PostgreSQL Cost Calculations Seem to Depend on Some Arbitrary Magical Numbers

Josef Machytka
4 min read · Just now

In November 2023, I began a small research project at NetApp Open Source Services with the goal of gathering practical knowledge about JSONB data and the use of indexes on JSONB columns for various use cases. The project was motivated by our clients' problems and questions, as available online resources were insufficient, to say the least.

What started as a modest exploration gradually expanded into a much larger project, the results of which I presented at two PostgreSQL conferences in June 2024: Prague PostgreSQL Developer Day 2024 and Swiss PGDay 2024 and also at the Berlin PostgreSQL meetup in October 2024.

I plan to publish a series of articles delving deeper into different the findings of this project. These articles will explore insights that were only briefly touched upon in my talks. Among the many interesting takeaways from the project, one a bit surprising discovery stands out, which I will detail in this article.

PostgreSQL and Cost-Based Query Optimization

PostgreSQL uses a cost-based query optimizer, and the accuracy of its cost calculations depends on several configuration parameters. PostgreSQL professionals are of course very well familiar with key settings such as *shared_buffers, effective_cache_size, random_page_cost, seq_page_cost*, and *effective_io_concurrency*. These parameters can be critical: if not optimized for the specific characteristics of a database instance, the query planner may select suboptimal execution plans. For example, incorrect settings might cause the planner to prefer a parallel sequential scan of an entire table over an index scan, due to inaccurate cost estimates making the former seem cheaper.

In my project, I needed to gain a deeper understanding of cost calculations, so I dived into PostgreSQL's source code. The most significant file for this purpose appears to be the file src/backend/optimizer/path/costsize.c, which contains numerous comments and detailed cost calculations. While examining the code, it became evident that the cost calculations for processing tables are heavily influenced by the *cpu_tuple_cost* parameter, while costs for processing indexes similarly depend on *cpu_index_tuple_cost*.

The Enigmatic cpu_tuple_cost and cpu_index_tuple_cost

As critical as these parameters seem to be, the documentation about them is surprisingly sparse. In Section 19.7 of the PostgreSQL documentation, the descriptions are very brief:

> *cpu_tuple_cost (floating point)*
> *Sets the planner's estimate of the cost of processing each row during a query. The*

> *default is 0.01.*
>
> *cpu_index_tuple_cost (floating point)*
> *Sets the planner's estimate of the cost of processing each index entry during an index scan. The default is 0.005.*

Both parameters are described in the code as being measured on an "arbitrary scale," leaving their precise meaning and optimal configuration open to interpretation.

It seems clear that these parameters depend heavily on the hardware and the specific characteristics of tables and indexes. Faster CPUs and more efficient memory access can reduce the time needed to process data, suggesting

settings might need to be fine-tuned even at the table/index level.

Why These Values Matter

The planner uses *cpu_tuple_cost* to weigh the relative costs of different operations. Intuitively, processing many rows without indexing incurs a higher CPU cost. Therefore, a higher *cpu_tuple_cost* and lower *cpu_index_tuple_cost* make sequential scans more expensive for the planner, compared to index scans.

However, the generic value assigned to *cpu_index_tuple_cost* might not suit all types of indexes equally. The default value likely works well for standard B-tree indexes but may be suboptimal for other index types like GIN, GiST, or BRIN, which have differing performance characteristics. This could mean that PostgreSQL's planner might not always make the best decisions for queries involving these specialized indexes. In some cases, it might even be necessary to tune these values specifically for each index based on its type.

The *cpu_tuple_cost* parameter also most likely affects the planner's choice of join algorithms (e.g., hash join, merge join, nested loop join). For instance, hash joins and merge joins generally involve fewer tuple comparisons, which may be reflected in cost calculations influenced by *cpu_tuple_cost*. Adjusting this parameter could potentially alter the planner's preference for one join strategy over another, especially in workloads with large datasets or complex join conditions.

The Need for Further Tests and Research

Despite the critical role these parameters play, there seems to be surprisingly little publicly available community knowledge about their optimal values. During my presentations at PostgreSQL conferences, I raised questions about *cpu_tuple_cost* and *cpu_index_tuple_cost* but received limited insights. This gap in understanding motivates me to continue researching their behavior and impact.

I would like to ask readers to kindly share any insights or ideas for further research on these parameters. Are there methodologies for measuring their optimal values? Are there not published use cases where tuning these settings led to significant performance improvements? I will of course share updates from my further tests and research, and I would greatly appreciate input from others who have some knowledge about this area.

Final Thoughts

Understanding the impact of *cpu_tuple_cost* and *cpu_index_tuple_cost* might not be critical for many routine use cases with modest amounts of data. However, in environments with diverse workloads and hardware configurations, fine-tuning these parameters might help to improve the behavior of PostgreSQL's query planner. While the documentation and source code provide a starting point, there is ample room for further exploration. Let's try to demystify these "magical numbers" and make PostgreSQL's performance optimization more precise.

