

# Query Optimization Profiling



## Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

## Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Profiling as a tool for identifying bottlenecks

Selecting a Subtask for Profiling

Profiling Tools

## Profiling

- Selecting Subtasks

- Duration

- execution count

## What Should Be Optimized?

- The larger the subtask's portion of the total execution time, the higher the potential gain.

- One must consider the costs of optimization.

- It's beneficial to approach the task from a broader perspective.

In previous sections, we've explored how queries work, the components of an execution plan, and the factors that influence the selection of a specific plan. This is the most challenging and crucial part. Once you understand the mechanisms, you can analyze any situation that arises using logic and common sense to assess whether the query is executing efficiently and identify ways to improve performance.

But how do you find the query that's worth optimizing?

Generally, addressing any optimization task (not limited to DBMS contexts) begins with profiling, although this term is not always used explicitly. We should break down the task causing issues into subtasks and measure how much of the total time they take. It's also useful to know how often each subtask is executed.

The larger the share of the subtask in the overall runtime, the more significant the performance improvement from optimizing it. In practice, you also have to account for the expected costs of optimization—getting the potential benefit can be challenging.

There are cases where a subtask is executed quickly but frequently (for example, common in queries generated by ORMs). It might be impossible to speed up individual queries, but consider whether the subtask needs to be executed so frequently? Such a question might lead to the need for architectural changes, which, while challenging, can ultimately yield substantial benefits.

## System activity in its entirety

- Beneficial for the administrator to identify resource-intensive tasks
- Monitoring and the `pg_profile` extension

## A specific task drawing criticism

- Helpful in addressing a specific issue
- The more precise, the better: broader coverage blurs the picture.

An overall activity profile can provide valuable insights to a database administrator who is not focused on addressing a specific issue but rather identifying the most resource-intensive tasks, the optimization of which could significantly reduce system load.

The monitoring system should display such a profile.

Another option is to use the `pg_profile` extension. It is built upon PostgreSQL's statistical views, with the data stored in the activity snapshots repository. By analyzing and comparing snapshots, you can identify issues and their underlying causes. The extension was created by Andrei Zubkoin([https://github.com/zubkov-andrei/pg\\_profile](https://github.com/zubkov-andrei/pg_profile)).

Explaining in detail how to build a monitoring system, which tools to use, and which metrics to collect is outside the course's scope, but we recommend Alexey Lesovskiy's "PostgreSQL Monitoring" book:

<https://postgrespro.ru/education/books/monitoring>

To solve a specific problem, you need to build a profile that targets only the actions necessary to reproduce this issue. For instance, if a user reports that "the window opens for a full minute," there's no point in analyzing all database activity during that period—those metrics will include actions unrelated to the window opening.

## Execution Time

- It makes sense for the user
- Highly unstable metric

## Page I/O

- Unaffected by external factors
- Means little to users

What units should be used to measure resources? The most meaningful metric for the end user is response time—the time between pressing a button and receiving a result.

However, from a technical perspective, time isn't always the most practical metric to focus on. It is heavily influenced by numerous external factors, such as cache utilization and current server load. If the issue is analyzed on a test server with different characteristics instead of the main server, it introduces variations in hardware, configurations, and workload profiles.

In this regard, considering I/O—that is, the number of read and written pages—might be more practical. This metric is more stable. It is not affected by hardware specifications or server load, so it will produce consistent results across different servers with identical data and configuration settings.

Therefore, it's crucial to diagnose issues using the full dataset. For example, you can use Database LabEngine — a tool developed by Postgres.ai for quickly creating thin clones: <https://github.com/postgres-ai/database-lab-engine>

I/O typically provides a good indication of the amount of work needed to execute a query, as most of the time is devoted to reading and processing data pages. Although this metric isn't meaningful for the end user.

## Subtasks

- client-side
- application server
- database serverThe problem is often, but not always, right here.
- network

## How to Profile

- It's technically challenging and requires various monitoring tools.
- It's usually not difficult to confirm the assumption.

Response time is meaningful to the user. This means that, generally speaking, the profile should encompass not just the DBMS, but also the client side, the application server, and network data transfer.

