



Optimizing PostgreSQL query performance

AWS Prescriptive Guidance



AWS Prescriptive Guidance: Optimizing PostgreSQL query performance

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Services or capabilities described in AWS documentation might vary by Region. To see the differences applicable to the AWS European Sovereign Cloud Region, see the [AWS European Sovereign Cloud User Guide](#).

Table of Contents

Introduction

Use cases for query performance tuning

EXPLAIN plan

The EXPLAIN statement

Using EXPLAIN ANALYZE

How to read the EXPLAIN query plan

.....

Collations

Data type mismatch

Function call in SELECT

IN or EXISTS

Subqueries or CTEs

FAQ

What is EXPLAIN?

What is EXPLAIN ANALYZE?

What is collation in PostgreSQL?

What is a CTE?

What are the categories of functions in PostgreSQL?

.....

Contributors

Document history

1

1

2

2

2

2

4

13

16

18

19

22

26

26

26

27

27

27

29

30

31

Optimizing PostgreSQL query performance

Amazon Web Services ([contributors](#))

April 2024 ([document history](#))

PostgreSQL is an open source object-relational database system that is powerful, flexible, and reliable. There are many ways to optimize the performance of a PostgreSQL query. The process of optimizing the query depends on the use case. Knowing the current query plan can help you to identify and understand any issues and make the necessary changes. Sometimes, you might need to analyze the tables to keep the database statistics up to date. The PostgreSQL optimizer will use those statistics to run the query faster. This guide focuses on best practices for improving the performance of PostgreSQL queries.

This guide assumes that you have an existing Amazon Relational Database Service (Amazon RDS) for PostgreSQL or Amazon Aurora PostgreSQL-Compatible database instance.

Use cases for query performance tuning

This guide covers five use cases, with explanations and examples:

- Collations
- Data type mismatch
- Function call in the SELECT statement
- IN or EXISTS
- Subqueries or Common Table Expressions (CTEs)

Each use case provides details of the initial run plan, how to analyze the plan to identify the problem, and a solution. Implementing these use cases typically results in faster response times for queries, reduced load on the server, and overall enhanced system efficiency. Those improvements can lead to a better user experience and increased system reliability.

The EXPLAIN query plan

PostgreSQL provides the `EXPLAIN` and `EXPLAIN ANALYZE` options for returning query plans with details about how the query will be run.

The EXPLAIN statement

The `EXPLAIN` statement returns the query plan that the PostgreSQL planner generates for a given statement. The query plan shows the following:

- How tables involved in a statement will be scanned (for example, by index scan or sequential scan)
- How multiple tables will be joined (for example, hash join, merge join, or nested loop join)

Understanding the plan is critical when improving the performance of the query. After you understand the plan, you can focus on where the query is taking too long and take action to reduce the time.

Using EXPLAIN ANALYZE

In PostgreSQL, `EXPLAIN` will only generate a plan for the given statement. If you add the `ANALYZE` keyword, `EXPLAIN` will return the plan, run the query, and show the actual runtime and row count for each step. This is indispensable for analyzing the query performance.

Important

When using `EXPLAIN ANALYZE`, be careful with `INSERT`, `UPDATE`, and `DELETE`.

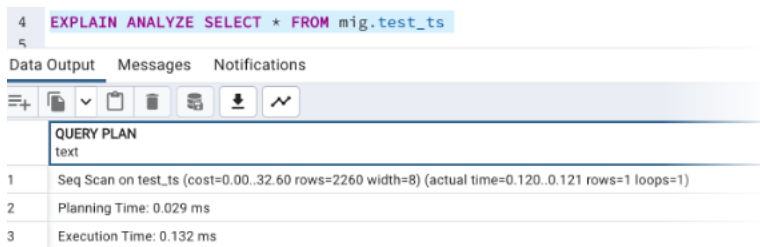
How to read the EXPLAIN query plan

A PostgreSQL query plan is a tree structure consisting of several nodes. The `EXPLAIN` query plan shows the steps that the database engine uses to run a query. The query plan provides the following information:

- The type of operations performed, such as sequential scans, index scans, or nested loop joins.

- A label, such as Seq Scan, Index Scan, or Nested Loop, to describe the operation being performed.
- The name of the table or index being processed by the query.
- Cost and row columns with information about the estimated cost in an arbitrary unit of computation and the number of rows processed.
- The filter condition of any filter applied on the operation, such as the where condition.
- A visual representation of the steps, with each operation shown as a node and arrows connecting the operations. The order of the operations is shown from left to right, with earlier operations feeding into later operations.

The following screenshot shows the query plan for a sequential scan.



The screenshot shows a PostgreSQL query window with the command `EXPLAIN ANALYZE SELECT * FROM mig.test_ts` entered. Below the command bar, there are tabs for 'Data Output', 'Messages', and 'Notifications'. The 'Data Output' tab is active, showing a 'QUERY PLAN' section. The query plan is displayed as a table with three rows:

	QUERY PLAN
1	Seq Scan on test_ts (cost=0.00..32.60 rows=2260 width=8) (actual time=0.120..0.121 rows=1 loops=1)
2	Planning Time: 0.029 ms
3	Execution Time: 0.132 ms

The cost estimate (`cost=0.00..32.60 rows=2260 width=8`) means that PostgreSQL expects that the query will require 32.60 units of computation to return results.

The `0.00` value is the cost at which this node can begin working (in this case, startup time for the query). The `rows` value is the estimated number of rows that the sequential scan will return. The `width` value is the estimated size in bytes of the returned rows.

Because the example shows EXPLAIN with the ANALYZE option, the query was run, and the timing information was captured. The result (`actual time=0.120..0.121 rows=1 loops=1`) means the following:

- The sequential scan was run one time (the `loops` value).
- The scan returned one row.
- The actual time was 0.12 milliseconds.

