

Query Optimization Approaches to Configuration



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

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.

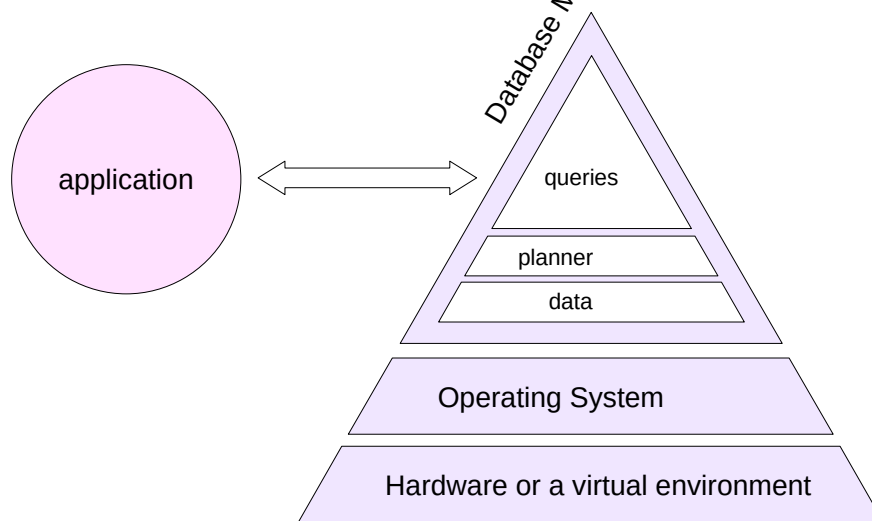
What Should Be Configured?

Server Configuration

Application Configuration

Queries

What Should Be Configured?



3

The performance of an information system is ensured at different levels.

We do not consider two important levels: hardware (which can be virtualized, thereby introducing an additional layer to the architecture) and the operating system. Specifically, the database management system (DBMS) interacts with the file system, and the efficiency and reliability of I/O operations rely on the disk subsystem. First and foremost, we need to ensure the necessary resources and configure the OS to maximize their utilization.

Database server is configured to ensure that administrative tasks don't consume more resources than necessary and that most queries run efficiently. At this level, we can not only configure configuration parameters (a small portion of which will be discussed) but also manage data placement. Our goal is to ensure effective resource allocation and proper planner operation.

The next step is configuring individual queries. In the "Profiling" section, we discussed how developers typically focus on optimizing queries they are currently writing, while the administrator focuses on optimizing queries that have the greatest impact on the server as a whole. In this section, we'll explore some query optimization techniques.

However, the queries that make up the workload are initiated by the application. Therefore, application configuration is just as important as database management system configuration.

Resource usage

Physical Data Placement

Optimizer Statistics and Configuration

Resources: CPU and Input/Output

Background Worker Processes

```
max_parallel_workers_per_gather = 2  
max_parallel_workers = 8  
max_worker_processes = 8
```

Input/Output

```
effective_io_concurrency = 1  
maintenance_io_concurrency = 10
```

We'll begin with the settings that determine which resources the DBMS can utilize.

PostgreSQL can utilize multiple CPU cores for parallel processing. This involves launching additional background worker processes. The detailed discussion of parallel query execution can be found in the "Parallel Access" section, while the use of background processes for application development is covered in the "Background Processes" section of the DEV2 course.

Among the input-output related settings, let's highlight the parameter `effective_io_concurrency`. It tells the system how many individual disks are in the disk array. Actually, this parameter only affects the number of pages pre-fetched into the cache during a bitmap scan.

A similar parameter used for some maintenance operations is referred to as `maintenance_io_concurrency`.

Memory Resources

Memory

work_mem
filter: (departure_airport = \$2)
maintenance_work_mem
shared_buffers
effective_io_concurrency = 4



Several settings affect memory allocation and usage.

The `work_mem` parameter determines the amount of memory available for specific operations (discussed in related sections).

It also influences the selection of the plan. For example, with smaller values, sorting is preferred as it performs better with limited memory compared to hashing. Therefore, for nodes using hashing, the working memory size can be increased by adjusting the `hash_mem_multiplier` parameter.

The `maintenance_work_mem` parameter impacts the index building speed and the operation of maintenance processes.

The `shared_buffers` parameter determines the size of the buffer cache for the instance. The configuration of this parameter is discussed in the DBA2 course, but it's clear that the default value is very small.

The `effective_cache_size` parameter tells PostgreSQL the total amount of cacheable data, including both the buffer cache and the OS cache. The higher its value, the more index access is preferred. This parameter does not impact the actual memory allocation.

Tablespaces

Distributing data across multiple physical devices

`ALTER TABLESPACE SET ...`

Partitioning

Partitioning a table into independently manageable parts to simplify administration and improve access speed

Sharding

Distributing partitions across multiple servers to scale read and write workload

Partitioning and extending PostgreSQL FDW or third-party solutions

The physical organization of data can greatly impact performance.

Tablespaces allow you to control the placement of objects on physical I/O devices. For example, store frequently accessed data on SSDs and archival data on slower HDDs.

Some server parameters that depend on storage device characteristics (such as `random_page_cost`) can be set at the tablespace level.

Partitioning enables efficient handling of very large data volumes. The main performance benefit lies in replacing a full table scan with a scan of an individual partition. Note that partitions can also be stored in different tablespaces.

Another approach is to place partitions on different servers (sharding), enabling distributed queries. Standard PostgreSQL includes only the core features required for sharding: partitioning and the `postgres-fdw` extension. Sharding can be more effectively and fully implemented using external solutions. This topic is covered in the final section of the DBA3 course.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

<https://postgrespro.com/docs/postgresql/16/sql-copy>

Relevance

Autovacuum and Autoanalyze Settings

`autovacuum_max_workers`, `autovacuum_analyze_scale_factor`, ...

Accuracy

`default_statistics_target` = 100

Expression Index, Extended Statistics

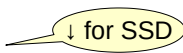
The planner relies on statistics for its estimates, so frequent collection is essential. This is accomplished by adjusting the autovacuum and autoanalyze settings. The settings are discussed in detail in the DBA2 course.

