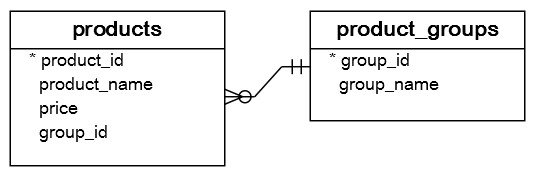
Setting up sample tables

First, [create two tables](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-create-table/) named products and product\_groups for the demonstration:



CREATE TABLE product\_groups (

group\_id serial PRIMARY KEY,

group\_name VARCHAR (255) NOT NULL

);

CREATE TABLE products (

product\_id serial PRIMARY KEY,

product\_name VARCHAR (255) NOT NULL,

price DECIMAL (11, 2),

group\_id INT NOT NULL,

FOREIGN KEY (group\_id) REFERENCES product\_groups (group\_id)

);

Code language: SQL (Structured Query Language) (sql)

Second, [insert](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-insert/) some rows into these tables:

INSERT INTO product\_groups (group\_name)

VALUES

('Smartphone'),

('Laptop'),

('Tablet');

INSERT INTO products (product\_name, group\_id,price)

VALUES

('Microsoft Lumia', 1, 200),

('HTC One', 1, 400),

('Nexus', 1, 500),

('iPhone', 1, 900),

('HP Elite', 2, 1200),

('Lenovo Thinkpad', 2, 700),

('Sony VAIO', 2, 700),

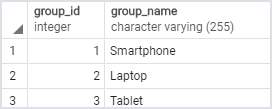
('Dell Vostro', 2, 800),

('iPad', 3, 700),

('Kindle Fire', 3, 150),

('Samsung Galaxy Tab', 3, 200);

Code language: SQL (Structured Query Language) (sql)



Introduction to PostgreSQL window functions

The easiest way to understand the window functions is to start by reviewing the [aggregate functions](https://www.postgresqltutorial.com/postgresql-aggregate-functions/). An aggregate function aggregates data from a set of rows into a single row.

The following example uses the [AVG()](https://www.postgresqltutorial.com/postgresql-avg-function/) aggregate function to calculate the average price of all products in the products table.

SELECT

AVG (price)

FROM

products;

Code language: SQL (Structured Query Language) (sql)

PostgreSQL Window Function - AVG function

To apply the aggregate function to subsets of rows, you use the [GROUP BY](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-group-by/) clause. The following example returns the average price for every product group.

SELECT

group\_name,

AVG (price)

FROM

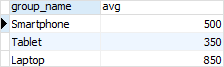
products

INNER JOIN product\_groups USING (group\_id)

GROUP BY

group\_name;

Code language: SQL (Structured Query Language) (sql)



As you see clearly from the output, the [AVG()](https://www.postgresqltutorial.com/postgresql-avg-function/) function reduces the number of rows returned by the queries in both examples.

Similar to an aggregate function, a window function operates on a set of rows. However, it does not reduce the number of rows returned by the query.

The term *window* describes the set of rows on which the window function operates. A window function returns values from the rows in a window.

For instance, the following query returns the product name, the price, product group name, along with the average prices of each product group.

SELECT

product\_name,

price,

group\_name,

AVG (price) OVER (

PARTITION BY group\_name

)

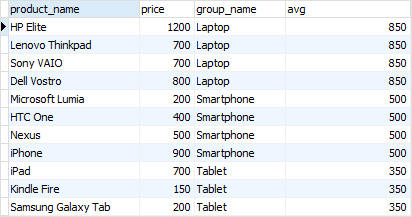
FROM

products

INNER JOIN

product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



In this query, the AVG() function works as a *window function* that operates on a set of rows specified by the OVER clause. Each set of rows is called a window.

The new syntax for this query is the OVER clause:

AVG(price) OVER (PARTITION BY group\_name)

Code language: SQL (Structured Query Language) (sql)

In this syntax, the PARTITION BY distributes the rows of the result set into groups and the AVG() function is applied to each group to return the average price for each.

Note that a window function always performs the calculation on the result set after the [JOIN](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-joins/), [WHERE](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-where/), [GROUP BY](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-group-by/) and [HAVING](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-having/) clause and before the final [ORDER BY](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-order-by/) clause in the evaluation order.