Performance problems often stem from the DBMS, as a single inadequately designed query plan can increase the time by orders of magnitude. But that's not always true. The issue could be due to a slow client-server connection, the client application taking too long to process the received data, and other factors.

Unfortunately, obtaining such a comprehensive profile is quite challenging. To achieve this, all components of the information system need to be equipped with monitoring and tracing subsystems that take into account the specific characteristics of this system. However, it's typically straightforward to measure the overall response time (even with a stopwatch) and compare it to the DBMS's total runtime; ensure there are no significant network delays, among other factors. Without this, it's entirely possible we'll end up searching where the light is better, not where the keys were actually lost. The whole point of profiling is to identify what needs optimization.

We will proceed under the assumption that the problem lies specifically with the DBMS.

## PL Profiler Extension

- Third-party extension
- Designed exclusively for PL/pgSQL functions
- Profiling a separate script or session
- Performance report in HTML format, including a flame graph

## The plpgsql\_check extension

- Third-party extension
- Validates PL/pgSQL and embedded SQL
- Enables detection of compilation errors
- Identifying dependent objects within functions
- Support for automatic function profiling

When a user performs an action, multiple queries are typically executed. How do you identify the query that's worth optimizing? This requires a profile that breaks down to the query level.

However, the server-side component of the application, which runs within the DBMS, isn't limited to SQL queries—it can also include procedural code. If the code is written in PL/pgSQL, you can use the external PL Profiler extension for profiling (primary developer: Yan Vik <https://github.com/bigsql/plprofiler>)

The extension allows profiling of separately executed scripts and running sessions. It also includes a call graph in the form of a "flame graph" (flame graph).

You can also recommend the plpgsql\_check extension (primary developer — Pavel Stéchure: [https://github.com/okbob/plpgsql\\_check](https://github.com/okbob/plpgsql_check)) This extension validates SQL identifiers used in PL/pgSQL code, identifies performance issues, and includes a built-in code profiler.

As the course focuses on SQL query optimization, we won't be covering these extensions, but we mention them to provide a complete picture.

## Server message log

Enabled by configuration parameters:

`log_min_duration_statement = 0`— Duration and text of all queries

— Identifying details

Enabling it for a specific session can be challenging

large volume; increasing the time threshold leads to loss of information

Nested queries are not monitored(the `auto_explain` extension can be used)

analysis using external tools, such as pgBadger

PostgreSQL includes two primary built-in tools for profiling executed SQL queries: the server log and statistics.

In the log file, you can enable the logging of query information and their execution time using configuration parameters. The parameter `log_min_duration_statement` is typically used for this, though others exist.

How can you identify the queries in the log that correspond to user actions? It would be useful to enable or disable the parameter for a specific session, but there are no built-in tools for this. You can filter the overall message stream to isolate the relevant entries; this is conveniently achieved by configuring the `log_line_prefix` parameter to include additional identifying information. Connection pooling adds even more complexity to the situation.

To analyze nested queries—when a query invokes a function containing additional queries—the `auto_explain` module is required (<https://postgrespro.ru/docs/postgresql/16/auto-explain>)

The next challenge is analyzing the log. This requires using external tools, with pgBadger being the de facto standard (<https://github.com/darold/pgbadger>)

Of course, you can include your own messages in the log if they are helpful.



## The pg\_stat\_statements extension

- Detailed query information in the view (including nested queries)

- storage size is limited

- Queries are considered the same "except for constants", even if they have different execution plans

- Authentication is limited to the username and database.

- Unified Query Identifier (compute\_query\_id = auto)

The second approach involves using statistics, specifically the pg\_stat\_statements extension

(<https://postgrespro.ru/docs/postgresql/16/pgstatstatements>)

The extension gathers detailed information about executed queries (including I/O page terms) and presents it in the pg\_stat\_statements view.

Since the number of distinct queries can be very high, the storage is constrained by a configuration parameter, retaining only the most frequently executed queries.

In this case, queries are considered identical if they have the same parse tree (up to constants). Keep in mind that these queries may have different execution plans and varying run times.