Use cases for tuning queries

This guide covers the following use cases for tuning query performance:

- Collations
- Data type mismatch
- Function call in the SELECT statement
- IN or EXISTS
- Subqueries or Common Table Expressions (CTEs)

To test performance tuning for these query-performance use cases, use your existing database and the example data provided by this guide. The example uses data for a fictitious XX airline. To prepare the example data, run the following example code:

```
--Creating required tables along with data.

--Creating user and schema
create user perf_user;
create schema perf_user AUTHORIZATION perf_user;
set search_path to perf_user;

--Table1:

CREATE TABLE IF NOT EXISTS perf_user.rnr_expiry_date
(
    airline_iata_code character(2) COLLATE pg_catalog."default",
    pnr_number character varying(15) COLLATE pg_catalog."default" NOT NULL,
    calculated_pnr_expiry_date timestamp(0) without time zone,
    row_num bigint,
    arc_expiry_date timestamp(0) without time zone,
    status character varying(10) COLLATE pg_catalog."default"
);

insert into perf_user.rnr_expiry_date
select 'XX' , upper(substring(concat(md5(random()::text), md5(random()::text)), 0,
7)), '2023-01-01 00:00:00', generate_series(1,100000), '2023-02-02 00:00:00' ,null;
```

```

CREATE INDEX rnr_expiry_date_idx1 ON perf_user.rnr_expiry_date (row_num ASC NULLS
LAST);

CREATE INDEX rnr_expiry_date_idx2 ON perf_user.rnr_expiry_date (airline_iata_code
COLLATE pg_catalog."default" ASC NULLS LAST, pnr_number COLLATE pg_catalog."default"
ASC NULLS LAST);

CREATE INDEX rnr_expiry_date_idx3 ON perf_user.rnr_expiry_date (pnr_number ASC NULLS
LAST);

vacuum analyze perf_user.rnr_expiry_date;

-----
--Table2:

CREATE TABLE IF NOT EXISTS perf_user.rnr_segment_pax
(
    airline_iata_code character varying(6) COLLATE pg_catalog."default" NOT NULL,
    pnr_number character varying(15) COLLATE pg_catalog."default" NOT NULL,
    segment_pax_id numeric(25,0) NOT NULL,
    oandd_id numeric(25,0) NOT NULL,
    segment_id numeric(25,0) NOT NULL,
    cabin_class character varying(15) COLLATE pg_catalog."default",
    pax_id numeric(25,0) NOT NULL,
    ticket_number character varying(25) COLLATE pg_catalog."default",
    ticket_type character varying(10) COLLATE pg_catalog."default",
    archive_status smallint NOT NULL DEFAULT (0)::smallint,
    certificate_number character varying(100) COLLATE pg_catalog."default",
    loyalty_number character varying(25) COLLATE pg_catalog."default",
    arc_expiry_date timestamp(0) without time zone,
    CONSTRAINT rnr_segment_pax_pk PRIMARY KEY (airline_iata_code, pnr_number,
segment_id, pax_id),
    CONSTRAINT rnr_segment_pax_ck1 CHECK (ticket_type::text = ANY (ARRAY['E'::character
varying::text, 'A'::character varying::text, 'C'::character varying::text,
'M'::character varying::text, 'I'::character varying::text]))
);

insert into perf_user.rnr_segment_pax (airline_iata_code, pnr_number, segment_pax_id,
oandd_id, segment_id, pax_id, ticket_type, arc_expiry_date )
select 'XX',upper(substring(concat(md5(random()::text), md5(random()::text)), 0, 7)),
generate_series(1,10000000),generate_series(1,10000000),
generate_series(1,10000000),generate_series(1,10000000),'A','2023-01-01 00:00:00';

```

```
insert into perf_user.rnr_segment_pax (airline_iata_code, pnr_number, segment_pax_id,
  oandd_id, segment_id, pax_id, ticket_type, arc_expiry_date )
select 'XX',upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(10000001,20000000),generate_series(10000001,20000000),
generate_series(10000001,20000000),generate_series(10000001,20000000),'I','2023-01-01
00:00:00';
```

```
insert into perf_user.rnr_segment_pax (airline_iata_code, pnr_number, segment_pax_id,
  oandd_id, segment_id, pax_id, ticket_type, arc_expiry_date)
select 'XX',upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(20000001,30000000),generate_series(20000001,30000000),
generate_series(20000001,30000000),generate_series(20000001,30000000),'E','2023-01-01
00:00:00';
```

```
insert into perf_user.rnr_segment_pax (airline_iata_code, pnr_number, segment_pax_id,
  oandd_id, segment_id, pax_id, ticket_type, arc_expiry_date)
select 'XX',upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(30000001,40000000),generate_series(30000001,40000000),
generate_series(30000001,40000000),generate_series(30000001,40000000),'M','2023-01-01
00:00:00';
```

```
CREATE INDEX rnr_segment_pax_idx1
  ON perf_user.rnr_segment_pax USING btree
  (loyalty_number COLLATE pg_catalog."default" ASC NULLS LAST, airline_iata_code
  COLLATE pg_catalog."default" ASC NULLS LAST, arc_expiry_date ASC NULLS LAST);
```

```
CREATE INDEX IF NOT EXISTS rnr_segment_pax_pn_idx1
  ON perf_user.rnr_segment_pax USING btree
  (pnr_number COLLATE pg_catalog."default" ASC NULLS LAST);
```

```
CREATE INDEX IF NOT EXISTS rnr_segment_pax_seq_idx1
  ON perf_user.rnr_segment_pax USING btree
  (segment_id ASC NULLS LAST);
```

```
vacuum analyze perf_user.rnr_segment_pax;
```

```
-----
```

```
--Table3:
```

```
CREATE TABLE IF NOT EXISTS perf_user.rnr_segment
(
  airline_iata_code character varying(6) COLLATE pg_catalog."default" NOT NULL,
```

```

pnr_number character varying(15) COLLATE pg_catalog."C" NOT NULL,
segment_id numeric(25,0) NOT NULL,
oandd_id numeric(25,0),
price_id numeric(25,0),
flight_carrier character varying(6) COLLATE pg_catalog."default" ,
flight_number integer ,
flight_suffix character varying(1) COLLATE pg_catalog."default" ,
flight_date_ltc timestamp(0) without time zone ,
airline_company_code character varying(6) COLLATE pg_catalog."default",
bd_airport_code character varying(5) COLLATE pg_catalog."default" ,
off_airport_code character varying(5) COLLATE pg_catalog."default" ,
segment_status character varying(50) COLLATE pg_catalog."default" ,
flight_status character varying(30) COLLATE pg_catalog."default",
flight_type character varying(15) COLLATE pg_catalog."default",
cabin_class character varying(15) COLLATE pg_catalog."default",
arc_expiry_date timestamp(0) without time zone,
oandd_dep_date_ltc timestamp(0) without time zone,
added_time timestamp(6) without time zone,
dep_date_ltc timestamp(0) without time zone ,
arr_date_utc timestamp(0) without time zone,
dep_date_utc timestamp(0) without time zone,
origin character varying(5) COLLATE pg_catalog."default",
destination character varying(5) COLLATE pg_catalog."default",
CONSTRAINT rnr_segment_pk PRIMARY KEY (pnr_number, segment_id, airline_iata_code)
);

```

```

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(1,10000000),'XX',110,'*', '2023-01-01 00:00:00';

```

```

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(10000001,20000000),'XX',120,'*', '2023-01-01 00:00:00';

```

```

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text)), 0, 7)),
generate_series(20000001,30000000),'XX',130,'*', '2023-01-01 00:00:00';