PostgreSQL Window Function Syntax

PostgreSQL has a sophisticated [syntax for window function call](https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS). The following illustrates the simplified version:

window\_function(arg1, arg2,..) OVER (

[PARTITION BY partition\_expression]

[ORDER BY sort\_expression [ASC | DESC] [NULLS {FIRST | LAST }])

Code language: SQL (Structured Query Language) (sql)

In this syntax:

window\_function(arg1,arg2,...)

The window\_function is the name of the window function. Some window functions do not accept any argument.

PARTITION BY clause

The PARTITION BY clause divides rows into multiple groups or partitions to which the window function is applied. Like the example above, we used the product group to divide the products into groups (or partitions).

The PARTITION BY clause is optional. If you skip the PARTITION BY clause, the window function will treat the whole result set as a single partition.

ORDER BY clause

The ORDER BY clause specifies the order of rows in each partition to which the window function is applied.

The ORDER BY clause uses the NULLS FIRST or NULLS LAST option to specify whether nullable values should be first or last in the result set. The default is NULLS LAST option.

 frame\_clause

The frame\_clause defines a subset of rows in the current partition to which the window function is applied. This subset of rows is called a frame.

If you use multiple window functions in a query:

SELECT

wf1() OVER(PARTITION BY c1 ORDER BY c2),

wf2() OVER(PARTITION BY c1 ORDER BY c2)

FROM table\_name;

Code language: SQL (Structured Query Language) (sql)

you can use the WINDOW clause to shorten the query as shown in the following query:

SELECT

wf1() OVER w,

wf2() OVER w,

FROM table\_name

WINDOW w AS (PARTITION BY c1 ORDER BY c2);

Code language: SQL (Structured Query Language) (sql)

It is also possible to use the WINDOW clause even though you call one window function in a query:

SELECT wf1() OVER w

FROM table\_name

WINDOW w AS (PARTITION BY c1 ORDER BY c2);

Code language: SQL (Structured Query Language) (sql)

PostgreSQL window function List

The following table lists all window functions provided by PostgreSQL. Note that some aggregate functions such as AVG(), MIN(), MAX(), SUM(), and COUNT() can be also used as window functions.



| **Name** | **Description** |
| --- | --- |
| [CUME\_DIST](https://www.postgresqltutorial.com/postgresql-cume_dist-function/) | Return the relative rank of the current row. |
| [DENSE\_RANK](https://www.postgresqltutorial.com/postgresql-dense_rank-function/) | Rank the current row within its partition without gaps. |
| [FIRST\_VALUE](https://www.postgresqltutorial.com/postgresql-first_value-function/) | Return a value evaluated against the first row within its partition. |
| [LAG](https://www.postgresqltutorial.com/postgresql-lag-function/) | Return a value evaluated at the row that is at a specified physical offset row before the current row within the partition. |
| [LAST\_VALUE](https://www.postgresqltutorial.com/postgresql-last_value-function/) | Return a value evaluated against the last row within its partition. |
| [LEAD](https://www.postgresqltutorial.com/postgresql-lead-function/) | Return a value evaluated at the row that is offset rows after the current row within the partition. |
| [NTILE](https://www.postgresqltutorial.com/postgresql-ntile-function/) | Divide rows in a partition as equally as possible and assign each row an integer starting from 1 to the argument value. |
| [NTH\_VALUE](https://www.postgresqltutorial.com/postgresql-nth_value-function/) | Return a value evaluated against the nth row in an ordered partition. |
| [PERCENT\_RANK](https://www.postgresqltutorial.com/postgresql-percent_rank-function/) | Return the relative rank of the current row (rank-1) / (total rows – 1) |
| [RANK](https://www.postgresqltutorial.com/postgresql-rank-function/) | Rank the current row within its partition with gaps. |
| [ROW\_NUMBER](https://www.postgresqltutorial.com/postgresql-row_number/) | Number the current row within its partition starting from 1. |

The ROW\_NUMBER(), RANK(), and DENSE\_RANK() functions

The [ROW\_NUMBER()](https://www.postgresqltutorial.com/postgresql-row_number/), [RANK()](https://www.postgresqltutorial.com/postgresql-rank-function/), and [DENSE\_RANK()](https://www.postgresqltutorial.com/postgresql-dense_rank-function/) functions assign an integer to each row based on its order in its result set.

