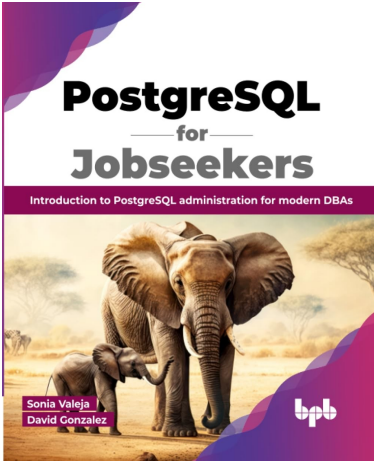
Database Administration

Lecture 07: Performance Tuning.

Valeja & Gonzales.

8 de septiembre de 2025

Database Administration: Performance Tuning.



Content has been extracted from *PostgreSQL for Jobseekers (Chapter 13)*, by Sonia Valeja and David Gonzales, 2023. Visit https://bpbonline.com/products/postgresql-for-jobseekers.

Plan

Introduction to Performance Tuning

Any **Relational Database Management System (RDBMS)** intends to resolve user queries as quickly as possible, and PostgreSQL is no exception. Keeping PostgreSQL performant is one of the main goals for Database Administrators (DBAs).

Now, we will dive into the **performance tuning topics**. We will learn about the basic concepts and components involved in database tuning and some best practices we can use as an initial guide.

Introduction to Performance Tuning

- PostgreSQL aims to resolve queries as quickly as possible.
- Performance tuning is a critical aspect of database administration.
- This presentation covers key topics including: indexes, statistics, query planning, and best practices.

Plan

Indexes

- Indexes help locate data efficiently, similar to a book index.
- Types of indexes:
  - B-tree (default, supports range queries)
  - Hash (optimized for equality comparisons)
  - GiST and SP-GiST (used for geometric data)
  - GIN (for arrays and full-text search)
  - BRIN (efficient for large datasets with ordered values)

Indexes

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
[ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass
    [ ( opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS
    { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

```
CREATE INDEX <schema_name>.<index_name> on <table_name> (<column/column-list>)
```

Index Creation Notes

- **CREATE** and **DROP** indexes are not online activities by default. The table is locked for modifications during these operations.
- Read operations can still be performed during index creation/modification if the `CONCURRENTLY` keyword is used.
- Multi-column indexes can be created to improve performance on queries involving multiple columns.
- When creating multi-column indexes, ensure the column order matches the order used in the query's `WHERE` clause.

Reindexing

- Rebuilding an index can help maintain performance.
- Command: `REINDEX INDEX <index-name>`
- Use `CONCURRENTLY` to rebuild without locking writes.

Indexes

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }
[ CONCURRENTLY ] name
```

where option can be one of:

```
CONCURRENTLY [ boolean ] { Reindex can be created concurrently online
TABLESPACE new_tablespace { Reindexing could be done in tablespace new
VERBOSE [ boolean ] { Logs will be printed while reindexing
```

Automatic Index Creation

- When a **Primary Key** or **Unique Key** constraint is created, PostgreSQL automatically creates indexes on those columns.
- These columns are frequently used for fetching data in queries.
- It is also a good practice to create indexes on:
  - Columns used in `WHERE` clauses.
  - Foreign key columns in child tables (often used in `JOIN` conditions).

Index Usage Over Time

- For small databases, default indexes (Primary Key, Unique) are often enough.
- As data grows, queries may slow down, making additional indexes useful.
- Use `EXPLAIN PLAN` to verify whether queries benefit from indexes.
- Indexes are essential building blocks for improving performance of:
  - Reporting queries.
  - `SELECT` queries requiring faster execution.

Indexes and Query Performance

- Indexes can also improve the performance of `INSERT`, `UPDATE`, and `DELETE` queries with `WHERE` conditions.
- The `ANALYZE` command keeps index statistics up to date.
- Updated statistics ensure the query planner chooses the most efficient execution plan.

Indexes and Expressions

- An index column does not need to be a raw table column.
- It can be a **function** or **scalar expression** computed from one or more columns.
- This allows PostgreSQL to optimize queries involving computed values.
- Example: concatenate `first_name` and `last_name`.

Example Query