```

```

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text), 0, 7)),
generate_series(30000001,40000000),'XX',140,'*', '2023-01-01 00:00:00';

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text), 0, 7)),
generate_series(40000001,50000000),'XX',150,'*', '2023-01-01 00:00:00';

insert into perf_user.rnr_segment (airline_iata_code, pnr_number, segment_id,
    FLIGHT_CARRIER, FLIGHT_NUMBER, FLIGHT_SUFFIX, FLIGHT_DATE_LTC)
select 'XX',
upper(substring(concat(md5(random())::text), md5(random())::text), 0, 7)),
generate_series(50000001,60000000),'XX',160,'*', '2023-01-01 00:00:00';

CREATE INDEX rnr_segment_idx1 ON perf_user.rnr_segment USING btree
    (flight_date_ltc ASC NULLS LAST, bd_airport_code COLLATE pg_catalog."default"
    ASC NULLS LAST, off_airport_code COLLATE pg_catalog."default" ASC NULLS LAST,
    flight_number ASC NULLS LAST, flight_carrier COLLATE pg_catalog."default" ASC NULLS
    LAST, flight_suffix COLLATE pg_catalog."default" ASC NULLS LAST, airline_iata_code
    COLLATE pg_catalog."default" ASC NULLS LAST, arc_expiry_date ASC NULLS LAST);

CREATE INDEX rnr_segment_idx2
    ON perf_user.rnr_segment USING btree
    (dep_date_ltc ASC NULLS LAST, flight_number ASC NULLS LAST, bd_airport_code COLLATE
    pg_catalog."default" ASC NULLS LAST, off_airport_code COLLATE pg_catalog."default" ASC
    NULLS LAST, flight_carrier COLLATE pg_catalog."default" ASC NULLS LAST, flight_suffix
    COLLATE pg_catalog."default" ASC NULLS LAST, arc_expiry_date ASC NULLS LAST);

CREATE INDEX rnr_segment_idx3
    ON perf_user.rnr_segment USING btree
    (pnr_number COLLATE pg_catalog."default" ASC NULLS LAST, arr_date_utc ASC NULLS
    LAST, airline_iata_code COLLATE pg_catalog."default" ASC NULLS LAST, arc_expiry_date
    ASC NULLS LAST);

CREATE INDEX rnr_segment_idx4
    ON perf_user.rnr_segment USING btree
    (dep_date_utc ASC NULLS LAST, added_time ASC NULLS LAST, airline_iata_code COLLATE
    pg_catalog."default" ASC NULLS LAST, arc_expiry_date ASC NULLS LAST);

```

```

CREATE INDEX rnr_segment_idx5
    ON perf_user.rnr_segment USING btree
    (origin COLLATE pg_catalog."default" ASC NULLS LAST, destination COLLATE
pg_catalog."default" ASC NULLS LAST, oandd_dep_date_ltc ASC NULLS LAST,
airline_iata_code COLLATE pg_catalog."default" ASC NULLS LAST, arc_expiry_date ASC
NULLS LAST);

CREATE INDEX rnr_segment_idx6
    ON perf_user.rnr_segment USING btree
    (pnr_number COLLATE pg_catalog."default" ASC NULLS LAST, oandd_id ASC NULLS LAST,
segment_id ASC NULLS LAST, airline_iata_code COLLATE pg_catalog."default" ASC NULLS
LAST, arc_expiry_date ASC NULLS LAST);

vacuum analyze perf_user.rnr_segment;

-----

--Table4:

CREATE TABLE IF NOT EXISTS perf_user.rnr_seat_numbers
(
    airline_iata_code character varying(6) COLLATE pg_catalog."default" NOT NULL,
    pnr_number character varying(15) COLLATE pg_catalog."default" NOT NULL,
    segment_id numeric(25,0) NOT NULL,
    pax_id numeric(25,0) NOT NULL,
    seat_id numeric(25,0) NOT NULL,
    bd_airport_code character varying(5) COLLATE pg_catalog."default",
    off_airport_code character varying(5) COLLATE pg_catalog."default",
    seat_number character varying(5) COLLATE pg_catalog."default",
    seat_status character varying(20) COLLATE pg_catalog."default",
    ssr_id character varying(100) COLLATE pg_catalog."default",
    archive_status smallint DEFAULT (0)::smallint,
    seat_alloc_id numeric(25,0),
    archive_date timestamp(0) without time zone,
    seat_attribute_code character varying(201) COLLATE pg_catalog."default",
    channel_code character varying(20) COLLATE pg_catalog."default",
    arc_expiry_date timestamp(0) without time zone,
    CONSTRAINT rnr_seat_numbers_pk PRIMARY KEY (pnr_number, segment_id, pax_id,
seat_id, airline_iata_code)
);

```

```

insert into perf_user.rnr_seat_numbers (pnr_number, segment_id, pax_id, seat_id,
    airline_iata_code)
select upper(substring(concat(md5(random()::text), md5(random()::text)), 0, 7)),
generate_series(1,10000000),generate_series(1,10000000),generate_series(1,10000000),'XX';

insert into perf_user.rnr_seat_numbers (pnr_number, segment_id, pax_id, seat_id,
    airline_iata_code)
select upper(substring(concat(md5(random()::text), md5(random()::text)), 0, 7)),
generate_series(100000001,200000000),generate_series(100000001,200000000),generate_series(100000001,
200000000),'XX';

insert into perf_user.rnr_seat_numbers (pnr_number, segment_id, pax_id, seat_id,
    airline_iata_code)
select upper(substring(concat(md5(random()::text), md5(random()::text)), 0, 7)),
generate_series(200000001,300000000),generate_series(200000001,300000000),generate_series(200000001,
300000000),'XX';

insert into perf_user.rnr_seat_numbers (pnr_number, segment_id, pax_id, seat_id,
    airline_iata_code)
select upper(substring(concat(md5(random()::text), md5(random()::text)), 0, 7)),
generate_series(300000001,400000000),generate_series(300000001,400000000),generate_series(300000001,
400000000),'XX';

vacuum Analyze perf_user.rnr_seat_numbers;

--Table5:
CREATE TABLE IF NOT EXISTS perf_user.test_veh
(
    test_veh_id bigint NOT NULL,
    oiltype_id bigint,
    vehicle_id character varying(50) COLLATE pg_catalog."default",
    serviceprogram_id character varying(100) COLLATE pg_catalog."default",
    startdate timestamp without time zone,
    enddate timestamp without time zone,
    last_update_dt timestamp without time zone,
    CONSTRAINT test_veh_pkey PRIMARY KEY (test_veh_id),
    CONSTRAINT test_veh_oiltype_id_fkey FOREIGN KEY (oiltype_id)
        REFERENCES perf_user.oiltype (oiltype_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT test_veh_oiltype_id_fkey1 FOREIGN KEY (oiltype_id)
        REFERENCES perf_user.oiltype (oiltype_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
);

CREATE INDEX IF NOT EXISTS test_veh_enddate_ind