The ROW\_NUMBER() function assigns a sequential number to each row in each partition. See the following query:

SELECT

product\_name,

group\_name,

price,

ROW\_NUMBER () OVER (

PARTITION BY group\_name

ORDER BY

price

)

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



The [RANK()](https://www.postgresqltutorial.com/postgresql-rank-function/) function assigns ranking within an ordered partition. If rows have the same values, the  RANK() function assigns the same rank, with the next ranking(s) skipped.

See the following query:

SELECT

product\_name,

group\_name,

price,

RANK () OVER (

PARTITION BY group\_name

ORDER BY

price

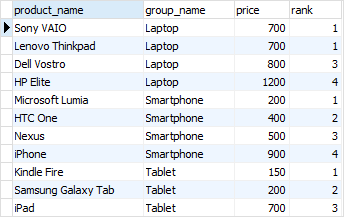
)

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



In the laptop product group, both Dell Vostro and Sony VAIO products have the same price, therefore, they receive the same rank 1. The next row in the group is HP Elite that receives the rank 3 because the rank 2 is skipped.

Similar to the RANK() function, the [DENSE\_RANK()](https://www.postgresqltutorial.com/postgresql-dense_rank-function/) function assigns a rank to each row within an ordered partition, but the ranks have no gap. In other words, the same ranks are assigned to multiple rows and no ranks are skipped.

SELECT

product\_name,

group\_name,

price,

DENSE\_RANK () OVER (

PARTITION BY group\_name

ORDER BY

price

)

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



Within the laptop product group, rank 1 is assigned twice to Dell Vostro and Sony VAIO. The next rank is 2 assigned to HP Elite.

The FIRST\_VALUE and LAST\_VALUE functions

The [FIRST\_VALUE()](https://www.postgresqltutorial.com/postgresql-first_value-function/) function returns a value evaluated against the first row within its partition, whereas the [LAST\_VALUE()](https://www.postgresqltutorial.com/postgresql-last_value-function/) function returns a value evaluated against the last row in its partition.

The following statement uses the FIRST\_VALUE() to return the lowest price for every product group.

SELECT

product\_name,

group\_name,

price,

FIRST\_VALUE (price) OVER (

PARTITION BY group\_name

ORDER BY

price

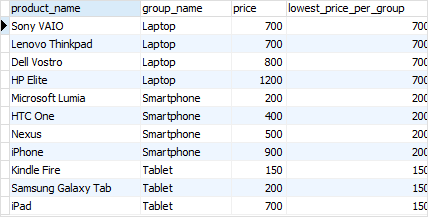
) AS lowest\_price\_per\_group

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



The following statement uses the LAST\_VALUE() function to return the highest price for every product group.

SELECT

product\_name,

group\_name,

price,

LAST\_VALUE (price) OVER (

PARTITION BY group\_name

ORDER BY

price RANGE BETWEEN UNBOUNDED PRECEDING

AND UNBOUNDED FOLLOWING

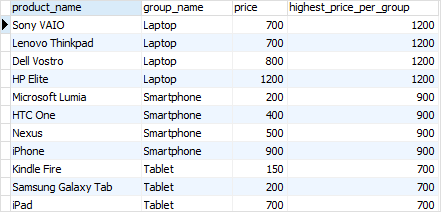
) AS highest\_price\_per\_group

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



Notice that we added the frame clause RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING because by default the frame clause is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

The LAG and LEAD functions

The [LAG()](https://www.postgresqltutorial.com/postgresql-lag-function/) function has the ability to access data from the previous row, while the [LEAD()](https://www.postgresqltutorial.com/postgresql-lead-function/) function can access data from the next row.

Both [LAG()](https://www.postgresqltutorial.com/postgresql-lag-function/) and [LEAD()](https://www.postgresqltutorial.com/postgresql-lead-function/) functions have the same syntax as follows:

LAG (expression [,offset] [,default]) over\_clause;

LEAD (expression [,offset] [,default]) over\_clause;

Code language: SQL (Structured Query Language) (sql)

In this syntax:

* expression – a column or expression to compute the returned value.
* offset – the number of rows preceding ( LAG)/ following ( LEAD) the current row. It defaults to 1.
* default – the default returned value if the offset goes beyond the scope of the window. The default is NULL if you skip it.

The following statement uses the LAG() function to return the prices from the previous row and calculates the difference between the price of the current row and the previous row.