Unfortunately, there are challenges in identifying queries: they can be associated with a specific user and database, but not with a session.

When the compute\_query\_id configuration parameter is set to auto or on, the PostgreSQL core generates a unique query identifier. It can be logged by setting the log\_line\_prefix parameter, and also used to combine data from the kernel (the pg\_stat\_activity.query\_id column), pg\_stat\_statements, and other extensions.

<https://postgrespro.ru/docs/postgresql/16/runtime-config-statistics#GUC-COMPUTE-QUERY-ID>

## The pg\_stat\_statements Extension

Enable the pg\_stat\_statements extension to profile queries:

```
=> CREATE EXTENSION pg_stat_statements;

CREATE EXTENSION

=> ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';

ALTER SYSTEM
```

Loading the shared library requires a restart.

```
student$ sudo pg_ctlcluster 16 main restart

student$ psql demo
```

Configure the extension to collect information about all queries, including subqueries:

```
=> SET pg_stat_statements.track = 'all';

SET
```

Let's create a view to review the collected statistics on operator execution:

```
=> CREATE VIEW statements_v AS
SELECT
    queryid,
    toplevel,
    substring(regexp_replace(query, ' +', ' ', 'g') FOR 55) AS query,
    calls,
    round(total_exec_time)/1000 AS time_sec,
    shared_blks_hit + shared_blks_read + shared_blks_written AS shared_blks
FROM pg_stat_statements
ORDER BY total_exec_time DESC;

CREATE VIEW
```

For simplicity's sake, we display only some fields and show the total cache page count.

---

## Execution Profile

Let's examine the problem. The goal is to create a report that generates a pivot table of passenger counts, with aircraft models as rows and service categories as columns.

First, let's create a function that returns the number of passengers for a specified model and service category:

```
=> CREATE FUNCTION qty(aircraft_code char, fare_conditions varchar)
RETURNS bigint AS $$
SELECT count(*)
FROM flights f
    JOIN boarding_passes bp ON bp.flight_id = f.flight_id
    JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
WHERE f.aircraft_code = qty.aircraft_code AND s.fare_conditions = qty.fare_conditions;
$$ STABLE LANGUAGE sql;

CREATE FUNCTION
```

Let's create a function for the report that returns a set of rows.

```
=> CREATE FUNCTION report()
RETURNS TABLE(model text, economy bigint, comfort bigint, business bigint)
AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model LOOP
        report.model := r.model;
        report.economy := qty(r.aircraft_code, 'Economy');
        report.comfort := qty(r.aircraft_code, 'Comfort');
        report.business := qty(r.aircraft_code, 'Business');
        RETURN NEXT;
    END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Now, let's examine the proposed implementation. Let's reset the statistics:

```
=> SELECT pg_stat_statements_reset();
```

```
pg_stat_statements_reset
-----
(1 row)
```

The last statistics reset date is available in the pg\_stat\_statements\_info view:

```
=> SELECT stats_reset FROM pg_stat_statements_info;
```

```
stats_reset
-----
2025-10-18 22:41:02.78494+03
(1 row)
```

Run the report() function:

```
=> SELECT * FROM report();
```

model	economy	comfort	business
Аэробус А319-100	337129	0	70232
Аэробус А320-200	0	0	0
Аэробус А321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

Let's examine the statistics we've gathered:

```
=> SELECT * FROM statements_v \gx
```

-[ RECORD 1 ]-----	
queryid	1399367483717482784
toplevel	t
query	SELECT * FROM report()
calls	1
time_sec	30.918
shared_blks	919428
-[ RECORD 2 ]-----	
queryid	-6704952473219487249
toplevel	f
query	SELECT count(*) FROM flights f JOIN boarding_passes
calls	27
time_sec	30.891
shared_blks	918912
-[ RECORD 3 ]-----	
queryid	1651686694542252645
toplevel	f
query	SELECT a.aircraft_code, a.model FROM aircrafts a ORDER
calls	1
time_sec	0.003
shared_blks	19
-[ RECORD 4 ]-----	
queryid	-7827778417405733903
toplevel	t
query	SELECT pg_stat_statements_reset()
calls	1
time_sec	0
shared_blks	0
-[ RECORD 5 ]-----	
queryid	-7737098410881999591
toplevel	t
query	SELECT stats_reset FROM pg_stat_statements_info
calls	1
time_sec	0
shared_blks	0

- The first is the main query, which is detailed below. Toplevel indicates that the query is executed at the top level.
- The second query, from the qty function, was executed 27 times.
- The third query is the one the loop operates on.

---

Note that the query identifier in pg\_stat\_statements matches the system identifier because of the compute\_query\_id setting:

=> **EXPLAIN** (verbose)

**SELECT** pg\_stat\_statements\_reset();

QUERY PLAN

---

Result (cost=0.00..0.01 rows=1 width=4)

Output: pg\_stat\_statements\_reset('0'::oid, '0'::oid, '0'::bigint)

Query Identifier: -7827778417405733903

(3 rows)

# Single Query Profile



## EXPLAIN ANALYZE

- subtasks correspond to plan nodes
- Duration – actual time or page I/O – buffers
- Execution count – loops

## Features

- Besides the most resource-intensive nodes, optimization candidates are those with a large cardinality estimation error.
- Any change may result in a complete overhaul of the execution plan
- Sometimes you have to settle for a basic EXPLAIN

11

Either way, we identify the query to optimize from among the executed ones. How to Work with the Query Itself? The EXPLAIN ANALYZE command also provides a detailed execution profile.

The subtasks of this profile are the plan's nodes (the plan isn't a flat list but a tree). The execution duration of a node and the number of its repetitions will display the "time" and "loops" values from the actual section. Using the buffers parameter, you can get the I/O volume (and the time spent on I/O operations if the track\_io\_timin parameter is enabled).

The execution plan also contains critical information about the optimizer's expected cardinality for each step. Typically, if there's no significant error in the cardinality estimate, the plan will be appropriate (if not, you should adjust the global settings). Therefore, you should focus not only on the most resource-intensive nodes but also on those with a substantial (an order of magnitude or more) difference between the estimated "rows" and actual "rows". Examine the most deeply nested problematic node, as the error will propagate upward through the tree from there.

There are cases where a query runs so long that EXPLAIN ANALYZE can't be executed. In such cases, you'll have to make do with a basic EXPLAIN and try to determine the cause of inefficiency without complete information.

Working with large execution plans in text format isn't always convenient. For better clarity, consider using third-party tools, such as <https://explain.tensor.ru/about>.

## Single Query Profile

Let's now generate the same report with a single query and analyze its execution using EXPLAIN with the analyze and buffers options, so the execution plan displays the actual number of data and temporary file pages read and written:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
WITH t AS (
  SELECT f.aircraft_code,
         count(*) FILTER (WHERE s.fare_conditions = 'Economy') economy,
         count(*) FILTER (WHERE s.fare_conditions = 'Comfort') comfort,
         count(*) FILTER (WHERE s.fare_conditions = 'Business') business
  FROM flights f
       JOIN boarding_passes bp ON bp.flight_id = f.flight_id
       JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
  GROUP BY f.aircraft_code
)
SELECT a.model,
       coalesce(t.economy,0) economy,
       coalesce(t.comfort,0) comfort,
       coalesce(t.business,0) business
FROM aircrafts a
LEFT JOIN t ON a.aircraft_code = t.aircraft_code
ORDER BY a.model;
```