Additionally, the statistics should be accurate enough. Absolute accuracy cannot be achieved (and isn't necessary), but errors should not result in incorrect execution plans. A sign of outdated or inaccurate statistics is a significant discrepancy between the expected and actual row counts in the leaf-level nodes of the execution plan.

To enhance accuracy, you might need to adjust the `default_statistics_target` setting, either globally or for individual table columns. An expression index with its own statistics can sometimes be useful. In certain situations, extended statistics can be useful.

Input/Output

`seq_page_cost`
`random_page_cost`



↓ for SSD

CPU Time

`cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`
User function cost

There are numerous cost parameters for basic operations, from which, as we've already seen, the query plan's cost is ultimately determined. It's advisable to adjust these settings if queries, despite accurate cardinality estimation, are not performing efficiently.

The `seq_page_cost` and `random_page_cost` parameters relate to I/O and determine the relative cost of reading a page during sequential or random access.

The `seq_page_cost` parameter is set to one and shouldn't be modified. A high value for the `random_page_cost` parameter reflects the realities of HDD disk operations. For SSD drives (as well as when all data is highly likely to be cached), this parameter should be significantly reduced, for example, to 1.1.

CPU time parameters determine the weights used to account for the cost of processing retrieved data. Usually, these parameters aren't changed because it's hard to predict the impact of changes.

But, as discussed in the "Functions" section, there are cases where defining the cost of a user-defined function in terms of `cpu_operator_cost` can be beneficial.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

Cost Settings

Example of a query with accurate cardinality estimation:

```
=> EXPLAIN (analyze, timing off) SELECT *  
FROM bookings  
WHERE book_ref < '9000';
```

QUERY PLAN

```
-----  
-----  
Seq Scan on bookings (cost=0.00..39835.88 rows=1202926 width=21) (actual rows=1187454  
loops=1)  
  Filter: (book_ref < '9000'::bpchar)  
    Rows Removed by Filter: 923656  
  Planning Time: 4.927 ms  
  Execution Time: 551.540 ms  
(5 rows)
```

The planner selected a sequential scan, based on full information.

Is this a good choice? Let's test it by instructing the planner not to use sequential scanning if other methods are available:

```
=> SET enable_seqscan = off; -- active until the end of the session
```

SET

Analogous enable parameters are available for other connection methods, access methods, and various other operations. These parameters significantly interfere with the planner's functionality but are highly useful for debugging and experimentation.

```
=> EXPLAIN (analyze, timing off) SELECT *  
FROM bookings  
WHERE book_ref < '9000';
```

QUERY PLAN

```
-----  
-----  
Index Scan using bookings_pkey on bookings (cost=0.43..41921.64 rows=1202926 width=21)  
(actual rows=1187454 loops=1)  
  Index Cond: (book_ref < '9000'::bpchar)  
  Planning Time: 0.074 ms  
  Execution Time: 404.562 ms  
(4 rows)
```

The result is likely to favor an index scan since all data is cached, allowing for fast random access. This scenario is also possible with fast SSDs. Drawing conclusions based on a single query is incorrect, but a systematic error should prompt adjustments to global settings. In such a case, it's advisable to either lower the `random_page_cost` value to prevent the planner from overestimating index access costs or increase `effective_cache_size` to make index scanning more appealing.

```
=> RESET enable_seqscan; -- disable
```

RESET

Clients and Connections

Data schema

Multiple Sessions

- connection pool

Cursors

- if a portion of the sample is required

- cursor_tuple_fraction*

Multiple short queries

- prepared operators

- Moving logic to SQL

If an application opens too many connections or establishes them too frequently for short durations, it may be necessary to implement connection pooling between the application and the DBMS. However, the pool places limitations on the application. This question is covered in detail in the DEV2 course.

Cursors should be used only when retrieving a small portion of a result set, and the size depends on user actions. In this case, the *cursor_tuple_fraction* parameter will guide the planner to optimize fetching the initial rows of the result set.

If the application executes too many small queries (each of which is efficient individually), the overall performance will be poor. This is commonly observed when using object-relational mapping (ORM) tools. In the "Profiling" topic, we discussed a similar example involving a function that was executed in a loop.

In such cases, the DBMS has virtually no optimization tools (except for prepared statements when queries are repeated). A proven approach is to eliminate procedural code in the application and move it to the database server as a small set of large SQL commands. This enables the query planner to utilize more efficient access and join methods, while reducing the frequent data transfers between the client and server. A proven approach is to eliminate procedural code in the application and move it to the database server as a small set of large SQL commands. This enables the query planner to utilize more efficient access and join methods, while reducing the frequent data transfers between the client and server.

Normalization - removal of data redundancy

Simplifies queries and consistency checks

Denormalization involves introducing redundancy

may enhance performance, but requires synchronization

indexes

precomputed fields (such as generated columns or triggers)

materialized views

Caching application results

At the logical level, a database should be normalized—informally, this means eliminating redundancy in the data. If that's not the case, we're dealing with a design flaw: data consistency checks will be challenging, and various anomalies can occur during data changes, and so on.

While data duplication at the storage level can significantly improve performance, this comes at the cost of maintaining synchronization between redundant data and primary data.

The most common method of denormalization is indexes—though they're typically not considered in this context. Indexes are automatically updated.

You can duplicate some data (or calculated results based on this data) in table columns. You can use generated columns or keep the data in sync with triggers. Regardless of the approach, the database is responsible for denormalization.

Another example is materialized views. They must also be updated, for example, on a schedule or through other methods. This topic was thoroughly covered in the "Materialization" section.

Data can also be duplicated at the application level by caching query results. This is a common approach, but it's often used due to the application's improper interaction with the database (e.g., when using ORM). The application is responsible for timely cache updates, ensuring access controls for cache data, and other related tasks.

Data Types

Selecting the right data types

Using composite types, such as arrays and JSON, instead of separate tables.

Data Integrity Constraints

Beyond ensuring data integrity, the planner may take into account eliminating unnecessary joins, enhancing selectivity estimates, and performing other optimizations.

