Rebuilding a Hash Index Can Take Ages in PostgreSQL

Josef Machytka
3 min read · 4 days ago

PostgreSQL offers multiple types of indexes, with B-Tree indexes being the most commonly used and widely discussed. For specialized use cases, the database also provides other indexing types, such as GIN indexes for full-text search and JSONB data, and GiST indexes for geospatial data. However, PostgreSQL also implements hash indexes, which can be useful in some specific scenarios.

Users sometimes experiment with hash indexes as potential replacements for B-Tree indexes. In certain use cases, hash indexes may indeed perform better. However, during my internal NetApp research project that explored various index types — primarily on JSONB data — I found no significant advantages of hash indexes over B-Tree indexes. In every tested scenario, hash indexes were consistently slightly larger and slower than their B-Tree counterparts.

A Hidden Problem: Creation/Reindexation Time

While conducting tests, I encountered one major issue with hash indexes: their creation/reindexation can take an extraordinarily long time if the underlying data is unsuitable. PostgreSQL documentation states that "*hash indexes are most suitable for unique, nearly unique data, or data with a low number of rows per hash bucket*." And this limitation is really crucial to understand before using hash indexes.

I tested hash indexes on both integer and short string data, experimenting with columns containing either entirely unique or repeated values. My findings showed that hash index creation and rebuilding are relatively quick when working with fully unique data, regardless of whether the data type is integer or short string. Also integer data with high repetition of values is processed relatively efficiently. However, short strings with high repetition rates can lead to significant performance degradation.

Test Results: A Performance Breakdown

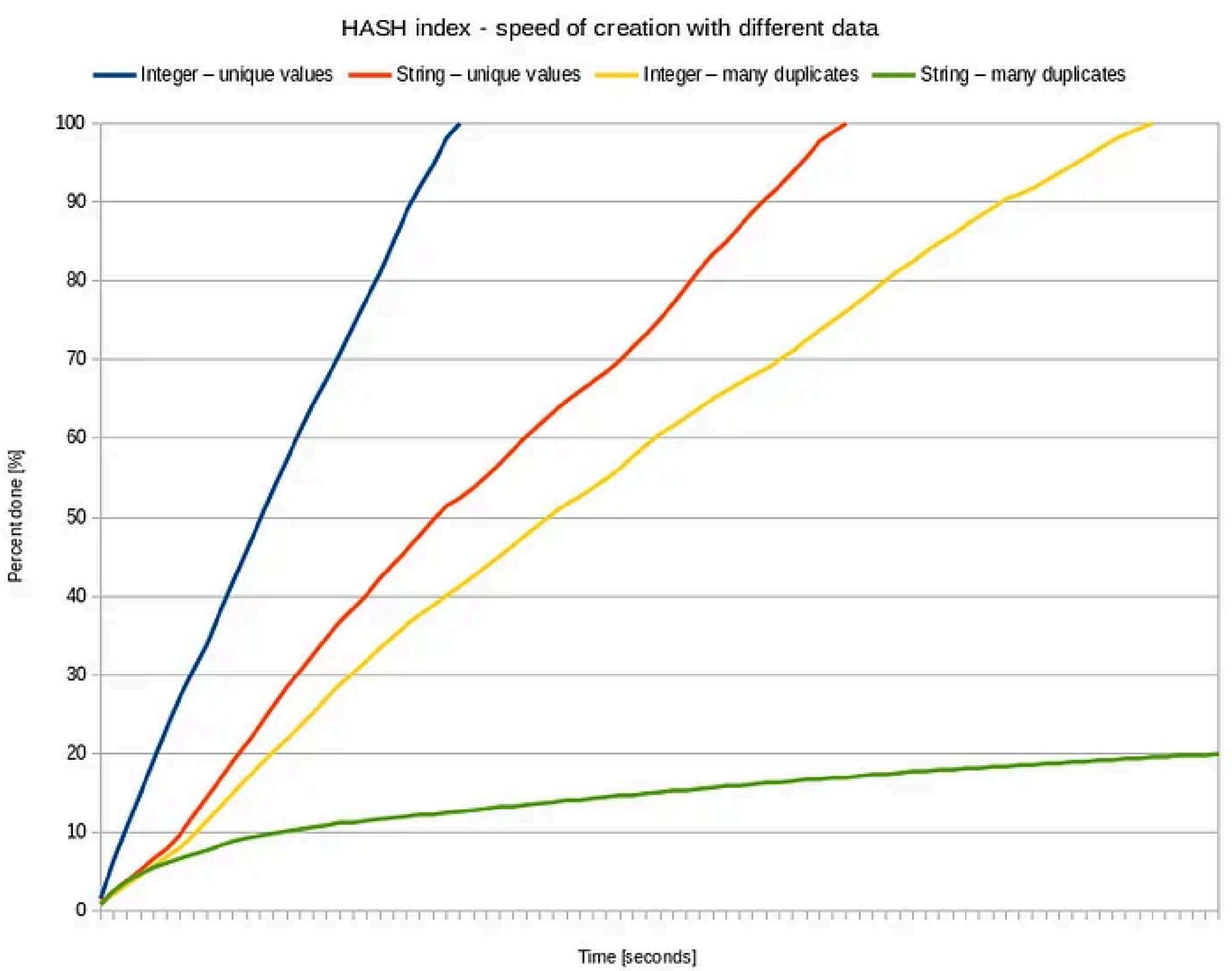
To illustrate this behavior, I generated a dataset of 20 million records with various column types and measured the speed of index creation and reindexing. I tested each index separately. Below are the results, showing how long hash index creation/recreation took on different types of data:

- Unique integer values — approximately 30 seconds.
- Highly duplicate integer values — around 1.5 minutes.
- Unique short string values — about 1 minute.
- Highly duplicate short string values — *almost 55 minutes*.

For each test, I monitored the progress of index creation/recreation using simple query shown below on the *pg_stat_progress_create_index* view, logging results every second into a temporary table.

```
SELECT
    clock_timestamp() AS ts,
    index_relid::regclass::text AS index_name,
    command,
    round(blocks_done/blocks_total::numeric*100,2) AS perc_done
FROM pg_stat_progress_create_index
```

After completing all tests, I compiled the data into a graph for comparison and here is what I got:



The progress of creating/recreating a hash index on highly duplicate short string values was extremely slow. The higher the occurrence of repeated values, the longer the creation/recreation process took. When a column contained only three to five distinct values across 20 million records, the indexing process took hours, and I eventually canceled it. By contrast, creating B-Tree indexes on the same data was relatively quick, taking only 1 to 3 minutes in all cases.

Choosing Hash Indexes Wisely

To avoid performance bottlenecks, it is essential to analyze your data carefully before choosing a hash index. The *pg_stats* view can provide valuable insights. Specifically:

- The *n_distinct* column indicates the level of data uniqueness. A value of -*1* signifies fully unique data, making it ideal for hash indexes.
- For other values, further examination of the *most_common_vals* and *most_common_freqs* columns is necessary. If frequencies in *most_common_freqs* are 0.1 or higher, a hash index is likely a poor choice.

In some cases, using partial index can help mitigate issues by excluding problematic or NULL values. However, partial indexes may not fit all use cases.

Conclusion

While hash indexes can be beneficial in specific scenarios, their limitations — especially regarding reindexation on repetitive text data — make them less versatile than other index types in PostgreSQL. Always analyze your data and PostgreSQL statistics carefully to determine the best indexing strategy for your use case.