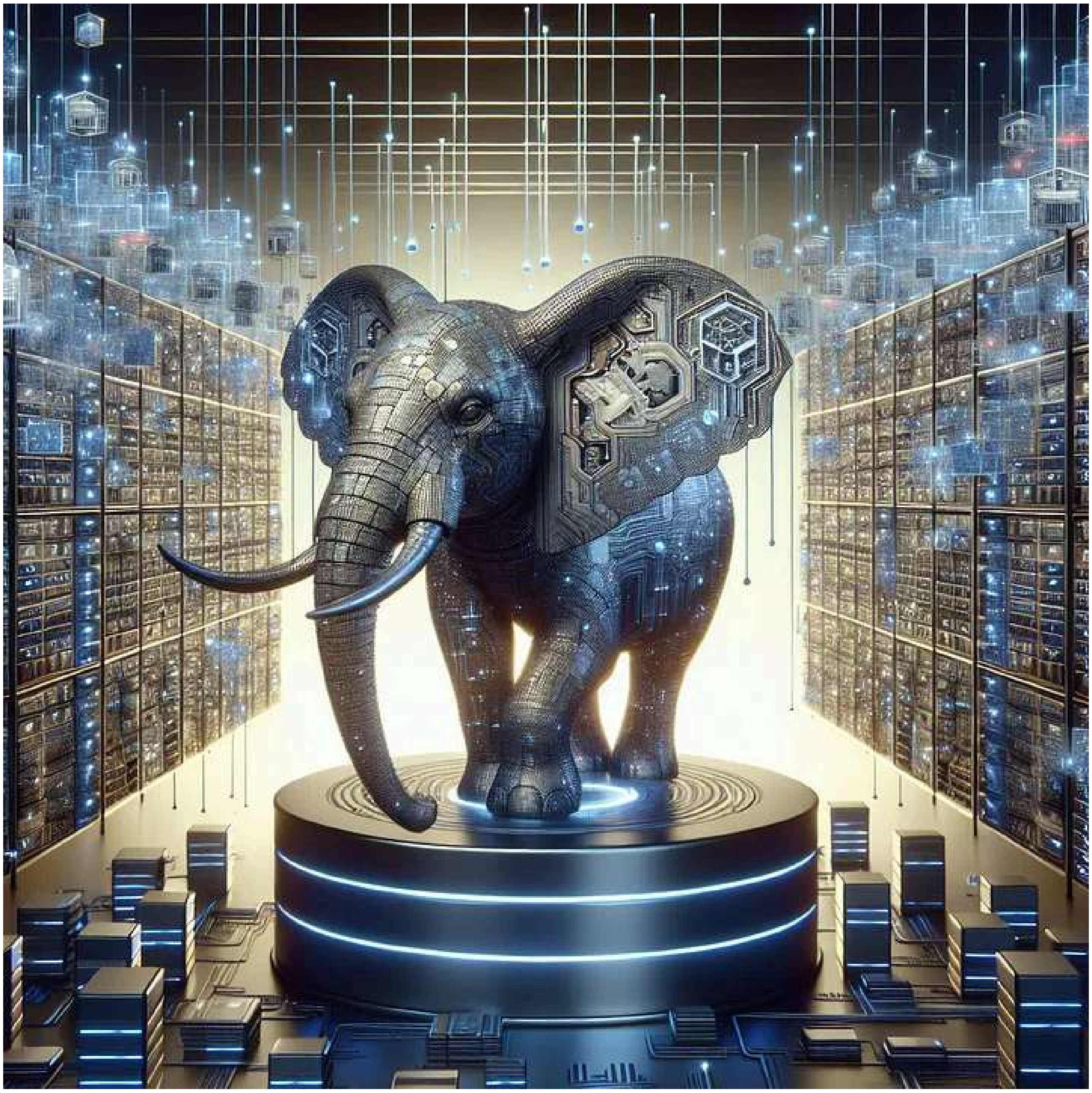


Image created by the author using DeepDreamGenerator

Postgresql Optimization Query Planning Tuning

Written by Josef Machytka

26 Followers · 5 Following

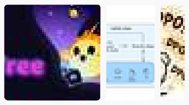
Edit profile

I work as Professional Service Consultant - PostgreSQL specialist in NetApp Deutschland GmbH, Open Source Services division.

Oct 23 20K 482

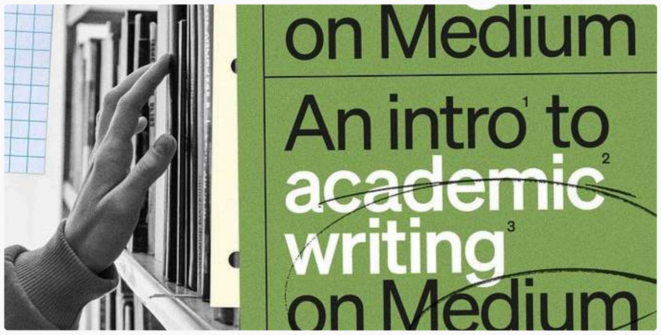
Sep 23 30K 659

Lists



Natural Language Processing

1851 stories · 1473 saves

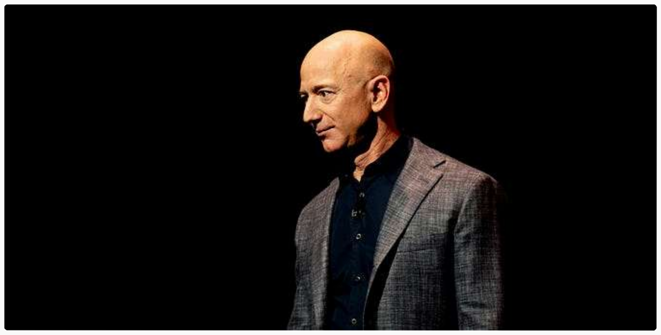


In The Medium Blog by Brittany Jezouit

An introduction to academic writing on Medium

How academics use Medium to share research and ideas easily, connect with non-...

4d ago 6.4K 87



Jessica Stillman

Jeff Bezos Says the 1-Hour Rule Makes Him Smarter. New...

Jeff Bezos's morning routine has long included the one-hour rule. New...

Oct 30 14.8K 355

Harendra

Omar Hussein

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 6.9K 101

Why Do We Always Forget Names Right After Being Introduced?

Read for free here

5d ago 10.1K 251

See more recommendations