A primary key enforces uniqueness through a unique index.

foreign key

Absence of null values

CHECK constraint (constraint_exclusion)

Choosing the right data types from the wide range of options PostgreSQL offers is crucial. For instance, representing date intervals using range types (daterange, tstzrange) instead of two separate columns enables the use of GiST and SP-GiST indexes for operations like interval intersections.

In certain situations, employing composite types like arrays or JSON instead of the traditional method of creating a separate table can yield benefits. This reduces the need for joins and avoids storing extensive metadata in row version headers. However, such a solution should be approached with caution, as it has its own drawbacks.

Integrity constraints are important in their own right, but the planner can sometimes leverage them for optimization.

- Primary key and unique constraints are enforced via unique indexes. This enables more accurate statistics and more efficient join operations.
- The presence of a foreign key and NOT NULL constraints allows for eliminating unnecessary joins (particularly important when using views) and also improves selectivity estimation when joining on multiple columns.
- A CHECK constraint using the constraint_exclusion parameter can avoid scanning tables (or partitions) when they are guaranteed to contain no relevant data.

Data Schema

Referential integrity constraints are particularly important when dealing with composite keys. Example of accurate row estimation resulting from a join:

```
=> EXPLAIN SELECT *
FROM ticket_flights tf
JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
AND tf.ticket_no = bp.ticket_no;

QUERY PLAN
-----
Hash Join (cost=310609.60..677435.99 rows=7925944 width=57)
  Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf.ticket_no = bp.ticket_no))
    -> Seq Scan on ticket_flights tf (cost=0.00..153852.60 rows=8391960 width=32)
    -> Hash (cost=137538.44..137538.44 rows=7925944 width=25)
        -> Seq Scan on boarding_passes bp (cost=0.00..137538.44 rows=7925944 width=25)
(5 rows)
```

However, removing the foreign key results in an inaccurate estimate. This is another manifestation of the correlated predicates problem:

```
=> BEGIN;

BEGIN

=> ALTER TABLE boarding_passes
DROP CONSTRAINT boarding_passes_ticket_no_fkey;

ALTER TABLE

=> EXPLAIN SELECT *
FROM ticket_flights tf
JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
AND tf.ticket_no = bp.ticket_no;

QUERY PLAN
-----
-----
Gather (cost=164416.92..358090.03 rows=342 width=57)
  Workers Planned: 2
    -> Parallel Hash Join (cost=163416.92..357055.83 rows=142 width=57)
        Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf.ticket_no = bp.ticket_no))
        -> Parallel Seq Scan on ticket_flights tf (cost=0.00..104899.50 rows=3496650
width=32)
            -> Parallel Hash (cost=91303.77..91303.77 rows=3302477 width=25)
                -> Parallel Seq Scan on boarding_passes bp (cost=0.00..91303.77
rows=3302477 width=25)
(7 rows)

=> ROLLBACK;

ROLLBACK
```

Optimization Strategies

Short queries

Long-running queries

The goal of optimization is to achieve an effective plan.

Addressing Inefficiencies

- somehow identify and fix the bottleneck
- It's often hard to pinpoint the issue
- often leads to a conflict with the planner

Accurate Cardinality Calculation

- Ensure accurate cardinality calculation in each node and rely on the planner
- If the plan is still inadequate, tune the global parameters

The goal of query optimization is to achieve an adequate execution plan. There are various ways to achieve this goal.

You can examine the query plan, identify the cause of inefficient execution, and take steps to address the issue. Unfortunately, the problem isn't always obvious, and fixing it often ends up being a battle with the optimizer.

If taking this approach, you'd want the option to fully or partially disable the planner and manually create the execution plan. This capability is referred to as hints and is not explicitly available in PostgreSQL.

Another approach involves ensuring accurate cardinality estimation at each node in the execution plan. While accurate statistics are certainly necessary, they're often not enough.

If we take this approach, we're not battling the planner but helping it make the right decision. Unfortunately, this often proves to be too complex a task.

If the planner still creates an inefficient plan despite accurate cardinality estimates, it's time to adjust the global configuration parameters.

It's usually worthwhile to use both approaches, based on the situation and applying common sense.

Read a small amount of data and return few rows.

Typical of OLTP

It's important to get the answer quickly

Key Features of the Plan

- high-selectivity conditions

- indexes

- nested loop joins

Queries can be somewhat arbitrarily divided into two groups, each with distinct characteristics and requiring different optimization approaches: "short" and "long".

The first group consists of queries typical of OLTP systems. Such queries retrieve limited data and return one or more rows. They can access large tables, but when they do, they employ high-selectivity conditions that allow only a small portion of the data to be retrieved.

Short queries typically handle the user interface, making it crucial that they run as fast as possible. Response time is prioritized.

This is achieved by reading the data needed for the queries either from very small tables (one or two pages) or using an index. Typically, joins are performed using a nested loop, which doesn't need any setup (unlike hash joins) and doesn't run into problems when joining a small number of rows.

Short Queries

Let's immediately disable parallel execution. It's unnecessary for short queries, as parallel plans are harder to read.

```
=> SET max_parallel_workers_per_gather = 0;
```

SET

Let's look at a query that displays boarding passes issued for flights departing within a one-hour window:

```
=> EXPLAIN (analyze, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE date_trunc('hour',f.scheduled_departure) = '2017-06-01 12:00:00';
```

QUERY PLAN

```
-----
Hash Join (cost=5860.43..164204.88 rows=39617 width=25) (actual rows=1799 loops=1)
  Hash Cond: (bp.flight_id = f.flight_id)
    -> Seq Scan on boarding_passes bp (cost=0.00..137538.44 rows=7925944 width=25)
(actual rows=7925812 loops=1)
    -> Hash (cost=5847.00..5847.00 rows=1074 width=4) (actual rows=51 loops=1)
        Buckets: 2048 Batches: 1 Memory Usage: 18kB
        -> Seq Scan on flights f (cost=0.00..5847.00 rows=1074 width=4) (actual
rows=51 loops=1)
            Filter: (date_trunc('hour'::text, scheduled_departure) = '2017-06-01
12:00:00+03'::timestamp with time zone)
            Rows Removed by Filter: 214816
        Planning Time: 0.575 ms
        Execution Time: 2637.333 ms
(10 rows)
```

This query is considered a short one: it requires a much smaller percentage of data from two large tables. However, the execution plan shows that the tables are being fully scanned. Additionally, there's a significant discrepancy between the estimated and actual cardinality.