SELECT

product\_name,

group\_name,

price,

LAG (price, 1) OVER (

PARTITION BY group\_name

ORDER BY

price

) AS prev\_price,

price - LAG (price, 1) OVER (

PARTITION BY group\_name

ORDER BY

price

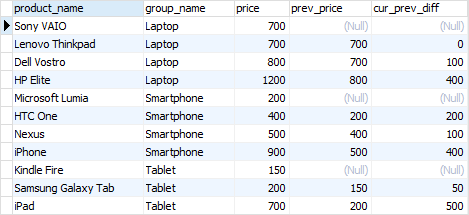
) AS cur\_prev\_diff

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



The following statement uses the LEAD() function to return the prices from the next row and calculates the difference between the price of the current row and the next row.

SELECT

product\_name,

group\_name,

price,

LEAD (price, 1) OVER (

PARTITION BY group\_name

ORDER BY

price

) AS next\_price,

price - LEAD (price, 1) OVER (

PARTITION BY group\_name

ORDER BY

price

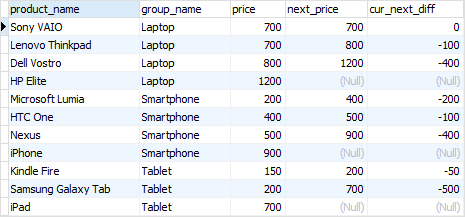
) AS cur\_next\_diff

FROM

products

INNER JOIN product\_groups USING (group\_id);

Code language: SQL (Structured Query Language) (sql)



In this tutorial, we have introduced you to the PostgreSQL window functions and shown you some examples of using them to query data.

## Understanding the Postgres EXPLAIN cost

EXPLAIN is very useful for understanding the performance of a Postgres query. It returns the execution plan generated by PostgreSQL query planner for a given statement. The EXPLAIN command specifies whether the tables referenced in a statement will be searched using an index scan or a sequential scan.

Some of the first things you’ll notice when reviewing the output of an EXPLAIN command are the cost statistics, so it’s natural to wonder what they mean, how they’re calculated, and how they’re used.

In short, the PostgreSQL query planner is estimating how much time the query will take (in an arbitrary unit), with both a startup cost and a total cost for each operation. More on that later. When it has multiple options for executing a query, it uses these costs to choose the cheapest, and therefore hopefully fastest, option.

## What unit are the costs in?

**The costs are in an arbitrary unit.** A common misunderstanding is that they are in milliseconds or some other unit of time, but that’s not the case.

The cost units are anchored (by default) to a single sequential page read costing 1.0 units (seq\_page\_cost). Each row processed adds 0.01 (cpu\_tuple\_cost), and each non-sequential page read adds 4.0 (random\_page\_cost). There are many more constants like this, all of which are configurable. That last one is a particularly common candidate, at least on modern hardware. We’ll look into that more in a bit.

## Startup Costs

The first numbers you see after cost= are known as the “startup cost”. This is an estimate of how long it will take to **fetch the first row**. As such, the startup cost of an operation includes the cost of its children.

For a sequential scan, the startup cost will generally be close to zero, as it can start fetching rows straight away. For a sort operation, it will be higher because a large proportion of the work needs to be done before rows can start being returned.

To look at an example, let’s create a simple test table with 1000 usernames:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE users (      id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,      username text NOT NULL);  INSERT INTO users (username)  SELECT 'person' || n  FROM generate\_series(1, 1000) AS n;  ANALYZE users; |

Let’s take a look at a simple query plan, with a couple of operations:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | EXPLAIN SELECT \* FROM users ORDER BY username;    QUERY PLAN                                                    |  --------------------------------------------------------------+  Sort  (cost=66.83..69.33 rows=1000 width=17)                  |    Sort Key: username                                          |    ->  Seq Scan on users  (cost=0.00..17.00 rows=1000 width=17)| |

In the above query plan, as expected, the estimated statement execution cost for the Seq Scan is 0.00, and for the Sort is 66.83.

## Total costs

The second cost statistic, after the startup cost and the two dots, is known as the “total cost”. This is an estimate of how long it will take to **return all the rows**.

