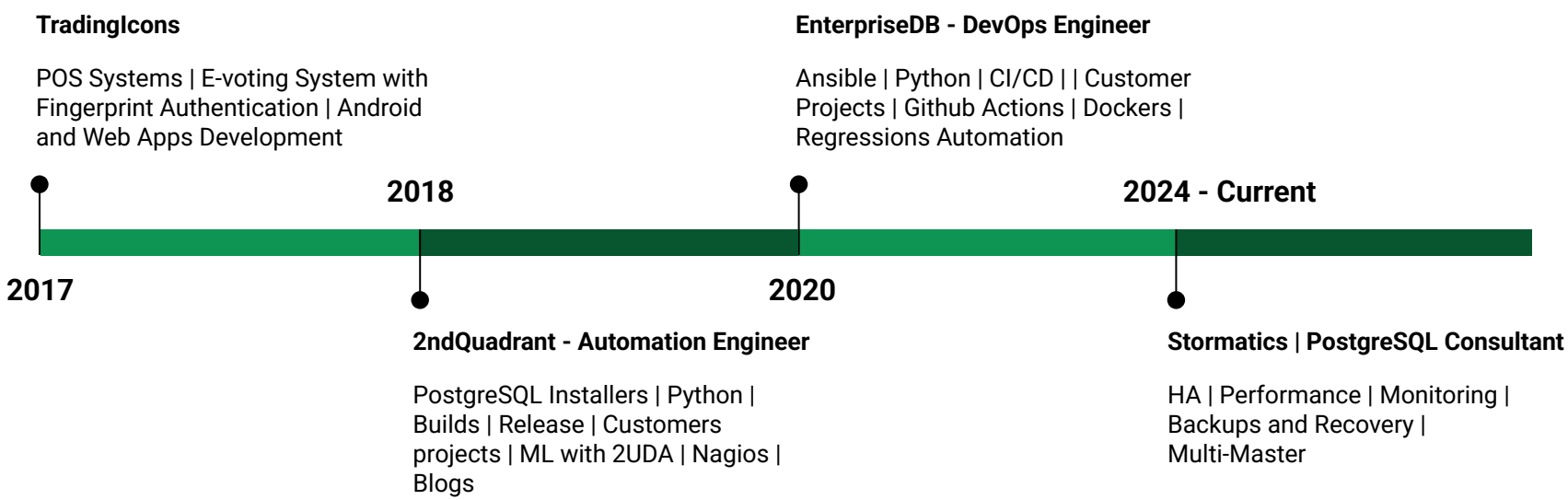
Boost Postgres Performance & Reduce Operational Costs

5 May 2025

STORMATICS

Semab Tariq | PostgreSQL Consultant



STORMATICS

Agenda

01 Database Performance: What You Need to Know?

02 PostgreSQL Performance Optimization

- 2.1 Indexing
- 2.2 Partitioning
- 2.3 PostgreSQL Parameters
- 2.4 JIT (Just-In-Time Compilation)
- 2.5 Query Restructuring
- 2.6 Explain and Explain Analyze

03 Q&A

01 Database Performance: What You Need to Know?

What Does Database Performance Mean?

- How efficiently a database processes queries and handles workloads.

How do we measure performance?

- Query response time
- Throughput
- Resource utilization
  - CPU
  - Memory and
  - Disk I/O

Source: monkeyconsultancy

01 Database Performance: What You Need to Know?

A well-performing database should

- Execute queries quickly, even under heavy load.
- Efficiently handle concurrent user requests.
- Maintain low latency while reading and writing data.
- Scale effectively as data grows

01 Database Performance: What You Need to Know?

Performance Tuning vs. Performance Optimization – Are They the Same?

- Performance Tuning
  - Adjusting database settings to improve efficiency.
- Performance Optimization
  - A broader approach that includes design changes, indexing strategies, query rewriting, and resource management.

In practice, many people use these terms interchangeably, but tuning is often reactive, while optimization is proactive.

01 Database Performance: What You Need to Know?

Which Businesses Need High-Performance Databases?