## QUERY PLAN

```

-----
Sort (actual rows=9 loops=1)
  Sort Key: ((ml.model ->> lang()))
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=3114 read=57798, temp read=13666 written=13666
  -> Hash Left Join (actual rows=9 loops=1)
    Hash Cond: (ml.aircraft_code = t.aircraft_code)
    Buffers: shared hit=3114 read=57798, temp read=13666 written=13666
    -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
      Buffers: shared hit=1
    -> Hash (actual rows=8 loops=1)
      Buckets: 1024  Batches: 1  Memory Usage: 9kB
      Buffers: shared hit=3113 read=57798, temp read=13666 written=13666
      -> Subquery Scan on t (actual rows=8 loops=1)
        Buffers: shared hit=3113 read=57798, temp read=13666 written=13666
        -> HashAggregate (actual rows=8 loops=1)
          Group Key: f.aircraft_code
          Batches: 1  Memory Usage: 24kB
          Buffers: shared hit=3113 read=57798, temp read=13666
written=13666
          -> Hash Join (actual rows=7925812 loops=1)
            Hash Cond: ((f.aircraft_code = s.aircraft_code) AND
((bp.seat_no)::text = (s.seat_no)::text))
            Buffers: shared hit=3113 read=57798, temp read=13666
written=13666
            -> Hash Join (actual rows=7925812 loops=1)
              Hash Cond: (bp.flight_id = f.flight_id)
              Buffers: shared hit=3105 read=57798, temp
read=13666 written=13666
            -> Seq Scan on boarding_passes bp (actual
rows=7925812 loops=1)
              Buffers: shared hit=481 read=57798
            -> Hash (actual rows=214867 loops=1)
              Buckets: 262144  Batches: 2  Memory Usage:
6256kB
              Buffers: shared hit=2624, temp written=366
              -> Seq Scan on flights f (actual
rows=214867 loops=1)
                Buffers: shared hit=2624
            -> Hash (actual rows=1339 loops=1)
              Buckets: 2048  Batches: 1  Memory Usage: 79kB
              Buffers: shared hit=8
              -> Seq Scan on seats s (actual rows=1339 loops=1)
                Buffers: shared hit=8

Planning:
  Buffers: shared hit=29
Planning Time: 1.782 ms
Execution Time: 11907.954 ms
(40 rows)

```

Notice how the query execution time was reduced and how many fewer pages were read as a result of removing redundant reads (top line in the Buffers section).

You can calculate the total number of pages across all the tables involved:

```

=> SELECT sum(relpages)
FROM pg_class
WHERE relname IN ('flights', 'boarding_passes', 'aircrafts', 'seats');

sum
-----
60911
(1 row)

```

This number can act as a rough estimate for queries requiring all rows: processing many more pages than expected may suggest that the data is being scanned multiple times.

It's clear that the plan is nearly optimal in this case. It can still be optimized, but not as drastically (a clear limitation is the lack of random access memory for hash joins).

Profiling is used to identify queries needing optimization.

The available profiling tools depend on the specific task

server Message Log FROM pg\_stat\_statements

EXPLAIN ANALYZE



1. Run the first version of the report displayed during the demonstration, and ensure that query text and execution time are recorded in the log file.
2. Check what information was recorded in the log file.
3. Repeat the previous steps by including the `auto_explain` extension with nested query output.

## 1. Report 1. Output

```
=> CREATE FUNCTION qty(aircraft_code char, fare_conditions varchar)
RETURNS bigint AS $$
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
WHERE f.aircraft_code = qty.aircraft_code AND s.fare_conditions = qty.fare_conditions;
$$ STABLE LANGUAGE sql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION report()
RETURNS TABLE(model text, economy bigint, comfort bigint, business bigint)
AS $$
DECLARE
    r record;
BEGIN
FOR r IN SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model LOOP
    report.model := r.model;
    report.economy := qty(r.aircraft_code, 'Economy');
    report.comfort := qty(r.aircraft_code, 'Comfort');
    report.business := qty(r.aircraft_code, 'Business');
RETURN NEXT;
END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Include SQL statement output and their run time in the log:

```
=> SET log_min_duration_statement = 0;
```

SET

```
=> SELECT * FROM report();
```

model	economy	comfort	business
Аэробус A319-100	337129	0	70232
Аэробус A320-200	0	0	0
Аэробус A321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

## 2. Log Entries

```
student$ tail -n 1 /var/log/postgresql/postgresql-16-main.log
```

```
2025-10-18 23:08:14.909 MSK [128153] postgres@demo LOG: duration: 45341.760 ms
statement: SELECT * FROM report();
```

Nested SQL statements are not output.

### The auto\_explain Extension