```

```
ON perf_user.test_veh USING btree
(enddate ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS test_veh_oiltype_id_ind
ON perf_user.test_veh USING btree
(oiltype_id ASC NULLS LAST);

--Table6:
CREATE TABLE IF NOT EXISTS perf_user.oiltype
(
    oiltype_id bigint NOT NULL,
    descr character varying(50) COLLATE pg_catalog."default",
    CONSTRAINT oiltype_pkey PRIMARY KEY (oiltype_id)
);

CREATE INDEX IF NOT EXISTS oiltype_oiltyp_in
ON perf_user.oiltype USING btree
(oiltype_id ASC NULLS LAST);

--Table7:
CREATE TABLE IF NOT EXISTS perf_user.serviceprogram
(
    serial bigint NOT NULL,
    serviceprogram_id character varying(50) COLLATE pg_catalog."default",
    proname character varying(150) COLLATE pg_catalog."default",
    CONSTRAINT serviceprogram_pkey PRIMARY KEY (serial)
);

CREATE INDEX IF NOT EXISTS proname_id_ind
ON perf_user.serviceprogram USING btree
(proname COLLATE pg_catalog."default" ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS serviceprogram_id_ind
ON perf_user.serviceprogram USING btree
(serviceprogram_id COLLATE pg_catalog."default" ASC NULLS LAST);

--Table8:
CREATE TABLE IF NOT EXISTS perf_user.vehicleservicehistory
(
    v_id bigint NOT NULL,
    test_veh_id bigint,
```

```

desc_1 character varying(50) COLLATE pg_catalog."default",
start_date timestamp without time zone,
end_date timestamp without time zone,
CONSTRAINT vehicleservicehistory_pkey PRIMARY KEY (v_id)
);

CREATE INDEX IF NOT EXISTS veh_end_date_id_ind
ON perf_user.vehicleservicehistory USING btree
(end_date ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS veh_ser_ind
ON perf_user.vehicleservicehistory USING btree
(test_veh_id ASC NULLS LAST);

CREATE INDEX IF NOT EXISTS vehicleservicehistory_v_id_ind
ON perf_user.vehicleservicehistory USING btree
(test_veh_id ASC NULLS LAST);

--Function creation
CREATE OR REPLACE FUNCTION perf_user.return_data()
RETURNS character varying
LANGUAGE 'plpgsql'
COST 100
VOLATILE PARALLEL UNSAFE
AS $BODY$
BEGIN
return 'EE9F41' ;
END;
$BODY$;
-----
CREATE TABLE IF NOT EXISTS ITEM_DETAILS
(
ITEMID INTEGER,
ORDID INTEGER,
ITEMNAME CHARACTER VARYING(200)
);

CREATE TABLE IF NOT EXISTS ORDER_DETAILS
(
ORDID INTEGER,
ORDNAME CHARACTER VARYING(200),
ORDEREDPLACE CHARACTER VARYING(55)
);

```

```
CREATE TABLE IF NOT EXISTS PAYMENT_DETAILS
(
    PAYID INTEGER,
    ORDID INTEGER,
    PAYPLACE CHARACTER VARYING(55)
);
```

Use case 1 – Collations

In a database, a collation is a set of rules for determining how data is sorted and compared. A collation is usually applied to how text data is sorted in different languages for indexing for making comparisons between text values. Different languages have different character sets and ordering. With a collation, you can sort character data for a given language by using rules that define the correct character sequence. You can also specify the following:

- Case-sensitivity
- Accent marks
- Kana character types
- Use of symbols or punctuation
- Character width
- Word sorting

There might be a performance impact if the join column uses a different collation. The following example query uses three tables, with a different collation for the join column.

Table name

rn_r_segment

rn_r_segment_pax

Column name

pnr_number character varying(15)
COLLATE pg_catalog."C" NOT NULL

pnr_number character varying(15)
COLLATE pg_catalog."default" NOT
NULL

`rn_r_seat_numbers`

`pnr_number character varying(15)`
`COLLATE pg_catalog."default" NOT`
`NULL`

```
EXPLAIN ANALYZE SELECT
A.PNR_NUMBER,
A.PAX_ID,
A.SEGMENT_ID,
B.OANDD_ID,
C.SEAT_ID,
C.BD_AIRPORT_CODE,
C.OFF_AIRPORT_CODE,
C.SEAT_NUMBER ,
B.CABIN_CLASS ,
A.SEGMENT_PAX_ID,
C.SEAT_ALLOC_ID,
C.SSR_ID,
C.SEAT_ATTRIBUTE_CODE
from
RNR_SEGMENT_PAX A,
RNR_SEGMENT B,
RNR_SEAT_NUMBERS C
where
B.AIRLINE_IATA_CODE = 'XX'
and B.FLIGHT_CARRIER = 'XX'
and B.FLIGHT_NUMBER = 140
and B.FLIGHT_SUFFIX = '*'
and B.FLIGHT_DATE_LTC = TO_DATE('01-JAN-2023', 'DD-MON-YYYY')
and A.AIRLINE_IATA_CODE = B.AIRLINE_IATA_CODE
and A.PNR_NUMBER = B.PNR_NUMBER
and A.SEGMENT_ID = B.SEGMENT_ID
and C.AIRLINE_IATA_CODE = B.AIRLINE_IATA_CODE
and C.PNR_NUMBER = B.PNR_NUMBER
and C.SEGMENT_ID = B.SEGMENT_ID
and A.PAX_ID = C.PAX_ID
and B.PNR_NUMBER in ('9F1588', 'E37DE0', '04E82B', '813D11', 'BFF10F');
```