```
SELECT * FROM people
WHERE (first_name || ' ' || last_name) = 'John Smith';
```

Expression Index Benefits

- Create an index on the same expression used in queries:

Example Index

```
CREATE INDEX people_names
  ON people ((first_name || ' ' || last_name));
```

- The system treats the expression as a normal indexed column.
- Search speed becomes equivalent to simple index lookups.
- Useful when retrieval speed is more important than insertion or update speed.
- Metadata views `pg_index` and `pg_indexes` provide details on schema, index name, and more.

Plan

Statistics in PostgreSQL

- Statistics are **metadata** — data about the data.
- The **query planner** relies on statistics to determine the fastest and cheapest execution path.
- Collected statistics guide how PostgreSQL retrieves the required data efficiently.
- Two main types of statistics exist, maintained by different components.

Types of Statistics

- **Server activity statistics**
  - Collected by the **stats collector** background process.
  - Includes: table/index access counts, user session information.
  - Used mainly by DBAs to verify system state.
- **Data distribution statistics**
  - Collected by `ANALYZE` or `VACUUM ANALYZE`, manually or via autovacuum.
  - Provides data distribution details for the planner.
  - Directly impacts query execution decisions.

System Catalogs for Statistics

- **pg_class**: Stores statistics about number of tuples (rows), disk blocks, and object identity details.
- **pg_statistics**: Contains selectivity information per table column.
  - One or two rows per column (depending on inheritance).
  - Intended for system use, but `pg_stats` view provides a user-friendly version.
- **pg_statistics_ext_data**: Stores additional information for extended statistics objects on sets of columns.

Statistics in pg_class

```
SELECT
    relname AS relation_name,
    relkind AS relation_kind,
    reltuples AS relation_tuples,
    relpages AS relation_pages,
    pg_size_pretty(pg_relation_size(oid)) AS relation_size
FROM
    pg_class
WHERE
    relname LIKE 'address%'
    AND relkind IN ('r', 'i');
```

Statistics in pg_class

```
-[ RECORD 1 ]---+-------------
relation_name| address
relation_kind| r
relation_tuples | 603
relation_pages| 8
relation_size| 64 kB
-[ RECORD 2 ]---+-------------
relation_name| address_pkey
relation_kind| i
relation_tuples | 603
relation_pages| 4
relation_size| 32 kB
```

Statistics in pg_statistics (pg_stats view)

```sql
SELECT
    attname AS column_name,
    n_distinct AS distinct_rate,
    array_to_string(most_common_vals, E'\n') AS most_common_values,
    array_to_string(most_common_freqs, E'\n') AS most_common_frequencies
FROM
    pg_stats
WHERE tablename = 'address'
    AND attname = 'postal_code';
```

Statistics in pg_class

```
-[ RECORD 1 ]----------+-------------
column_name            | postal_code
distinct_rate          | 0.9900498
most_common_values     |   +
                       | 22474+
                       | 52137+
                       | 9668
most_common_frequencies| 0.006633499 +
                       | 0.0033167496+
                       | 0.0033167496+
                       | 0.0033167496
```

Extended Statistics in PostgreSQL

- Standard statistics track rows, disk space, and column selectivity.
- Some queries involve correlations across multiple columns, which single-column stats cannot capture.
- PostgreSQL supports **extended statistics**, but they are not gathered by default.
- DBAs must define them explicitly using:
  - `CREATE STATISTICS` command.
- Goal: describe correlations between columns to improve the planner's row estimations.
- Requires human intervention since relationships depend on the database model.

Extended Statistics in PostgreSQL

```
pagila=# \h CREATE STATISTICS
Command:     CREATE STATISTICS
Description: define extended statistics
Syntax:
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    ON ( expression )
    FROM table_name

CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON { column_name | ( expression ) }, { column_name | ( expression ) } [, ...]
    FROM table_name

URL: https://www.postgresql.org/docs/14/sql-createstatistics.html
```

Homework: Summarize sections about the three types of extended statistics[1], each covering a distinct kind of column values correlation:

1. Functional dependencies.
2. Number of distinct values counts.
3. Most common values list.

[1] page 243 in the textbook

EXPLAIN PLAN

- Used to analyze query execution plans.
- Syntax: `EXPLAIN ANALYZE <query>`
- Helps identify if indexes are used or if sequential scans occur.

EXPLAIN PLAN

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

where option can be one of:

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
SETTINGS [ boolean ]
BUFFERS [ boolean ]
WAL [ boolean ]
TIMING [ boolean ]
SUMMARY [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

In action

```
postgres=# EXPLAIN (ANALYZE,BUFFERS)
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                                    QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
-----------
 Nested Loop  (cost=10000000000.13..10000000009.22 rows=1 width=668) (actual time=0.068..0.077 rows=2 loops=1)
   Buffers: shared hit=5
   ->  Seq Scan on emp e  (cost=10000000000.00..10000000001.06 rows=1 width=300) (actual time=0.034..0.037 rows=
2 loops=1)
         Filter: (dept_id = '2'::numeric)
         Rows Removed by Filter: 3
         Buffers: shared hit=1
   ->  Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368) (actual time=0.017..0.017 rows=1
loops=2)
         Index Cond: (dept_id = '2'::numeric)
         Buffers: shared hit=4
 Planning Time: 0.371 ms
 Execution Time: 0.149 ms
(11 rows)
```

In action