Let’s look at that example query plan again:

|  |  |
| --- | --- |
| 1  2  3  4  5 | QUERY PLAN                                                    |  --------------------------------------------------------------+  Sort  (cost=66.83..69.33 rows=1000 width=17)                  |    Sort Key: username                                          |    ->  Seq Scan on users  (cost=0.00..17.00 rows=1000 width=17)| |

We can see that the total cost of the Seq Scan operation is 17.00. For the Sort operation is 69.33, which is not much more than its startup cost (as expected).

Total costs usually include the cost of the operations preceding them. For example, the total cost of the Sort operation above includes that of the Seq Scan.

An important exception is LIMIT clauses, which the planner uses to estimate whether it can abort early. If it only needs a small number of rows, the conditions for which are common, it may calculate that a simpler scan choice is cheaper (likely to be faster).

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | EXPLAIN SELECT \* FROM users LIMIT 1;    QUERY PLAN                                                    |  --------------------------------------------------------------+  Limit  (cost=0.00..0.02 rows=1 width=17)                      |    ->  Seq Scan on users  (cost=0.00..17.00 rows=1000 width=17)| |

As you can see, the total cost reported on the Seq Scan node is still 17.00, but the full cost of the Limit operation is reported to be 0.02. This is because the planner expects that it will only have to process 1 row out of 1000, so the cost, in this case, is estimated to be 1000th of the total.

## How the costs are calculated

In order to calculate these costs, the Postgres query planner uses both constants (some of which we’ve already seen) and metadata about the contents of the database. The metadata is often referred to as “statistics”.

Statistics are gathered via [**ANALYZE**](https://www.postgresql.org/docs/current/sql-analyze.html) (not to be confused with the EXPLAIN parameter of the same name), and stored in **[pg\_statistic](https://www.postgresql.org/docs/current/catalog-pg-statistic.html" \t "_blank)**. They are also refreshed automatically as part of **[autovacuum](https://www.postgresql.org/docs/current/routine-vacuuming.html" \l "VACUUM-FOR-STATISTICS" \t "_blank)**.

These statistics include a number of very useful things, like roughly the number of rows each table has, and what the most common values in each column are.

Let’s look at a simple example, using the same query data as before:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | EXPLAIN SELECT count(\*) FROM users;    QUERY PLAN                                                   |  -------------------------------------------------------------+  Aggregate  (cost=19.50..19.51 rows=1 width=8)                |    ->  Seq Scan on users  (cost=0.00..17.00 rows=1000 width=0)| |

In our case, the planner’s statistics suggested the data for the table was stored within 7 pages (or blocks), and that 1000 rows would be returned. The cost parameters seq\_page\_cost, cpu\_tuple\_cost, and cpu\_operator\_cost were left at their defaults of 1, 0.01, and 0.0025 respectively.

As such, the Seq Scan total cost was calculated as:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Total cost of Seq Scan  = (estimated sequential page reads \* seq\_page\_cost) + (estimated rows returned \* cpu\_tuple\_cost)  = (7 \* 1) + (1000 \* 0.01)  = 7 + 10.00  = 17.00 |

And for the Aggregate as:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Total cost of Aggregate  = (cost of Seq Scan) + (estimated rows processed \* cpu\_operator\_cost) + (estimated rows returned \* cpu\_tuple\_cost)  = (17.00) + (1000 \* 0.0025) + (1 \* 0.01)  = 17.00 + 2.50 + 0.01  = 19.51 |

## How the planner uses the costs

Since we know Postgres will pick the query plan with the lowest total cost, we can use that to try to understand the choices it has made. For example, if a query is not using an index that you expect it to, you can use settings like **[enable\_seqscan](https://www.postgresql.org/docs/current/runtime-config-query.html" \l "GUC-ENABLE-SEQSCAN" \t "_blank)** to massively discourage certain query plan choices. By this point, you shouldn’t be surprised to hear that settings like this work by increasing the costs!  
Row numbers are an extremely important part of cost estimation. They are used to calculate estimates for different join orders, join algorithms, scan types, and more. Row cost estimates that are out by a lot can lead to cost estimation being out by a lot, which can ultimately result in a suboptimal plan choice being made.

### Using EXPLAIN ANALYZE to get a query plan

When you write SQL statements in PostgreSQL, the ANALYZE command is key to optimizing queries, making them faster and more efficient. In addition to displaying the query plan and PostgreSQL estimates, the EXPLAIN ANALYZE option performs the query (be careful with UPDATE and DELETE!), and shows the actual execution time and row count number for each step in the execution process. This is necessary for monitoring SQL performance.