Begin the optimization with the innermost node, the Seq Scan on the flights table. Why is the planner's cardinality estimate off?

The issue stems from the `date_trunc` function: since there's no planner helper function for it, a fixed selectivity estimate of 0.5% is applied. In the "Basic Statistics" and "Advanced Statistics" sections, we resolved similar problems using expression indexes and extended statistics. However, in this scenario, simply rewriting the condition to place the table's column on the left side of the operator is sufficient:

```
=> EXPLAIN (analyze, timing off) SELECT *
FROM flights
WHERE scheduled_departure >= '2017-06-01 12:00:00'
AND scheduled_departure < '2017-06-01 13:00:00';
```

QUERY

PLAN

```
-----
Index Scan using flights_flight_no_scheduled_departure_key on flights
(cost=0.42..5560.91 rows=22 width=63) (actual rows=51 loops=1)
  Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with time
zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
  Planning Time: 1.024 ms
  Execution Time: 45.063 ms
(4 rows)
```

Now the estimate is more accurate, and the query also started using the index. Is that a good thing?

Generally speaking, it's a good approach, but the index is built on the `flight_no` and `scheduled_departure` columns, with the condition applying only to the second column. As discussed in the "Access Methods" section, the index would be fully scanned in this scenario, which isn't an efficient data access method.

Let's create a new index: We'll create a new index:

```
=> CREATE INDEX ON flights(scheduled_departure);
```

CREATE INDEX

Check our query

```
=> EXPLAIN (analyze, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY PLAN

```
-----
-----
----
Nested Loop (cost=10.30..13319.12 rows=812 width=25) (actual rows=1799 loops=1)
  -> Bitmap Heap Scan on flights f (cost=4.65..86.93 rows=22 width=4) (actual rows=51
loops=1)
    Recheck Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
    Heap Blocks: exact=50
    -> Bitmap Index Scan on flights_scheduled_departure_idx (cost=0.00..4.64
rows=22 width=0) (actual rows=51 loops=1)
      Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp
with time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time
zone))
    -> Bitmap Heap Scan on boarding_passes bp (cost=5.66..599.88 rows=158 width=25)
(actual rows=35 loops=51)
      Recheck Cond: (f.flight_id = flight_id)
      Heap Blocks: exact=49
      -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (cost=0.00..5.62
rows=158 width=0) (actual rows=35 loops=51)
        Index Cond: (flight_id = f.flight_id)
    Planning Time: 0.351 ms
    Execution Time: 41.893 ms
(13 rows)
```

Overall, we've reached our goal: the planner is using index access, and the data is being joined with a nested loop. Let's also check how many pages were read:

```
=> EXPLAIN (analyze, buffers, costs off, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY PLAN

```
-----
-----
----
Nested Loop (actual rows=1799 loops=1)
  Buffers: shared hit=259
  -> Bitmap Heap Scan on flights f (actual rows=51 loops=1)
    Recheck Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
    Heap Blocks: exact=50
    Buffers: shared hit=53
    -> Bitmap Index Scan on flights_scheduled_departure_idx (actual rows=51 loops=1)
      Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp
with time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time
zone))
      Buffers: shared hit=3
    -> Bitmap Heap Scan on boarding_passes bp (actual rows=35 loops=51)
      Recheck Cond: (f.flight_id = flight_id)
      Heap Blocks: exact=49
      Buffers: shared hit=206
      -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (actual rows=35
loops=51)
        Index Cond: (flight_id = f.flight_id)
        Buffers: shared hit=157
  Planning:
    Buffers: shared hit=16
    Planning Time: 0.857 ms
    Execution Time: 0.732 ms
(20 rows)
```

A possible improvement would be to eliminate access to the flights table by performing an index-only scan. Let's remove the previously created index and create a new one including the column required for the join:

```
=> DROP INDEX flights_scheduled_departure_idx;
```

DROP INDEX

```
=> CREATE INDEX ON flights(scheduled_departure, flight_id);
```

CREATE INDEX

```
=> EXPLAIN (analyze, buffers, costs off, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY

PLAN

```
-----
Nested Loop (actual rows=1799 loops=1)
  Buffers: shared hit=207 read=3
  -> Index Only Scan using flights_scheduled_departure_flight_id_idx on flights f
      (actual rows=51 loops=1)
      Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
      Heap Fetches: 0
      Buffers: shared hit=1 read=3
      -> Bitmap Heap Scan on boarding_passes bp (actual rows=35 loops=51)
          Recheck Cond: (f.flight_id = flight_id)
          Heap Blocks: exact=49
          Buffers: shared hit=206
          -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (actual rows=35
loops=51)
              Index Cond: (flight_id = f.flight_id)
              Buffers: shared hit=157

Planning:
  Buffers: shared hit=40 read=1
Planning Time: 0.893 ms
Execution Time: 1.240 ms
(17 rows)
```

We achieved an additional 20% in read operations. This could matter for frequently executed queries, but it's probably not applicable in our case. Including all fields from the boarding_passes table in the index is likely unnecessary — the overhead would exceed the performance benefits. We'll stick with our current results.

Long-running queries

Read large tables in full

Common to OLAP

It's important to avoid reading the same data multiple times

Key Features of the Plan

- low selectivity conditions

- full table scan

- hash join

- aggregation

- parallel execution

Long queries are common in OLAP systems. They involve reading a large amount of data to produce the result. The number of rows returned by the query is irrelevant—aggregation may leave even a single row.

Such queries often involve reading entire large tables, as their conditions typically have low selectivity. Therefore, sequential scanning becomes more efficient than index access, and hash joins replace nested loop joins. Especially because, for a long query, delivering the entire result within a reasonable time is more important than getting the first rows as soon as possible.

It's very important to ensure that the same data isn't read multiple times in the query. This can occur for various reasons (such as correlated subqueries or the use of functions, among others), which result in explicit or implicit nested loops.

Of course, a long query's plan can involve index access (when high-selectivity conditions are present) and nested loop joins (when joining a small number of rows).

Since long queries handle large amounts of data and typically aggregate it (as users rarely need millions of rows), they can benefit from parallel execution.

Long-running Queries

Task: Calculate the number of passengers on flights where both departure and arrival were delayed by between one minute and four hours.

Here's the query that addresses the task, but the run time isn't acceptable.