The query plan for the previous query uses a sequence scan on the `rn_r_seat_numbers` table even though that table has a proper index on the joined columns. The planner isn't using an index scan because these joined columns are using different collations:

```

Nested Loop (cost=1112.14..927363.51 rows=1 width=833) (actual time=5395.367..5397.253
rows=0 loops=1)
  Join Filter: (((b.pnr_number)::text = (a.pnr_number)::text) AND (b.segment_id =
a.segment_id))
    -> Gather (cost=1111.58..670766.48 rows=1 width=843) (actual
time=5395.367..5397.251 rows=0 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Hash Join (cost=111.58..669766.38 rows=1 width=843) (actual
time=5388.992..5388.993 rows=0 loops=3)
        Hash Cond: (((c.pnr_number)::text = (b.pnr_number)::text) AND
(c.segment_id = b.segment_id))
          -> Parallel Seq Scan on rnr_seat_numbers c (cost=0.00..582154.96
rows=16666637 width=760) (actual time=0.008..2963.019 rows=13333333 loops=3)
            Filter: ((airline_iata_code)::text = 'XX'::text)
          -> Hash (cost=111.52..111.52 rows=4 width=86) (actual time=0.121..0.121
rows=2 loops=3)
            Buckets: 1024 Batches: 1 Memory Usage: 9kB
            -> Index Scan using rnr_segment_pk on rnr_segment b
(cost=0.56..111.52 rows=4 width=86) (actual time=0.082..0.116 rows=2 loops=3)
              Index Cond: (((pnr_number)::text = ANY
('{9F1588,E37DE0,04E82B,813D11,BFF10F}'::text[])) AND ((airline_iata_code)::text =
'XX'::text))
              Filter: (((flight_carrier)::text = 'XX'::text) AND
(flight_number = 140) AND ((flight_suffix)::text = '*'::text) AND (flight_date_ltc =
to_date('01-JAN-2023'::text, 'DD-MON-YYYY'::text)))
              Rows Removed by Filter: 20
            -> Index Scan using rnr_segment_pax_pk on rnr_segment_pax a (cost=0.56..256597.02
rows=1 width=28) (never executed)
              Index Cond: (((airline_iata_code)::text = 'XX'::text) AND (segment_id =
c.segment_id) AND (pax_id = c.pax_id))
              Filter: ((c.pnr_number)::text = (pnr_number)::text)
Planning Time: 0.982 ms
Execution Time: 5397.314 ms

```

To change the table column collation from the "C" language to the default collation provided by PostgreSQL, run the following alter statement, and then analyze the table:

```

alter table rnr_segment alter column pnr_number type character varying(15) COLLATE
pg_catalog."default";

Analyze rnr_segment;

```

The query plan now uses an index scan, and the runtime is reduced.

```
Nested Loop (cost=1.69..146.63 rows=1 width=833) (actual time=0.155..0.155 rows=0
loops=1)
-> Nested Loop (cost=1.13..145.89 rows=1 width=111) (actual time=0.154..0.155
rows=0 loops=1)
    -> Index Scan using rnr_segment_pk on rnr_segment b (cost=0.56..111.51 rows=4
width=86) (actual time=0.048..0.097 rows=2 loops=1)
        Index Cond: (((pnr_number)::text = ANY
('{9F1588,E37DE0,04E82B,813D11,BFF10F}'::text[])) AND ((airline_iata_code)::text =
'XX'::text))
        Filter: (((flight_carrier)::text = 'XX'::text) AND (flight_number =
140) AND ((flight_suffix)::text = '*'::text) AND (flight_date_ltc = to_date('01-
JAN-2023'::text, 'DD-MON-YYYY'::text)))
        Rows Removed by Filter: 20
    -> Index Scan using rnr_segment_pax_pk on rnr_segment_pax a (cost=0.56..8.58
rows=1 width=28) (actual time=0.027..0.027 rows=0 loops=2)
        Index Cond: (((airline_iata_code)::text = 'XX'::text) AND
((pnr_number)::text = (b.pnr_number)::text) AND (segment_id = b.segment_id))
    -> Index Scan using rnr_seat_numbers_pk on rnr_seat_numbers c (cost=0.56..0.72
rows=1 width=760) (never executed)
        Index Cond: (((pnr_number)::text = (a.pnr_number)::text) AND (segment_id =
a.segment_id) AND (pax_id = a.pax_id) AND ((airline_iata_code)::text = 'XX'::text))
Planning Time: 1.432 ms
Execution Time: 0.207 ms
```

Use case 2 – Data type mismatch

Choosing the proper data type based on the data helps to provide the optimum balance between storage size and performance.

The following example query uses the `pnr_number` column to join two tables. The `pnr_number` column has different data types in different tables.

Table name	Column name and data type
<code>perf_user.rnr_segment_pax</code>	<code>pnr_number character varying(6)</code>
<code>perf_user.rnr_expiry_date</code>	<code>pnr_number character(2)</code>

```
EXPLAIN ANALYZE UPDATE perf_user.RNR_SEGMENT_PAX x SET ARC_EXPIRY_DATE =
y.ARC_EXPIRY_DATE
FROM (SELECT AIRLINE_IATA_CODE, PNR_NUMBER, ARC_EXPIRY_DATE, 0+row_num ROW_NUM
FROM perf_user.RNR_EXPIRY_DATE
WHERE airline_iata_code = 'XX'
AND row_num BETWEEN (1*5000)+0 AND (1+1)*5000) y
WHERE x.airline_iata_code = y.airline_iata_code
AND x.PNR_NUMBER =y.PNR_NUMBER;
```

```
-----

Update on rnr_segment_pax x (cost=290.97..1104986.32 rows=15515 width=460) (actual
time=14574.118..14574.120 rows=0 loops=1)
-> Hash Join (cost=290.97..1104986.32 rows=15515 width=460) (actual
time=16.967..14101.983 rows=11953 loops=1)
Hash Cond: ((x.pnr_number)::text = (rnr_expiry_date.pnr_number)::text)
-> Seq Scan on rnr_segment_pax x (cost=0.00..954539.00 rows=40000320
width=446) (actual time=0.011..9702.989 rows=40000000 loops=1)
Filter: ((airline_iata_code)::bpchar = 'XX'::bpchar)
-> Hash (cost=225.37..225.37 rows=5248 width=24) (actual time=16.540..16.541
rows=5001 loops=1)
Buckets: 8192 Batches: 1 Memory Usage: 338kB
-> Index Scan using rnr_expiry_date_idx1 on rnr_expiry_date
(cost=0.29..225.37 rows=5248 width=24) (actual time=3.102..15.331 rows=5001 loops=1)
Index Cond: ((row_num >= 5000) AND (row_num <= 10000))
Filter: (airline_iata_code = 'XX'::bpchar)

Planning Time: 4.445 ms
Execution Time: 14574.322 ms
```

When you run `EXPLAIN ANALYZE`, the planner uses a sequence scan on `rnr_segment_pax` instead of an index scan even though the columns used in the join have indexes. The planner isn't using an index scan because the columns used in the join are different lengths.

Alter the table columns to keep the data type the same for both tables involved in the join condition, and then analyze the table:

```
alter table perf_user.rnr_expiry_date alter column airline_iata_code type character
varying(6) ;

analyze perf_user.rnr_expiry_date;
```

Now the tables have the same length on both of the columns that are used in the join condition.

