generally a good idea to look out for the SELECT statement, the DISTINCT clause and the LIKE operator.

**The SELECT Statement**

A first thing that you can already check when you have written your query is whether the SELECT statement is as compact as possible. Your aim here should be to remove unncessary columns from SELECT. This way you force yourself to only pull up data that serves your query goal.

In case you have correlated subqueries that have EXISTS, you should try to use a constant in the SELECT statement of that subquery instead of selecting the value of an actual column. This is especially handy when you’re checking the existence only.

**Remember** that a correlated subquery is a subquery that uses values from the outer query. And note that, even though NULL can work in this context as a “constant”, it’s very confusing!

Consider the following example to understand what is meant by using a constant:

SELECT driverslicensenr, name

FROM Drivers

WHERE EXISTS (SELECT '1' FROM Fines

WHERE fines.driverslicensenr = drivers.driverslicensenr);

**Tip**: it’s handy to know that having a correlated subquery isn’t always a good idea. You can always consider getting rid of them by, for example, rewriting them with an INNER JOIN:

SELECT driverslicensenr, name

FROM drivers

INNER JOIN fines ON fines.driverslicensenr = drivers.driverslicensenr;

**The DISTINCT Clause**

The SELECT DISTINCT statement is used to return only distinct (different) values. DISTINCT is a clause that you should definitely try to avoid if you can; Like you have read in other examples, the execution time only increases if you add this clause to your query. It’s therefore always a good idea to consider whether you really need this DISTINCT operation to take place to get the results that you want to accomplish.

**The LIKE Operator**

When you use the LIKE operator in a query, the index isn’t used if the pattern starts with % or \_. It will prevent the database from using an index (if it exists). Of course, from another point of view, you could also argue that this type of query potentially leaves the door open to retrieve too many records that don’t necessarily satisfy your query goal.

Once again, your knowledge of the data that is stored in the database can help you to formulate a pattern that will filter correctly through all the data to find only the rows that really matter for your query.

**2. Limit Your Results**

When you can not avoid filtering down on your SELECT statement, you can consider limiting your results in other ways. Here’s where approaches such as the LIMIT clause and data type conversions come in.

**TOP, LIMIT And ROWNUM Clauses**

You can add the LIMIT or TOP clauses to your queries to set a maximum number of rows for the result set. Here are some examples:

SELECT TOP 3 \*

FROM Drivers;

**Note** that you can further specify the PERCENT, for example, if you change the first line of the query by SELECT TOP 50 PERCENT \*.

SELECT driverslicensenr, name

FROM Drivers

LIMIT 2;

Additionally, you can also add the ROWNUM clause, which is equivalent to using LIMIT in your query:

SELECT \*

FROM Drivers

WHERE driverslicensenr = 123456 AND ROWNUM <= 3;

**Data Type Conversions**

You should always use the most efficient, that is, smallest, data types possible. There’s always a risk when you provide a huge data type when a smaller one will be more sufficient.

However, when you add data type conversion to your query, you only increase the execution time.

An alternative is just to avoid data type conversion as much as possible. Note also here that it’s not always possible to remove or omit the data type conversion from your queries, but that you should definitely aim to be careful in including them and that when you do, you test the effect of the addition before you run the query.

*Most forget that SQL isn’t just about writing queries, which is just the first step down the road. Ensuring that queries are performant or that they fit the context that you’re working in is a whole other thing. This SQL tutorial will provide you with a small peek at some steps that you can go through to evaluate your query.*

**3. Don’t Make Queries More Complex Than They Need To Be**

The data type conversions bring you to a next point: you should not over-engineer your queries. Try to keep them simple and efficient. This might seem too simple or stupid to even be a tip, especially because queries can get complex.

However, you’ll see in the examples that are mentioned in the next sections that you can easily start making simple queries more complex than they need to be.

**The OR Operator**

When you use the OR operator in your query, it’s likely that you’re not using an index.

**Remember** that an index is a data structure that improves the speed of the data retrieval in your database table, but it comes at a cost: there will be additional writes and additional storage space is needed to maintain the index data structure. Indexes are used to quickly locate or look up data without having to search every row in a database every time the database table is accessed. Indexes can be created by using one or more columns in a database table.

If you don’t make use of the indexes that the database includes, your query will inevitably take longer to run. That’s why it’s best to look for alternatives to using the OR operator in your query;

Consider the following query:

SELECT driverslicensenr, name

FROM Drivers

WHERE driverslicensenr = 123456

OR driverslicensenr = 678910

OR driverslicensenr = 345678;

You can replace the operator by:

* A condition with IN; or

SELECT driverslicensenr, name

FROM Drivers

WHERE driverslicensenr IN (123456, 678910, 345678);

* Two SELECT statements with a UNION.

**Tip**: here, you need to be careful not to unnecessarily use the UNIONoperation because you go through the same table multiple times. At the same time, you have to realize that when you use a UNION in your query, the execution time will increase. Alternatives to the UNION operation are: reformulating the query in such a way that all conditions are placed in one SELECT instruction, or using an OUTER JOIN instead of UNION.

**Tip**: keep in mind also here that, even though OR -and also the other operators that will be mentioned in the following sections- likely isn’t using an index, index lookups aren’t always preferred!