```
=> LOAD 'auto_explain';
```

LOAD

```
=> RESET log_min_duration_statement;
```

RESET

```
=> SET compute_query_id = on; -- to output query identifiers
```

SET

```
=> SET auto_explain.log_min_duration = 0;
```

SET

=> SET auto\_explain.log\_nested\_statements = on;

SET

=> SET auto\_explain.log\_verbose = on;

SET

=> SELECT \* FROM report();

model	economy	comfort	business
Аэробус A319-100	337129	0	70232
Аэробус A320-200	0	0	0
Аэробус A321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

Let's display the last few lines of the message log:

student\$ tail -n 50 /var/log/postgresql/postgresql-16-main.log

```
JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
WHERE f.aircraft_code = qty.aircraft_code AND s.fare_conditions =
qty.fare_conditions;

Query Parameters: $1 = 'SU9', $2 = 'Business'
Finalize Aggregate (cost=106686.64..106686.65 rows=1 width=8)
Output: count(*)
-> Gather (cost=106686.42..106686.63 rows=2 width=8)
Output: (PARTIAL count(*))
Workers Planned: 2
-> Partial Aggregate (cost=105686.42..105686.43 rows=1 width=8)
Output: PARTIAL count(*)
-> Hash Join (cost=4417.66..105491.55 rows=77950 width=0)
Inner Unique: true
Hash Cond: ((bp.seat_no)::text = (s.seat_no)::text)
-> Parallel Hash Join (cost=4401.39..104374.33 rows=412804
width=7)
Output: f.aircraft_code, bp.seat_no
Inner Unique: true
Hash Cond: (bp.flight_id = f.flight_id)
-> Parallel Seq Scan on bookings.boarding_passes bp
(cost=0.00..91303.77 rows=3302477 width=7)
Output: bp.ticket_no, bp.flight_id,
bp.boarding_no, bp.seat_no
-> Parallel Hash (cost=4203.90..4203.90 rows=15799
width=8)
Output: f.flight_id, f.aircraft_code
-> Parallel Seq Scan on bookings.flights f
(cost=0.00..4203.90 rows=15799 width=8)
Output: f.flight_id, f.aircraft_code
Filter: (f.aircraft_code = $1)
-> Hash (cost=15.64..15.64 rows=50 width=7)
Output: s.aircraft_code, s.seat_no
-> Bitmap Heap Scan on bookings.seats s
(cost=5.41..15.64 rows=50 width=7)
Output: s.aircraft_code, s.seat_no
Recheck Cond: (s.aircraft_code = $1)
Filter: ((s.fare_conditions)::text = ($2)::text)
-> Bitmap Index Scan on seats_pkey
(cost=0.00..5.39 rows=149 width=0)
Index Cond: (s.aircraft_code = $1)

Query Identifier: -6704952473219487249
2025-10-18 23:09:02.520 MSK [128153] postgres@demo CONTEXT: SQL function "qty" statement
1
PL/pgSQL function report() line 9 at assignment
2025-10-18 23:09:02.531 MSK [128153] postgres@demo LOG: duration: 0.665 ms plan:
Query Text: SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model
Sort (cost=3.51..3.53 rows=9 width=36)
Output: ml.aircraft_code, ((ml.model ->> lang()))
Sort Key: ((ml.model ->> lang()))
-> Seq Scan on bookings.aircrafts_data ml (cost=0.00..3.36 rows=9 width=36)
Output: ml.aircraft_code, (ml.model ->> lang())
```

```
2025-10-18 23:09:02.531 MSK [128153] postgres@demo CONTEXT: PL/pgSQL function report()  
line 5 at FOR over SELECT rows  
2025-10-18 23:09:02.532 MSK [128153] postgres@demo LOG: duration: 46984.392 ms plan:  
Query Text: SELECT * FROM report();  
Function Scan on bookings.report (cost=0.25..10.25 rows=1000 width=56)  
Output: model, economy, comfort, business  
Function Call: report()  
Query Identifier: 5264940823427202119
```

Logs capture nested queries and execution plans — this is the extension's primary function.

In addition, with the specified settings, query parameters and the query identifier are included in the log entries.