You can use EXPLAIN ANALYZE to compare the estimated number of rows with the actual rows returned by each operation.

Let’s look at an example, using the same data again:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | QUERY PLAN                                                                                                 |  -----------------------------------------------------------------------------------------------------------+  Sort  (cost=66.83..69.33 rows=1000 width=17) (actual time=20.569..20.684 rows=1000 loops=1)                |    Sort Key: username                                                                                       |    Sort Method: quicksort  Memory: 102kB                                                                    |    ->  Seq Scan on users  (cost=0.00..17.00 rows=1000 width=17) (actual time=0.048..0.596 rows=1000 loops=1)|  Planning Time: 0.171 ms                                                                                    |  Execution Time: 20.793 ms                                                                                  | |

We can see that the total execution cost is still 69.33, with the majority of that being the Sort operation, and 17.00 coming from the Sequential Scan. Note that the query execution time is just under 21ms.

#### **Sequential scan vs. Index Scan**

Now, let’s add an index to try to avoid that costly sort of the entire table:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ​​CREATE INDEX people\_username\_idx ON users (username);    EXPLAIN ANALYZE SELECT \* FROM users ORDER BY username;    QUERY PLAN                                                                                                                       |  ---------------------------------------------------------------------------------------------------------------------------------+  Index Scan using people\_username\_idx on users  (cost=0.28..28.27 rows=1000 width=17)  (actual time=0.052..1.494 rows=1000 loops=1)|  Planning Time: 0.186 ms                                                                                                          |  Execution Time: 1.686 ms                                                                                                         | |

As you can see, the query planner has now chosen an Index Scan, since the total cost of that plan is 28.27 (lower than 69.33). It looks that the index scan was more efficient than the sequential scan, as the query execution time is now just under 2ms.

## Helping the planner estimate more accurately

We can help the planner estimate more accurately in two ways:

1. Help it gather better statistics
2. Tune the constants it uses for the calculations

The statistics can be especially bad after a big change to the data in a table. As such, when loading a lot of data into a table, you can help Postgres out by running a manual ANALYZE on it. Statistics also do not persist over a major version upgrade, so that’s another important time to do this.

Naturally, tables also change over time, so [**tuning the autovacuum settings**](https://www.postgresql.org/docs/current/runtime-config-autovacuum.html) to make sure it runs frequently enough for your workload can be very helpful.

If you’re having trouble with bad estimates for a column with a skewed distribution, you may benefit from increasing the amount of information Postgres gathers by using the [**ALTER TABLE SET STATISTICS**](https://www.postgresql.org/docs/current/sql-altertable.html) command, or even the **[default\_statistics\_target](https://www.postgresql.org/docs/current/runtime-config-query.html" \l "GUC-DEFAULT-STATISTICS-TARGET" \t "_blank)** for the whole database.

Another common cause of bad estimates is that, by default, Postgres will assume that two columns are independent. You can fix this by asking it to gather correlation data on two columns from the same table via [**extended statistics**](https://www.postgresql.org/docs/current/sql-createstatistics.html).

On the constant tuning front, there are a lot of parameters you can tune to suit your hardware. Assuming you’re running on SSDs, you’ll likely at minimum want to tune your setting of random\_page\_cost. This defaults to 4, which is 4x more expensive than the seq\_page\_cost we looked at earlier. This ratio made sense on spinning disks, but on SSDs it tends to penalize random I/O too much. As such a setting closer to 1, or between 1 and 2, might make more sense. At ScaleGrid, we default to 1.

## Can I remove the costs from query plans?

For many of the reasons mentioned above, most people leave the costs on when running EXPLAIN. However, should you wish, you can turn them off using the COSTS parameter.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | EXPLAIN (COSTS OFF) SELECT \* FROM users LIMIT 1;    QUERY PLAN             |  -----------------------+  Limit                  |    ->  Seq Scan on users| |

## Conclusion

To re-cap, the costs in query plans are Postgres’ estimates for how long an SQL query will take, in an arbitrary unit.

It picks the plan with the lowest overall cost, based on some configurable constants and some statistics it has gathered.

Helping it estimate these costs more accurately is very important to help it make good choices, and keep your queries performant.