- High-Traffic Applications
  - E-commerce, social media
- Real-Time Analytics & Monitoring
  - Financial trading, IoT apps
- Low-Latency Requirements
  - Gaming, telecommunications, ad tech
- Large-Scale Transactional Systems
  - Banking, payments

In banking, where even a tiny data loss is unacceptable, sometimes data integrity takes priority over performance

01 Database Performance: What You Need to Know?

Balancing Performance vs. Cost – When to Scale vs. When to Enhance Hardware

When to Optimize First

- Slow Queries & Inefficient Indexing
  - EXPLAIN ANALYZE
- High Connection Overhead
  - Connection pooling
- Table Bloat & Dead Tuples
  - Tune autovacuum
- Unoptimized Workload
  - Partitioning

When to Scale or Upgrade Hardware

- CPU Bottlenecks
- Memory Constraints
- Disk I/O Issues
- High Availability & Load Distribution

02 PostgreSQL Performance Optimization

Indexes

- Indexes in PostgreSQL are special database objects that improve the speed of data retrieval operations on a table.
- `CREATE INDEX index_name ON table_name (column_name);`

Types of Indexes

- B-tree
  - The default and most commonly used type, ideal for equality, range and Pattern queries.
- Hash
  - Useful for equality comparisons.
  - They use a hash function to directly locate the desired value, avoiding the need for tree traversal.
  - `explain analyze SELECT * FROM orders WHERE customer_id = 10;`
  - `explain analyze SELECT * FROM orders WHERE customer_id > 0 AND customer_id < 10;`

02 PostgreSQL Performance Optimization

Types of Indexes

- GIN (Generalized Inverted Index)
  - Efficient for indexing composite values, such as arrays.
- BRIN (Block Range INdex)
  - Compact indexes that are efficient for large tables where the data is naturally ordered.
- GiST (Generalized Search Tree)
  - Can handle various types of queries, including geometric data.

02 PostgreSQL Performance Optimization

Partial Indexes

- Partial indexes are indexes that include only a subset of rows in a table, based on a specified condition.

Benefits

- Reduced Index Size
  - Smaller index size compared to a full index, leading to lower storage requirements.
- Improved Performance
  - Faster index scans for queries that match the index condition.
- Efficient Maintenance
  - Less overhead for index maintenance operations like updates and inserts.

02 PostgreSQL Performance Optimization

Indexing Best Practices

- Use composite Indexes
  - If your queries often filter by multiple columns, consider creating a composite index order.
  - **NOTE**: The order of columns in a composite index matters. Place the most selective column first.
- Avoid Over-Indexing
  - Avoid Over-Indexing Each index adds overhead for insert, update, and delete operations.
- Use Indexes for Foreign Keys
  - Index foreign key columns to speed up join operations.
- Avoid index
  - If workload is write heavy and perform relatively few read operations.

02 PostgreSQL Performance Optimization

Success Story: Optimizing Complex Queries for a Customer

Challenge

- Customer had a batch of three large queries (500+ lines each) that took 32 hours to run in production.

Our Approach

- Analyzed queries to identify the common filters used in WHERE clauses.
- Created targeted indexes to optimize filtering and reduce scan times.
- Avoided direct query modifications to ensure business logic remained intact.

Results

- Batch completion time reduced from 32 hours to 7 hours 18 minutes – a **4.3x** improvement!
- Business logic remained untouched while achieving massive performance gains

02 PostgreSQL Performance Optimization

Partitioning

- A technique to divide a large table into smaller, more manageable pieces.

Types of Partitioning

- Range Partitioning
  - Data is divided based on a value range (e.g., date ranges).
- List Partitioning
  - Data is split based on specific values (e.g., country names).
- Hash Partitioning
  - Data is distributed based on a hash function (e.g., even distribution).

When to Use Partitioning?

- When dealing with large tables (millions or billions of rows).
- When queries frequently filter on a column (e.g., date-based filtering).
- When archiving or deleting old data is required efficiently.
- Blog: When HASH partitioning works better than RANGE

02 PostgreSQL Performance Optimization

PostgreSQL Parameter Tuning

Adjust configuration settings to optimize database performance based on workload requirements

- Memory Parameters
- Parallelism Parameters
- JIT-Related Parameters
- Connection-Related Parameters
- Autovacuum Parameters

Blog 1: Important PostgreSQL Parameters: Understanding Their Importance and Recommended Values

Blog 2: Leveraging autovacuum in PostgreSQL to optimize performance and reduce costs

02 PostgreSQL Performance Optimization

Just-In-Time (JIT) Compilation

JIT (Just-In-Time) compilation optimizes query execution by compiling parts of SQL queries into native machine code at runtime.