Run EXPLAIN ANALYZE again. The planner performs an index scan, which improves the query performance significantly.

```
Update on rnr_segment_pax x (cost=0.86..59733.09 rows=14637 width=460) (actual
time=416.653..416.654 rows=0 loops=1)
-> Nested Loop (cost=0.86..59733.09 rows=14637 width=460) (actual
time=0.103..91.106 rows=11953 loops=1)
-> Index Scan using rnr_expiry_date_idx1 on rnr_expiry_date
(cost=0.29..212.69 rows=4951 width=24) (actual time=0.025..3.023 rows=5001 loops=1)
Index Cond: ((row_num >= 5000) AND (row_num <= 10000))
Filter: ((airline_iata_code)::text = 'XX'::text)
-> Index Scan using rnr_segment_pax_pk on rnr_segment_pax x (cost=0.56..11.99
rows=3 width=446) (actual time=0.014..0.016 rows=2 loops=5001)
Index Cond: (((airline_iata_code)::text = 'XX'::text) AND
((pnr_number)::text = (rnr_expiry_date.pnr_number)::text))
Planning Time: 0.310 ms
Execution Time: 416.696 ms
```

Use case 3 – Function call in the SELECT statement

Calling a function in a where clause can reduce query performance when the function is VOLATILE and you don't use the select keyword while calling the function:

```
Select * from tab_name where FieldName = FunctionName(parameters);
```

An index scan runs if the select statement is used while calling the function:

```
Select * from tab_name where FieldName = ( select FunctionName(parameters) );
```

The pnr_number field has an index in the rnr_expiry_date table. The index is used when comparing the value in the where clause.

```
explain analyze select * from perf_user.rnr_expiry_date where pnr_number= 'EE9F41';

"Index Scan using rnr_expiry_date_idx3 on rnr_expiry_date (cost=0.29..8.31 rows=1
width=72) (actual time=0.020..0.021 rows=1 loops=1)"
" Index Cond: ((pnr_number)::text = 'EE9F41'::text)"
"Planning Time: 0.063 ms"
"Execution Time: 0.038 ms"
```

A sequential scan is performed when a function is called without the `select` keyword even when an index is available on the field.

```
explain analyze select * from perf_user.rnr_expiry_date where pnr_number=
perf_user.return_data();

"Seq Scan on rnr_expiry_date (cost=0.00..27084.00 rows=1 width=72) (actual
time=0.112..135.917 rows=1 loops=1)"
"  Filter: ((pnr_number)::text = (perf_user.return_data())::text)"
"  Rows Removed by Filter: 99999"
"Planning Time: 0.053 ms"
"Execution Time: 136.803 ms"
```

An index scan is performed when the function is called with the `select` keyword.

```
explain analyze select * from perf_user.rnr_expiry_date where pnr_number= (select
perf_user.return_data() );

"Index Scan using rnr_expiry_date_idx3 on rnr_expiry_date (cost=0.55..8.57 rows=1
width=72) (actual time=0.058..0.061 rows=1 loops=1)"
"  Index Cond: ((pnr_number)::text = ($0)::text)"
"  InitPlan 1 (returns $0)"
"    -> Result (cost=0.00..0.26 rows=1 width=32) (actual time=0.021..0.022 rows=1
loops=1)"
"Planning Time: 0.147 ms"
"Execution Time: 0.111 ms"
```

Use case 4 – IN or EXISTS

If the query has `IN` or `NOT IN` operators, we recommend checking the query plan to confirm that the proper index is being used. If the proper index isn't being used and query performance is taking more time than expected, try to rewrite the query using the `EXISTS` or `NOT EXISTS` conditions.

Consider the following example, which uses `NOT IN`:

```
EXPLAIN ANALYZE SELECT
  TEST_VEH.TEST_VEH_ID,
  TEST_VEH.VEHICLE_ID,
  TEST_VEH.SERVICEPROGRAM_ID,
  TEST_VEH.STARTDATE,
  TEST_VEH.ENDDATE,
```

```

    TEST_VEH.OILTYPE_ID
FROM PERF_USER.TEST_VEH TEST_VEH
JOIN PERF_USER.OILTYPE OT ON OT.OILTYPE_ID =TEST_VEH.OILTYPE_ID
JOIN PERF_USER.SERVICEPROGRAM SP ON SP.SERVICEPROGRAM_ID = TEST_VEH.SERVICEPROGRAM_ID
WHERE SP.PROGNAME = '18FCE8FDAF365BB'
    AND OT.OILTYPE_ID =3
    AND TEST_VEH.ENDDATE IS NOT NULL
    AND TEST_VEH.TEST_VEH_ID NOT IN
        (SELECT TEST_VEH_ID
         FROM PERF_USER.VEHICLESERVICEHISTORY
         WHERE TEST_VEH_ID > 1
        );
-----
"Nested Loop (cost=1009.16..1188860356305.01 rows=1 width=76) (actual
time=37299.891..37347.853 rows=0 loops=1)"
"  -> Gather (cost=1009.16..1188860356303.88 rows=1 width=76) (actual
time=37299.890..37347.849 rows=0 loops=1)"
"      Workers Planned: 2"
"      Workers Launched: 2"
"      -> Hash Join (cost=9.16..1188860355303.78 rows=1 width=76) (actual
time=37286.742..37286.751 rows=0 loops=3)"
"          Hash Cond: ((test_veh.serviceprogram_id)::text =
(sp.serviceprogram_id)::text)"
"          -> Parallel Index Scan using test_veh_oiltype_id_ind on test_veh
(cost=0.56..1188860351273.04 rows=1072570 width=76) (actual time=37276.290..37276.292
rows=1 loops=3)"
"              Index Cond: (oiltype_id = 3)"
"              Filter: ((enddate IS NOT NULL) AND (NOT (SubPlan 1)))"
"              Rows Removed by Filter: 0"
"              SubPlan 1"
"                  -> Materialize (cost=0.00..1025071.31 rows=33333332 width=8)
(actual time=0.418..23201.432 rows=25001498 loops=4)"
"                      -> Seq Scan on vehicleservicehistory
(cost=0.00..728195.65 rows=33333332 width=8) (actual time=0.416..13249.975
rows=25001498 loops=4)"
"                          Filter: (test_veh_id > 1)"
"                      -> Hash (cost=8.58..8.58 rows=1 width=11) (actual time=9.045..9.046
rows=0 loops=3)"
"                          Buckets: 1024 Batches: 1 Memory Usage: 8kB"
"                          -> Index Scan using progname_id_ind on serviceprogram sp
(cost=0.56..8.58 rows=1 width=11) (actual time=9.043..9.044 rows=0 loops=3)"
"                              Index Cond: ((progname)::text = '18FCE8FDAF365BB'::text)"
"                      -> Seq Scan on oiltype ot (cost=0.00..1.12 rows=1 width=8) (never executed)"
"                          Filter: (oiltype_id = 3)"

```

```
"Planning Time: 37.696 ms"
"Execution Time: 37366.335 ms"
```

The query is taking more than 37 seconds 366 milliseconds to retrieve 4 million records.