```
postgres=# EXPLAIN
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Nested Loop  (cost=0.26..16.31 rows=1 width=668)
   ->  Index Scan using indx_emp_dept_id on emp e  (cost=0.13..8.15 rows=1 width=300)
         Index Cond: (dept_id = '2'::numeric)
   ->  Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368)
         Index Cond: (dept_id = '2'::numeric)
(5 rows)
```

Plan

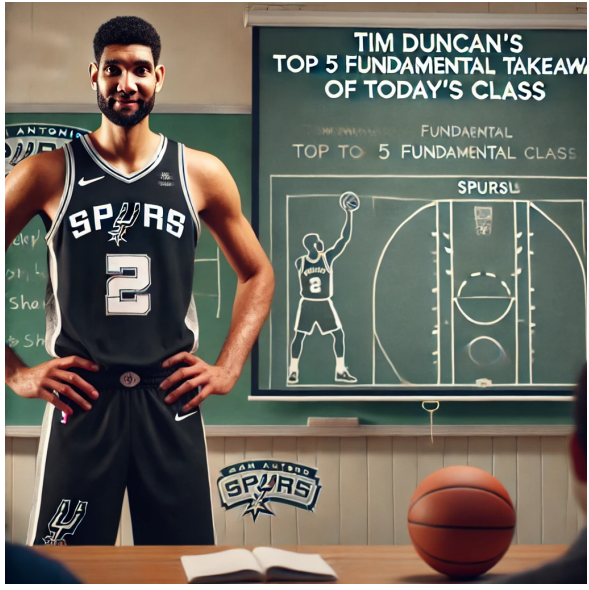
Best Practices for Performance Tuning

- Adjust key parameters in `postgresql.conf`:
  - `shared_buffers`: Set to 15-25 % of RAM.
  - `work_mem`: Allocate sufficient memory for sorting.
  - `effective_cache_size`: Set to 50-75 % of RAM.
  - `autovacuum`: Ensure it is enabled for table maintenance.
- Regularly analyze and vacuum tables.
- Use indexes strategically to avoid unnecessary overhead.

Best Practices for Performance Tuning

Parameter	Default values	Best practice value
`shared_buffers`	128 MB	Between 15 - 25% of total RAM
`work_mem`	4 MB	25% of total RAM / max_connextions
`autovacuum`	ON	ON
`effective_cache_size`	4 GB	Between 50% - 75% of total RAM
`maintenance_work_mem`	64 MB	Between 5% - 10% of total RAM
`max_connections`	100	Up to 20 per CPU

TDT5FTOTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAMENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURS
SPURS
2
SPURS

TDT5FTOTC

TDT5FTOTC

5 **Indexes are the foundation of query performance**, with PostgreSQL creating them automatically for Primary and Unique Keys while DBAs should add them strategically for foreign keys and frequent WHERE clauses.

TDT5FTOTC

5 **Indexes are the foundation of query performance**, with PostgreSQL creating them automatically for Primary and Unique Keys while DBAs should add them strategically for foreign keys and frequent WHERE clauses.

4 **Indexes improve SELECT performance** but add overhead to writes, so their effectiveness should be checked with `EXPLAIN PLAN` and maintained using `ANALYZE`.

TDT5FTOTC

5 **Indexes are the foundation of query performance**, with PostgreSQL creating them automatically for Primary and Unique Keys while DBAs should add them strategically for foreign keys and frequent WHERE clauses.

4 **Indexes improve SELECT performance** but add overhead to writes, so their effectiveness should be checked with `EXPLAIN PLAN` and maintained using `ANALYZE`.

3 PostgreSQL relies on **statistics as metadata** for query planning, with server activity statistics and data distribution statistics stored in catalogs like `pg_class`, `pg_stats`, and `pg_statistics_ext_data`.

TDT5FTOTC

5 **Indexes are the foundation of query performance**, with PostgreSQL creating them automatically for Primary and Unique Keys while DBAs should add them strategically for foreign keys and frequent WHERE clauses.

4 **Indexes improve SELECT performance** but add overhead to writes, so their effectiveness should be checked with `EXPLAIN PLAN` and maintained using `ANALYZE`.

3 PostgreSQL relies on **statistics as metadata** for query planning, with server activity statistics and data distribution statistics stored in catalogs like `pg_class`, `pg_stats`, and `pg_statistics_ext_data`.

2 **Extended statistics** defined with `CREATE STATISTICS` improve multi-column query optimization by capturing functional dependencies, distinct value counts, and common value lists.

TDT5FTOTC

5 **Indexes are the foundation of query performance**, with PostgreSQL creating them automatically for Primary and Unique Keys while DBAs should add them strategically for foreign keys and frequent WHERE clauses.

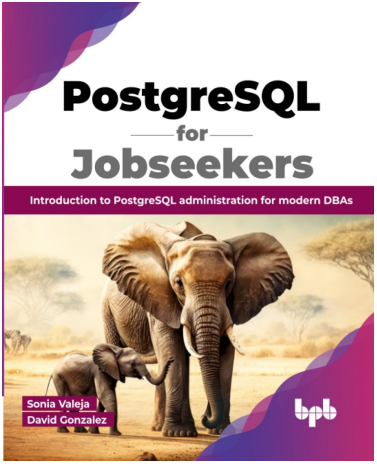
4 **Indexes improve SELECT performance** but add overhead to writes, so their effectiveness should be checked with `EXPLAIN PLAN` and maintained using `ANALYZE`.

3 PostgreSQL relies on **statistics as metadata** for query planning, with server activity statistics and data distribution statistics stored in catalogs like `pg_class`, `pg_stats`, and `pg_statistics_ext_data`.

2 **Extended statistics** defined with `CREATE STATISTICS` improve multi-column query optimization by capturing functional dependencies, distinct value counts, and common value lists.

1 **Best practices** include tuning key parameters in `postgresql.conf`, keeping autovacuum enabled, running `ANALYZE`/`VACUUM` regularly, and using indexes strategically.

Database Administration: Backup and Restore.



Content has been extracted from *PostgreSQL for Jobseekers (Chapter 13)*, by Sonia Valeja and David Gonzales, 2023. Visit https://bpbonline.com/products/postgresql-for-jobseekers.