```
=> \timing on
```

Timing is on.

```
=> SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

```
count
-----
6940272
(1 row)
```

Time: 165926.965 ms (02:45.927)

```
=> \timing off
```

Timing is off.

Is this a short or long-running query?

Determine the selectivity of the conditions:

```
=> SELECT count(*) FROM flights f;
```

```
count
-----
214867
(1 row)
```

```
=> SELECT count(*) FROM flights f
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

```
count
-----
173846
(1 row)
```

The conditions filter about 80% of the rows, and all tables involved in the query are large. It's evident that this is a long-running query.

Check the plan:

```
=> EXPLAIN
SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

QUERY PLAN

```

-----
Aggregate (cost=12247.18..12247.19 rows=1 width=8)
-> Nested Loop (cost=6.22..12246.72 rows=184 width=0)
    Join Filter: (f.flight_id = tf.flight_id)
    -> Nested Loop (cost=5.66..12133.84 rows=184 width=22)
        -> Seq Scan on flights f (cost=0.00..9070.01 rows=5 width=4)
            Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
        -> Bitmap Heap Scan on boarding_passes bp (cost=5.66..611.19 rows=158
width=18)
            Recheck Cond: (f.flight_id = flight_id)
            -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key
(cost=0.00..5.62 rows=158 width=0)
                Index Cond: (flight_id = f.flight_id)
            -> Index Only Scan using ticket_flights_pkey on ticket_flights tf
(cost=0.56..0.60 rows=1 width=18)
                Index Cond: ((ticket_no = bp.ticket_no) AND (flight_id = bp.flight_id))
(12 rows)

```

Here we observe index access and nested loop joins — operations that are entirely inappropriate for a long-running query. What caused this?

The issue stems from an error in estimating condition selectivity. The planner estimates that only five rows will be selected from the flights table.

Let's quickly test the idea that switching to a different join method will make a difference. Instruct the planner to avoid using a nested loop if possible:

```
=> SET enable_nestloop = off;
```

```
SET
```

```

=> EXPLAIN (analyze, buffers, timing off, costs off)
SELECT count(*)
FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
    JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
    AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';

```


QUERY PLAN

```

-----
-----
-----
Aggregate (actual rows=1 loops=1)
  Buffers: shared hit=7065 read=123771, temp read=102066 written=102066
  -> Hash Join (actual rows=6940272 loops=1)
    Hash Cond: ((tf.flight_id = f.flight_id) AND (tf.ticket_no = bp.ticket_no))
    Buffers: shared hit=7065 read=123771, temp read=102066 written=102066
    -> Seq Scan on ticket_flights tf (actual rows=8391852 loops=1)
      Buffers: shared read=69933
    -> Hash (actual rows=6940272 loops=1)
      Buckets: 262144 (originally 1024)  Batches: 64 (originally 1)  Memory
Usage: 7780kB
      Buffers: shared hit=7065 read=53838, temp written=34991
      -> Hash Join (actual rows=6940272 loops=1)
        Hash Cond: (bp.flight_id = f.flight_id)
        Buffers: shared hit=7065 read=53838
        -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
          Buffers: shared hit=4441 read=53838
        -> Hash (actual rows=173846 loops=1)
          Buckets: 262144 (originally 1024)  Batches: 1 (originally 1)
Memory Usage: 8160kB
          Buffers: shared hit=2624
          -> Seq Scan on flights f (actual rows=173846 loops=1)
            Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
            Rows Removed by Filter: 41021
            Buffers: shared hit=2624

Planning:
  Buffers: shared hit=52
Planning Time: 0.539 ms
Execution Time: 27523.514 ms
(26 rows)

```

Table accesses automatically switched to sequential reads. The query ran faster, requiring the reading of approximately 130,000 blocks. We didn't examine the execution plan of the first version with the buffers parameter, but you should check it — it reads over 200 times more data.

```
=> RESET enable_nestloop;
```

```
RESET
```

We still need to clearly explain to the planner which plan to use. The most straightforward approach is to supply more accurate statistics. Expression-based statistics, as discussed in the "Advanced Statistics" section, will suffice here:

```
=> CREATE STATISTICS ON (actual_departure - scheduled_departure) FROM flights;
```

```
CREATE STATISTICS
```

```
=> CREATE STATISTICS ON (actual_arrival - scheduled_arrival) FROM flights;
```

```
CREATE STATISTICS
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

Check the plan:

```

=> EXPLAIN
SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';

```

QUERY PLAN