The query plan states that a sequence scan is performed on the table used in the subquery `vehiclesservicehistory`. The sequence scan is producing a large number of records. For each of those records in the subquery, the query is performing a full table scan, which is causing the performance issue.

To avoid the sequence scan on the subquery, rewrite the subquery to use a correlated subquery with `NOT EXISTS`. The correlated subquery will use an index scan and a reduced number of table scans:

```
EXPLAIN ANALYZE SELECT
    TEST_VEH.TEST_VEH_ID,
    TEST_VEH.VEHICLE_ID,
    TEST_VEH.SERVICEPROGRAM_ID,
    TEST_VEH.STARTDATE,
    TEST_VEH.ENDDATE,
    TEST_VEH.OILTYPE_ID
FROM PERF_USER.TEST_VEH TEST_VEH
JOIN PERF_USER.OILTYPE OT ON OT.OILTYPE_ID =TEST_VEH.OILTYPE_ID
JOIN PERF_USER.SERVICEPROGRAM SP ON SP.SERVICEPROGRAM_ID = TEST_VEH.SERVICEPROGRAM_ID
WHERE SP.PROGNAME = '18FCE8FDAF365BB'
      AND OT.OILTYPE_ID =3
      AND TEST_VEH.ENDDATE IS NOT NULL
      AND NOT EXISTS
          (SELECT TEST_VEH_ID
           FROM PERF_USER.VEHICLESERVICEHISTORY
           WHERE
TEST_VEH.TEST_VEH_ID=VEHICLESERVICEHISTORY.TEST_VEH_ID
              AND TEST_VEH_ID > 1
          );

-----
"Nested Loop Anti Join (cost=1009.03..936146.10 rows=1 width=76) (actual
time=12.693..12.810 rows=0 loops=1)"
"  -> Nested Loop (cost=1008.59..936141.78 rows=1 width=76) (actual
time=12.692..12.809 rows=0 loops=1)"
"      -> Gather (cost=1008.59..936140.64 rows=1 width=76) (actual
time=12.691..12.807 rows=0 loops=1)"
"          Workers Planned: 2"
```

```

"           Workers Launched: 2"
"           -> Hash Join (cost=8.59..935140.54 rows=1 width=76) (actual
time=0.773..0.774 rows=0 loops=3)"
"           Hash Cond: ((test_veh.serviceprogram_id)::text =
(sp.serviceprogram_id)::text)"
"           -> Parallel Seq Scan on test_veh (cost=0.00..927087.67
rows=2145139 width=76) (actual time=0.672..0.672 rows=1 loops=3)"
"           Filter: ((enddate IS NOT NULL) AND (oiltype_id = 3))"
"           Rows Removed by Filter: 7"
"           -> Hash (cost=8.58..8.58 rows=1 width=11) (actual
time=0.040..0.040 rows=0 loops=3)"
"           Buckets: 1024 Batches: 1 Memory Usage: 8kB"
"           -> Index Scan using progname_id_ind on serviceprogram sp
(cost=0.56..8.58 rows=1 width=11) (actual time=0.039..0.040 rows=0 loops=3)"
"           Index Cond: ((progname)::text =
'18FCE8FDAF365BB'::text)"
"           -> Seq Scan on oiltype ot (cost=0.00..1.12 rows=1 width=8) (never executed)"
"           Filter: (oiltype_id = 3)"
" -> Index Only Scan using veh_ser_ind on vehicleservicehistory (cost=0.44..4.32
rows=1 width=8) (never executed)"
"           Index Cond: ((test_veh_id = test_veh.test_veh_id) AND (test_veh_id > 1))"
"           Heap Fetches: 0"
"Planning Time: 11.115 ms"
"Execution Time: 12.871 ms"

```

After the modification, the query is taking less than 13 ms to process 4 million records

According to the query plan of the modified query, the table `vehicleservicehistory` can have an index scan. Using an index scan reduces the cost and the number of affected rows. This way, you can reduce the runtime of a query and increase its performance.

Use case 5 – Subqueries or CTEs

Common Table Expressions (CTEs) help break down large queries into smaller queries. This makes the whole query easier to maintain.

Subquery joins are replaced by CTE joins, which are more readable because the query is named and separated inside the CTE section. This is especially helpful when the size of the query grows and the query becomes harder to maintain. In addition, the CTE results in PostgreSQL are materialized. If you call the CTE in multiple places, the actual query definition will be run only one time. The result will be stored in memory. You can use this for any complex logic that must be used in

multiple places in the same query. Put that logic inside a CTE, and call the CTE any number of times.

For example, a customer was using inline application queries with many subqueries within queries. The subqueries were filtered by input parameter values sent from the applications.

```
EXPLAIN ANALYZE
SELECT * FROM
ORDER_DETAILS A
WHERE A.ORDID IN (SELECT ORDID FROM PAYMENT_DETAILS)
AND A.ORDID IN (SELECT ORDID FROM ITEM_DETAILS )
AND A.ORDID = 1000000;
```

```
"Nested Loop Semi Join (cost=3000.00..194258.21 rows=5 width=74) (actual
time=201.605..747.945 rows=5 loops=1)"
"  -> Nested Loop Semi Join (cost=2000.00..135040.47 rows=5 width=74) (actual
time=146.016..666.779 rows=5 loops=1)"
"      -> Gather (cost=1000.00..78580.31 rows=5 width=74) (actual
time=58.893..463.570 rows=5 loops=1)"
"          Workers Planned: 2"
"          Workers Launched: 2"
"      -> Parallel Seq Scan on order_details a (cost=0.00..77579.81 rows=2
width=74) (actual time=165.627..549.702 rows=2 loops=3)"
"          Filter: (ordid = 1000000)"
"          Rows Removed by Filter: 1666665"
"      -> Materialize (cost=1000.00..56460.07 rows=3 width=4) (actual
time=17.424..40.638 rows=1 loops=5)"
"          -> Gather (cost=1000.00..56460.06 rows=3 width=4) (actual
time=87.113..203.178 rows=1 loops=1)"
"              Workers Planned: 2"
"              Workers Launched: 2"
"          -> Parallel Seq Scan on payment_details (cost=0.00..55459.76
rows=1 width=4) (actual time=174.431..423.792 rows=1 loops=3)"
"              Filter: (ordid = 1000000)"
"              Rows Removed by Filter: 1333002"
"      -> Materialize (cost=1000.00..59217.64 rows=4 width=4) (actual time=11.117..16.231
rows=1 loops=5)"
"          -> Gather (cost=1000.00..59217.62 rows=4 width=4) (actual
time=55.581..81.148 rows=1 loops=1)"
"              Workers Planned: 2"
"              Workers Launched: 2"
"          -> Parallel Seq Scan on item_details (cost=0.00..58217.22 rows=2
width=4) (actual time=287.030..411.004 rows=1 loops=3)"
```

```
"          Filter: (ordid = 1000000)"
"          Rows Removed by Filter: 1333080"
"Planning Time: 0.266 ms"
"Execution Time: 747.986 ms"
```