**The NOT Operator**

When your query contains the NOT operator, it’s likely that the index is not used, just like with the OR operator. This will inevitably slow down your query. If you don’t know what is meant here, consider the following query:

SELECT driverslicensenr, name

FROM Drivers

WHERE NOT (year > 1980);

This query will definitely run slower than you would maybe expect, mainly because it’s formulated a lot more complex than it could be: in cases like this one, it’s best to look for an alternative. Consider replacing NOT by comparison operators, such as >, <> or !>; The example above might indeed be rewritten and become something like this:

SELECT driverslicensenr, name FROM Drivers WHERE year <= 1980;

That already looks way neater, doesn’t it?

**The AND Operator**

The AND operator is another operator that doesn’t make use of the index and that can slow your query down if used in an overly complex and inefficient way, like in the example below:

SELECT driverslicensenr, name

FROM Drivers

WHERE year >= 1960 AND year <= 1980;

It’s better to rewrite this query and use BETWEEN operator:

SELECT driverslicensenr, name

FROM Drivers

WHERE year BETWEEN 1960 AND 1980;

**The ANY and ALL Operators**

Also the ALL and ALL operators are some that you should be careful with because, by including these into your queries, the index won’t be used. Alternatives that will come in handy here are aggregation functions like MINor MAX.

**Tip**: in cases where you make use of the proposed alternatives, you should be aware of the fact that all aggregation functions like SUM, AVG, MIN, MAXover many rows can result in a long-running query. In such cases, you can try to either minimize the amount of rows to handle or pre-calculate these values. You see once again that it’s important to be aware of your environment, your query goal, … when you make decisions on which query to use!

**Isolate Columns in Conditions**

Also in cases where a column is used in a calculation or in a scalar function, the index isn’t used. A possible solution would be to simply isolate the specific column so that it no longer is a part of the calculation or the function. Consider the following example:

SELECT driverslicensenr, name

FROM Drivers

WHERE year + 10 = 1980;

This is looks funky, huh? Instead, try to reconsider the calculation and rewrite the query to something like this:

SELECT driverslicensenr, name

FROM Drivers

WHERE year = 1970;

**4. No Brute Force**

This last tip actually means that you shouldn’t try to restrict the query too much because it can affect its performance. This is especially true for joins and for the HAVING clause.

**Joins**

* **The Order Of Tables**

When you join two tables, it can be important to consider the order of the tables in your join. If you notice that one table is considerably larger than the other one, you might want to rewrite your query so that the biggest table is placed last in the join.

* **Redundant Conditions on Joins**

When you add too many conditions to your joins, you basically obligate SQL to choose a certain path. It could be, though, that this path isn’t always the more performant one.

**The HAVING Clause**

The HAVING clause was originally added to SQL because the WHERE keyword could not be used with aggregate functions. HAVING is typically used with the GROUP BY clause to restrict the groups of returned rows to only those that meet certain conditions. However, if you use this clause in your query, the index is not used, which -as you already know- can result in a query that doesn’t really perform all that well.

If you’re looking for an alternative, consider using the WHERE clause. Consider the following queries:

SELECT state, COUNT(\*)

FROM Drivers

WHERE state IN ('GA', 'TX')

GROUP BY state

ORDER BY state

SELECT state, COUNT(\*)

FROM Drivers

GROUP BY state

HAVING state IN ('GA', 'TX')

ORDER BY state

The first query uses the WHERE clause to restrict the number of rows that need to be summed, whereas the second query sums up all the rows in the table and then uses HAVING to throw away the sums it calculated. In these types of cases, the alternative with the WHERE clause is obviously the better one, as you don’t waste any resources.

You see that this is not about limiting the result set, rather about limiting the intermediate number of records within a query.

**Note** that the difference between these two clauses lies in the fact that the WHERE clause introduces a condition on individual rows, while the HAVINGclause introduces a condition on aggregations or results of a selection where a single result, such as MIN, MAX, SUM,… has been produced from multiple rows.

You see, evaluating the quality, writing and rewriting of queries is not an easy job when you take into account that they need to be as performant as possible; Avoiding anti-patterns and considering alternatives will also be a part of responsibility when you write queries that you want to run on databases in a professional environment.

This list was just a small overview of some anti-patterns and tips that will hopefully help beginners; If you’d like to get an insight into what more senior developers consider the most frequent anti-patterns, check out [this discussion](https://stackoverflow.com/questions/346659/what-are-the-most-common-sql-anti-patterns).

**Bio:**[**Karlijn Willems**](https://www.linkedin.com/in/karlijnwillems) is a data science journalist and writes for the [DataCamp community](https://www.datacamp.com/community/authors/karlijn-willems" \t "_blank), focusing on data science education, the latest news and the hottest trends. She holds degrees in Literature and Linguistics and Information Management.

[Original](https://www.datacamp.com/community/tutorials/sql-tutorial-query#gs.QQP_Fhg). Reposted with permission.

**Related:**

* [7 Steps to Mastering SQL for Data Science](https://www.kdnuggets.com/2016/06/seven-steps-mastering-sql-data-science.html)
* [The Most Underutilized Function in SQL](https://www.kdnuggets.com/2017/03/most-underutilized-function-sql.html)
* [Doing Statistics with SQL](https://www.kdnuggets.com/2016/08/doing-statistics-sql.html)