```
-----  
-----  
-----  
Aggregate (cost=759267.82..759267.83 rows=1 width=8)  
-> Hash Join (cost=11645.81..744796.04 rows=5788709 width=0)  
    Hash Cond: (tf.flight_id = f.flight_id)  
    -> Merge Join (cost=1.20..649810.42 rows=7925944 width=8)  
        Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_id =  
bp.flight_id))  
        -> Index Only Scan using ticket_flights_pkey on ticket_flights tf  
(cost=0.56..292223.96 rows=8391960 width=18)  
        -> Index Only Scan using boarding_passes_pkey on boarding_passes bp  
(cost=0.56..275993.72 rows=7925944 width=18)  
    -> Hash (cost=9070.01..9070.01 rows=156928 width=4)  
        -> Seq Scan on flights f (cost=0.00..9070.01 rows=156928 width=4)  
            Filter: (((actual_departure - scheduled_departure) >=  
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=  
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >=  
'00:01:00'::interval) AND ((actual_arrival - scheduled_arrival) <=  
'04:00:00'::interval))  
(10 rows)
```

The estimates are now accurate, aligning closely with actual values across all nodes. However, the planner selected a different plan than the one we observed: it abandoned the nested loop join but retained index access and used a merge join instead. If we had executed the query, we'd have seen the runtime decrease further, even though over two million pages were read. Is this considered a positive outcome?

If we're confident that the cache size is sufficient to hold the data of our large tables — it's beneficial because the query runs faster. But we've already noted that time is an unreliable indicator, as real-world systems see data from multiple tables competing for cache space, and actual page access could end up being slower than anticipated.

This could prompt us to revisit the database server settings using the `random_page_cost` and `effective_cache_size` parameters. However, we should ask ourselves: are we reading unnecessary data?

Leveraging domain knowledge, we can conclude that the `ticket_flights` table is unnecessary in the query. The `flights` table can be directly joined with `boarding_passes` — foreign keys ensure data integrity but aren't required for the join. Adding `ticket_flights` imposes no restrictions on the data (there's no scenario where a boarding pass exists without a corresponding flight).

Generally, reformulating the query is the most varied and flexible approach to influencing performance. While the planner can handle some equivalent transformations on its own, we can:

- rewrite the conditions to enable (or prevent) index usage
- Replace correlated subqueries—which are essentially nested loops—with joins;
- Eliminate unnecessary scans (such as in this example, or by using window functions);
- use transformations that the planner can't handle, like converting OR conditions into UNION operations;
- and so forth

Let's simplify the query

```
=> EXPLAIN (analyze, buffers, timing off, costs off)  
SELECT count(*)  
FROM flights f  
    JOIN boarding_passes bp ON bp.flight_id = f.flight_id  
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'  
    AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

QUERY PLAN

```

-----
Aggregate (actual rows=1 loops=1)
  Buffers: shared hit=7097 read=53806, temp read=11797 written=11797
  -> Hash Join (actual rows=6940272 loops=1)
    Hash Cond: (bp.flight_id = f.flight_id)
    Buffers: shared hit=7097 read=53806, temp read=11797 written=11797
    -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
      Buffers: shared hit=4473 read=53806
    -> Hash (actual rows=173846 loops=1)
      Buckets: 262144 Batches: 2 Memory Usage: 5118kB
      Buffers: shared hit=2624, temp written=253
      -> Seq Scan on flights f (actual rows=173846 loops=1)
        Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
        Rows Removed by Filter: 41021
        Buffers: shared hit=2624

Planning:
  Buffers: shared hit=16
Planning Time: 0.301 ms
Execution Time: 5891.453 ms
(18 rows)

```

Only 61,000 pages needed to be read. Let's check the total number of pages in the tables:

```

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

sum
-----
60903
(1 row)

```

Now we're confident we're not reading extra data.

Is there anything else we can improve?

The query plan shows that a two-pass hash join was used. Increasing the `work_mem` parameter can allow the hash join to complete in a single pass without relying on temporary files.

```

=> SET work_mem = '8MB'; -- Consider increasing this for the entire server

```

SET

```

=> EXPLAIN (analyze, timing off, costs off)
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';

```

QUERY PLAN

```

-----
Aggregate (actual rows=1 loops=1)
  -> Hash Join (actual rows=6940272 loops=1)
        Hash Cond: (bp.flight_id = f.flight_id)
        -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
        -> Hash (actual rows=173846 loops=1)
              Buckets: 262144 Batches: 1 Memory Usage: 8160kB
              -> Seq Scan on flights f (actual rows=173846 loops=1)
                    Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
                    Rows Removed by Filter: 41021
Planning Time: 1.083 ms
Execution Time: 4026.004 ms
(11 rows)

```

Earlier in this section, we turned off parallel plans. However, for long-running queries, parallel plans can be beneficial. Of course, if there are available CPU cores, as discussed in the "Parallel Access" section. Let's take a look:

=> **RESET** max_parallel_workers_per_gather;

RESET

=> **EXPLAIN** (analyze, timing off, costs off)

SELECT count(*)

FROM flights f

JOIN boarding_passes bp **ON** bp.flight_id = f.flight_id

WHERE f.actual_departure - f.scheduled_departure **BETWEEN** interval '1 min' **AND** interval '4 hours'

AND f.actual_arrival - f.scheduled_arrival **BETWEEN** interval '1 min' **AND** interval '4 hours';

QUERY PLAN

```

-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=2313424 loops=3)
                    Hash Cond: (bp.flight_id = f.flight_id)
                    -> Parallel Seq Scan on boarding_passes bp (actual rows=2641937
loops=3)
                    -> Parallel Hash (actual rows=57949 loops=3)
                          Buckets: 262144 Batches: 1 Memory Usage: 8896kB
                          -> Parallel Seq Scan on flights f (actual rows=57949 loops=3)
                                Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
                                Rows Removed by Filter: 13674
Planning Time: 4.228 ms
Execution Time: 5006.774 ms
(15 rows)

```

We see that the planner uses a parallel execution plan, but in a single-core virtual machine, it doesn't offer any benefits.

Not explicitly present

While there are ways to influence execution, such as configuration parameters, CTE materialization, and other techniques.

Third-Party Extensions

`pg_hint_plan`

Another (traditional in other DBMS) way to influence—optimizer hints—is not present in PostgreSQL. This is a core decision made by the community: <https://wiki.postgresql.org/wiki/OptimizerHintsDiscussionn>.

Actually, some hints are still implicitly present in the form of configuration parameters and other mechanisms.

Additionally, there are specific extensions, such as <https://postgrespro.ru/docs/enterprise/16/pg-hint-plan> (author: Kyotaro Horiguchi). Don't forget that using hints that severely restrict the planner's flexibility can backfire if data distribution changes in the future.