After modifying the subqueries by using a CTE and adding filters so that only required row sets are retrieved, the query performance improves.

```
EXPLAIN ANALYZE
WITH PAYMENT AS
(
  SELECT * FROM PAYMENT_DETAILS WHERE  ORDID = 1000000
),
ITEM AS
(SELECT * FROM ITEM_DETAILS  WHERE  ORDID = 1000000)
SELECT * FROM
ORDER_DETAILS A JOIN PAYMENT B
ON A.ORDID=B.ORDID
JOIN ITEM C ON B.ORDID=C.ORDID
```

```
"Nested Loop (cost=3000.00..194258.91 rows=60 width=166) (actual time=586.410..732.918
rows=80 loops=1)"
"  -> Nested Loop (cost=2000.00..115677.83 rows=12 width=92) (actual
time=456.760..457.083 rows=16 loops=1)"
"    -> Gather (cost=1000.00..59217.62 rows=4 width=48) (actual
time=153.802..154.060 rows=4 loops=1)"
"      Workers Planned: 2"
"      Workers Launched: 2"
"    -> Parallel Seq Scan on item_details (cost=0.00..58217.22 rows=2
width=48) (actual time=85.417..249.045 rows=1 loops=3)"
"      Filter: (ordid = 1000000)"
"      Rows Removed by Filter: 1333332"
"    -> Materialize (cost=1000.00..56460.07 rows=3 width=44) (actual
time=75.738..75.753 rows=4 loops=4)"
"      -> Gather (cost=1000.00..56460.06 rows=3 width=44) (actual
time=302.947..303.005 rows=4 loops=1)"
"        Workers Planned: 2"
"        Workers Launched: 2"
"      -> Parallel Seq Scan on payment_details (cost=0.00..55459.76
rows=1 width=44) (actual time=184.609..294.784 rows=1 loops=3)"
"        Filter: (ordid = 1000000)"
"        Rows Removed by Filter: 1333332"
```

```
" -> Materialize (cost=1000.00..78580.34 rows=5 width=74) (actual time=8.103..17.238
rows=5 loops=16)"
"      -> Gather (cost=1000.00..78580.31 rows=5 width=74) (actual
time=129.641..275.795 rows=5 loops=1)"
"          Workers Planned: 2"
"          Workers Launched: 2"
"      -> Parallel Seq Scan on order_details a (cost=0.00..77579.81 rows=2
width=74) (actual time=78.556..268.994 rows=2 loops=3)"
"          Filter: (ordid = 1000000)"
"          Rows Removed by Filter: 1666665"
"Planning Time: 0.108 ms"
"Execution Time: 732.953 ms"
```

These are the observations from the example data. When you run the query on a huge dataset, the difference in performance will be very high.

FAQ

Find answers to frequently raised questions about tuning query performance.

What is EXPLAIN?

EXPLAIN is a keyword that you prepend to a PostgreSQL query (SELECT, UPDATE, INSERT, DELETE) to generate a query plan. The PostgreSQL query plan details how the database intends to run the query. This plan includes information about the order of a table scan, index usage, and joins.

Use the query plan to identify potential bottlenecks, optimize queries, and improve overall performance. When reviewing the query plan, consider the following factors:

- Table access approaches
- Join approaches
- Filter conditions
- Sort operations
- Index usage
- Parallelism
- Statistics
- Cost estimations
- Rows retrieved from each step
- Data distribution

For more information about [EXPLAIN](#), see the PostgreSQL documentation.

What is EXPLAIN ANALYZE?

When you prepend EXPLAIN ANALYZE to a query and run the query, PostgreSQL runs the query and returns both the query plan and runtime statistics. The actual runtime, rows processed from each step, and other relevant information are displayed along with the query plan. Using EXPLAIN ANALYZE on a production database should be done with caution, because running the query could impact database performance during the analysis.

For more information about [EXPLAIN ANALYZE](#), see the PostgreSQL documentation.

What is collation in PostgreSQL?

In PostgreSQL, a collation is a set of rules for determining how strings are compared and sorted. The collation defines the order in which characters are considered in comparisons, considering language-specific rules and conversions.

For more information about [collation](#), see the PostgreSQL documentation.

What is a CTE?

In a PostgreSQL database, a Common Table Expression (CTE) is a named temporary result set that you can reference. CTEs provide a way to create more readable and modular SQL queries by breaking down complex logic into smaller, named units.

For more information about [CTEs](#), see the PostgreSQL documentation.

What are the categories of functions in PostgreSQL?

Every PostgreSQL function has a volatility classification, with the possibilities being VOLATILE, STABLE, or IMMUTABLE:

- **VOLATILE** – A VOLATILE function can do anything, including modify the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
- **STABLE** – A STABLE function can't modify the database. It's guaranteed to return the same results given the same arguments for all rows within a single statement. When you use this classification, the optimizer can optimize multiple calls of the function to a single call. In particular, it's safe to use an expression that contains such a function in an index scan condition. (Because an index scan will evaluate the comparison value only one time, not one time at each row, it isn't valid to use a VOLATILE function in an index scan condition.)
- **IMMUTABLE** – An IMMUTABLE function can't modify the database and is guaranteed to return the same results given the same arguments forever. When you use this classification, the optimizer can pre-evaluate the function when a query calls it with constant arguments. For example, a query such as `SELECT ... WHERE x = 2 + 2` can be simplified on sight to

`SELECT ... WHERE x = 4`, because the function underlying the integer addition operator is marked IMMUTABLE.

VOLATILE is the default if the `CREATE FUNCTION` command doesn't specify a category. For more information about [function types](#), see the PostgreSQL documentation.

Resources

References

- [EXPLAIN](#)
- [Using EXPLAIN](#)
- [Collation Support](#)
- [WITH Queries \(Common Table Expressions\)](#)

Guides

- [Maintenance activities for PostgreSQL databases in Amazon RDS and Amazon Aurora to avoid performance issues](#)
- [Tuning PostgreSQL parameters in Amazon RDS and Amazon Aurora](#)

Contributors

Contributors to this document include:

- Tirumala Dasari, Lead Consultant – Databases, AWS
- Veeranjanyulu Grandhi, Lead Consultant – Databases, AWS
- Vamsikrishna Jammula, Consultant – Databases, AWS
- Srinivas Potlachervoo, Senior Lead Consultant – Databases, AWS
- Naga Srinivas Reddy Ravulapati, Consultant – Databases, AWS

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an [RSS feed](#).

Change	Description	Date
Initial publication	—	April 23, 2024