Which Queries Benefit from JIT?

- Queries with heavy aggregations (e.g., SUM(), AVG(), COUNT() on large datasets).
- CPU bound queries

When JIT Can Overburden You?

- Short, simple queries
    - Overhead of compilation outweighs performance gains.
- To many joins
    - If your query involve to many joins do not use JIT

02 PostgreSQL Performance Optimization

Just-In-Time (JIT) Compilation

NoteL: Setting JIT parameters too aggressively or too low can have drawbacks.

- Lower values enable more JIT compilation
- Higher values keep it selective.

So, if set too low, unnecessary parts of the query may be compiled, potentially increasing execution time instead of improving performance.

02 PostgreSQL Performance Optimization

Just-In-Time (JIT) Compilation

When not to use JIT

While JIT = off

- Planning Time: 26.136 ms
- Execution Time: 7.888 ms

While JIT = on

- Planning Time: 27.201 ms
- JIT:
  - Options: Inlining true, Optimization true, Expressions true, Deforming true
  - Timing: Generation 96.421 ms, Inlining 14.554 ms, Optimization 8423.264 ms, Emission 20500.050 ms, Total 29034.289 ms
- Execution Time: 29091.121 ms

Query structure

- Hundreds of nested joins

02 PostgreSQL Performance Optimization

Caution: JIT is not effective for queries with multiple complex joins, so it is best to avoid using it in this scenario.

Source: Percona

02 PostgreSQL Performance Optimization

EXPLAIN and EXPLAIN ANALYZE

- EXPLAIN
  - Shows the execution plan PostgreSQL intends to use for a query without running it.
- EXPLAIN ANALYZE
  - Executes the query and provides the actual execution plan with runtime statistics.
- Key Difference: EXPLAIN estimates the plan without executing, while EXPLAIN ANALYZE runs the query and shows real execution details.

Caution: Avoid running EXPLAIN ANALYZE on modification queries (INSERT, UPDATE, DELETE) unless you are sure, as it will execute the query.

02 PostgreSQL Performance Optimization

EXPLAIN ANALYZE: Scenario 1

```
quote> Nested Loop  (cost=1000.42..25607.38 rows=1 width=129) (actual time=78.521..90.445 rows=0 loops=1)
  ->  Gather  (cost=1000.00..25598.95 rows=1 width=65) (actual time=78.520..90.443 rows=0 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Parallel Seq Scan on <table_1> e  (cost=0.00..24598.85 rows=1 width=65) (actual time=75.351..75.351 rows=0 loops=3)
              Filter: ((<column_1>) = '<date_value>'::date) AND ((<column_2>)::text = '<event_type>'::text)
              Rows Removed by Filter: <number_removed_rows>
  ->  Index Scan using <index_name> on <table_2> a  (cost=0.42..8.43 rows=1 width=41) (never executed)
        Index Cond: ((<column_3>)::text = (<column_4>)::text)
        Filter: ((<column_5>)::text = '<default_value>'::text)
Planning Time: 0.466 ms
Execution Time: 90.580 ms
```

02 PostgreSQL Performance Optimization

EXPLAIN ANALYZE: Scenario 2

```
quote> Nested Loop  (cost=0.14..18.32 rows=1 width=642) (actual time=0.003..0.004 rows=0 loops=1)
  Join Filter: ((<column_1>)::text = (<column_2>)::text)
  ->  Index Scan using <index_name> on <table_1> e  (cost=0.14..8.15 rows=1 width=546) (actual time=0.003..0.003 rows=0 loops=1)
        Index Cond: (<column_3> = '<date_value>'::date)
  ->  Seq Scan on <table_2> ps  (cost=0.00..10.16 rows=1 width=318) (never executed)
        Filter: ((<column_4>)::text = '<default_value>'::text)
Planning Time: 0.424 ms
Execution Time: 0.032 ms
```

02 PostgreSQL Performance Optimization

EXPLAIN ANALYZE: https://explain.dalibo.com/

03 Q&A