System configuration is handled at multiple levels

There are a variety of ways to influence the query execution plan

Different query types require different approaches

Nothing can replace a sharp mind and good judgment

1. Optimize the query that retrieves contact information for passengers who bought business class tickets on flights delayed by more than 5 hours: `SELECT t.*FROM tickets t JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no JOIN flights f ON f.flight_id = tf.flight_idWHERE tf.fare_conditions = 'Business' AND f.actual_departure > f.scheduled_departure + interval '5 hour';`Optimize the query that retrieves contact information for passengers who bought business class tickets on flights delayed by more than 5 hours: `SELECT t.*FROM tickets t JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no JOIN flights f ON f.flight_id = tf.flight_idWHERE tf.fare_conditions = 'Business' AND f.actual_departure > f.scheduled_departure + interval '5 hour';`
2. Optimize the query that computes the average ticket price for flights operated by various aircraft types.Begin with the version proposed in the comment.

2.

```
SELECT a.aircraft_code, ( SELECT round(avg(tf.amount)) FROM
flights f JOIN ticket_flights tf ON tf.flight_id = f.flight_id
WHERE f.aircraft_code = a.aircraft_code)FROM aircrafts a
```

1. Optimizing a Short Query. Disable parallel execution. This is an OLTP query involving a small number of rows, where only a portion of the data needs to be retrieved. Therefore, the general optimization approach is to shift from full table scans and hash joins to indexes and nested loops. Note that there's only one flight delayed by more than five hours, and the query planner is scanning the entire table. You can create an index on an expression based on the difference between two columns and slightly adjust the condition:

Turn off parallel execution.

```
=> SET max_parallel_workers_per_gather = 0;
```

SET

```
=> EXPLAIN (analyze, timing off)
```

```
SELECT t.*
```

```
FROM tickets t
```

```
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
```

```
JOIN flights f ON f.flight_id = tf.flight_id
```

```
WHERE tf.fare_conditions = 'Business'
```

```
AND f.actual_departure > f.scheduled_departure + interval '5 hour';
```

QUERY PLAN

```
-----
Merge Join (cost=214793.36..366306.55 rows=287283 width=104) (actual rows=2 loops=1)
  Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t (cost=0.43..139111.54 rows=2949850
width=104) (actual rows=1336684 loops=1)
    -> Materialize (cost=214792.93..216229.35 rows=287283 width=14) (actual rows=2
loops=1)
    -> Sort (cost=214792.93..215511.14 rows=287283 width=14) (actual rows=2
loops=1)
      Sort Key: tf.ticket_no
      Sort Method: quicksort Memory: 25kB
      -> Hash Join (cost=6742.28..183837.19 rows=287283 width=14) (actual
rows=2 loops=1)
        Hash Cond: (tf.flight_id = f.flight_id)
        -> Seq Scan on ticket_flights tf (cost=0.00..174832.50 rows=861854
width=18) (actual rows=859656 loops=1)
          Filter: ((fare_conditions)::text = 'Business'::text)
          Rows Removed by Filter: 7532196
        -> Hash (cost=5847.00..5847.00 rows=71622 width=4) (actual rows=1
loops=1)
          Buckets: 131072 Batches: 1 Memory Usage: 1025kB
          -> Seq Scan on flights f (cost=0.00..5847.00 rows=71622
width=4) (actual rows=1 loops=1)
            Filter: (actual_departure > (scheduled_departure +
'05:00:00'::interval))
            Rows Removed by Filter: 214866
Planning Time: 15.914 ms
Execution Time: 4834.990 ms
(19 rows)
```

This is an OLTP query involving a small number of rows, where only a portion of the data needs to be retrieved. Therefore, the general optimization approach is to shift from full table scans and hash joins to indexes and nested loops.

Notice that only one flight is delayed by more than five hours, yet the query planner is scanning the entire table. You could create an index on an expression that calculates the difference between two columns and slightly modify the condition:

```
=> CREATE INDEX ON flights ((actual_departure - scheduled_departure));
```

CREATE INDEX

```
=> ANALYZE flights;
```

ANALYZE

```
=> EXPLAIN (analyze, timing off)
```

```
SELECT t.*
```

```
FROM tickets t
```

```
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
```

```
JOIN flights f ON f.flight_id = tf.flight_id
```

```
WHERE tf.fare_conditions = 'Business'
```

```
AND f.actual_departure - f.scheduled_departure > interval '5 hour';
```


QUERY PLAN

```
-----
Nested Loop (cost=12.81..177113.12 rows=8 width=104) (actual rows=2 loops=1)
  -> Hash Join (cost=12.38..177107.29 rows=8 width=14) (actual rows=2 loops=1)
    Hash Cond: (tf.flight_id = f.flight_id)
    -> Seq Scan on ticket_flights tf (cost=0.00..174832.50 rows=861854 width=18)
(actual rows=859656 loops=1)
      Filter: ((fare_conditions)::text = 'Business'::text)
      Rows Removed by Filter: 7532196
    -> Hash (cost=12.35..12.35 rows=2 width=4) (actual rows=1 loops=1)
      Buckets: 1024 Batches: 1 Memory Usage: 9kB
      -> Index Scan using flights_expr_idx on flights f (cost=0.42..12.35
rows=2 width=4) (actual rows=1 loops=1)
        Index Cond: ((actual_departure - scheduled_departure) >
'05:00:00'::interval)
      -> Index Scan using tickets_pkey on tickets t (cost=0.43..0.73 rows=1 width=104)
(actual rows=1 loops=2)
        Index Cond: (ticket_no = tf.ticket_no)
Planning Time: 72.253 ms
Execution Time: 2639.410 ms
(14 rows)
```

Now let's focus on the `ticket_flights` table, which is also being fully scanned, even though only a small fraction of its rows are actually read.

An index on `fare_conditions` could be beneficial, but a more effective approach would be to create an index on the `flight_id` column, enabling efficient nested loop joins with `flights`:

```
=> CREATE INDEX ON ticket_flights(flight_id);
```

```
CREATE INDEX
```

```
=> EXPLAIN (analyze, timing off)
```

```
SELECT t.*
```

```
FROM tickets t
```

```
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
```

```
JOIN flights f ON f.flight_id = tf.flight_id
```

```
WHERE tf.fare_conditions = 'Business'
```

```
AND f.actual_departure - f.scheduled_departure > interval '5 hour';
```

QUERY PLAN

```
-----
Nested Loop (cost=1.28..825.43 rows=8 width=104) (actual rows=2 loops=1)
  -> Nested Loop (cost=0.85..819.59 rows=8 width=14) (actual rows=2 loops=1)
    -> Index Scan using flights_expr_idx on flights f (cost=0.42..12.35 rows=2
width=4) (actual rows=1 loops=1)
      Index Cond: ((actual_departure - scheduled_departure) >
'05:00:00'::interval)
    -> Index Scan using ticket_flights_flight_id_idx on ticket_flights tf
(cost=0.43..403.51 rows=11 width=18) (actual rows=2 loops=1)
      Index Cond: (flight_id = f.flight_id)
      Filter: ((fare_conditions)::text = 'Business'::text)
      Rows Removed by Filter: 10
  -> Index Scan using tickets_pkey on tickets t (cost=0.43..0.73 rows=1 width=104)
(actual rows=1 loops=2)
    Index Cond: (ticket_no = tf.ticket_no)
Planning Time: 0.527 ms
Execution Time: 42.955 ms
(12 rows)
```

The execution time was reduced to milliseconds.

2. Optimizing Long-Running Queries

It's clear that we're dealing with a long-running query: it includes the large `'ticket_flights'` table, but no conditions are applied to any tables. Let's execute the query and examine its execution plan:

```
=> EXPLAIN (analyze, buffers, timing off)
SELECT a.aircraft_code, (
  SELECT round(avg(tf.amount))
  FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.aircraft_code = a.aircraft_code
)
FROM aircrafts a;
```

QUERY PLAN

```
-----
Seq Scan on aircrafts_data ml (cost=0.00..1657338.13 rows=9 width=36) (actual rows=9
loops=1)
  Buffers: shared hit=24480 read=558601
  SubPlan 1
    -> Aggregate (cost=184148.55..184148.56 rows=1 width=32) (actual rows=1 loops=9)
      Buffers: shared hit=24480 read=558600
      -> Hash Join (cost=5645.56..181526.13 rows=1048967 width=6) (actual
rows=932428 loops=9)
        Hash Cond: (tf.flight_id = f.flight_id)
        Buffers: shared hit=24480 read=558600
        -> Seq Scan on ticket_flights tf (cost=0.00..153851.52 rows=8391852
width=10) (actual rows=8391852 loops=8)
          Buffers: shared hit=864 read=558600
          -> Hash (cost=5309.84..5309.84 rows=26858 width=4) (actual rows=23874
loops=9)
            Buckets: 65536 (originally 32768) Batches: 1 (originally 1)
            Buffers: shared hit=23616
            -> Seq Scan on flights f (cost=0.00..5309.84 rows=26858 width=4)
(actual rows=23874 loops=9)
              Filter: (aircraft_code = ml.aircraft_code)
              Rows Removed by Filter: 190993
              Buffers: shared hit=23616

Planning:
  Buffers: shared hit=54 read=8
Planning Time: 3.707 ms
Execution Time: 35372.438 ms
(21 rows)
```

It's clear from the query plan that cardinality estimates are accurate across all nodes, large tables are scanned sequentially, and a hash join is used.

However, the query ends up reading several times more pages than necessary:

```
=> SELECT sum(relpages) FROM pg_class
WHERE relname IN ('flights', 'ticket_flights', 'aircrafts_ml');

 sum
-----
72557
(1 row)
```

The issue stems from a correlated subquery that runs once per each of the nine aircraft models, resulting in an implicit nested loop. To address this, rewrite the subquery by adding a grouping clause.

It might be tempting to remove the aircrafts table, as the aircraft code can be directly obtained from the flights table. However, this would exclude the model with no flights from the results. Therefore, a left join is required.

Additionally, it's worth noting that during the demonstration, we increased work_mem to avoid a two-pass join.

```
=> SET work_mem = '8MB';

SET

=> EXPLAIN (analyze, buffers, timing off)
SELECT a.aircraft_code, round(avg(tf.amount))
FROM aircrafts a
  LEFT JOIN flights f ON f.aircraft_code = a.aircraft_code
  LEFT JOIN ticket_flights tf ON tf.flight_id = f.flight_id
GROUP BY a.aircraft_code;
```

QUERY PLAN

```

-----
HashAggregate (cost=257701.41..257701.54 rows=9 width=36) (actual rows=9 loops=1)
  Group Key: ml.aircraft_code
  Batches: 1 Memory Usage: 24kB
  Buffers: shared hit=2765 read=69793
  -> Hash Right Join (cost=7459.71..215742.15 rows=8391852 width=10) (actual
rows=8456132 loops=1)
    Hash Cond: (f.aircraft_code = ml.aircraft_code)
    Buffers: shared hit=2765 read=69793
    -> Hash Right Join (cost=7458.51..183339.07 rows=8391852 width=10) (actual
rows=8456131 loops=1)
      Hash Cond: (tf.flight_id = f.flight_id)
      Buffers: shared hit=2764 read=69793
      -> Seq Scan on ticket_flights tf (cost=0.00..153851.52 rows=8391852
width=10) (actual rows=8391852 loops=1)
        Buffers: shared hit=140 read=69793
        -> Hash (cost=4772.67..4772.67 rows=214867 width=8) (actual rows=214867
loops=1)
          Buckets: 262144 Batches: 1 Memory Usage: 10442kB
          Buffers: shared hit=2624
          -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=8)
(actual rows=214867 loops=1)
            Buffers: shared hit=2624
            -> Hash (cost=1.09..1.09 rows=9 width=4) (actual rows=9 loops=1)
              Buckets: 1024 Batches: 1 Memory Usage: 9kB
              Buffers: shared hit=1
              -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 width=4)
(actual rows=9 loops=1)
                Buffers: shared hit=1
Planning:
  Buffers: shared hit=26 read=2
Planning Time: 27.118 ms
Execution Time: 13180.862 ms
(26 rows)

```

The unnecessary page reads are gone, and the query has speeded up.