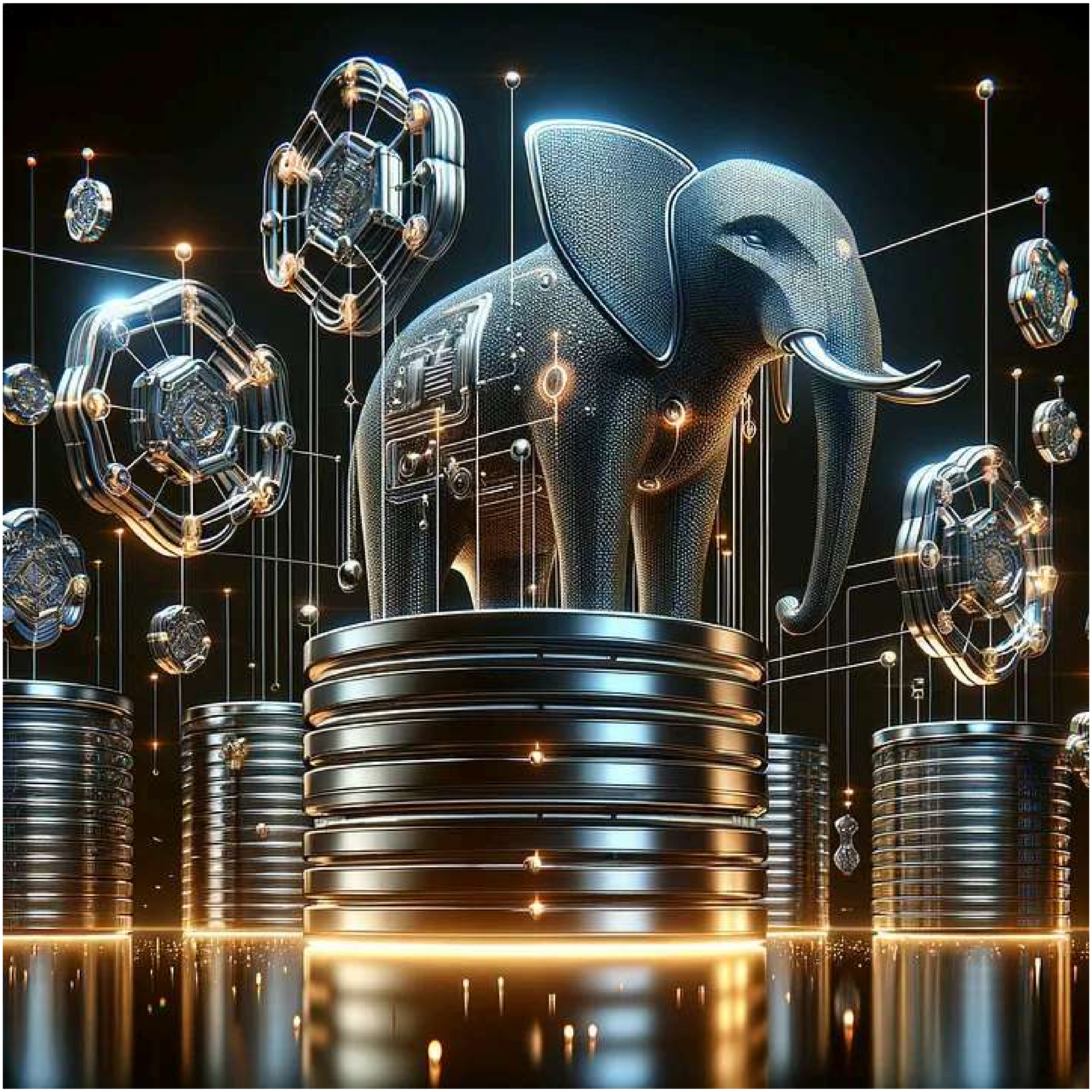


Image created by the author using DeepDreamGenerator

Postgresql Indexing Hash Index

Written by Josef Machytka

37 Followers · 7 Following

Edit profile

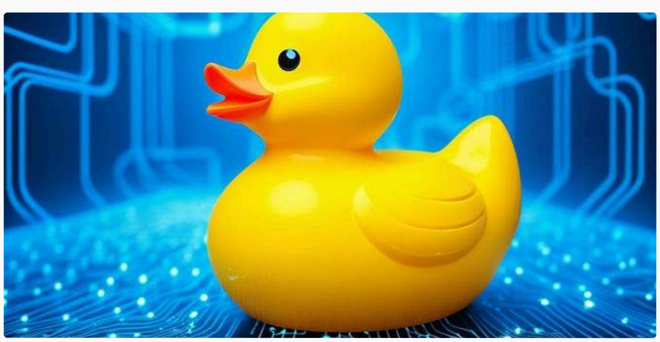
I work as Professional Service Consultant - PostgreSQL specialist in NetApp Deutschland GmbH, Open Source Services division.

No responses yet

What are your thoughts?

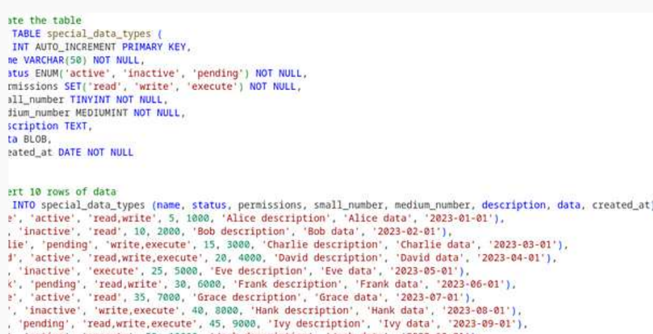
Respond

More from Josef Machytka



Josef Machytka

Extending DuckDB ETL Capabilities with Python



Josef Machytka

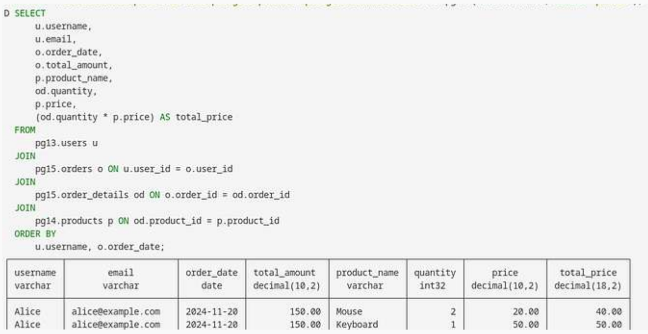
DuckDB as a Rudimentary Data Migration Tool

DuckDB has recently become my go-to solution for small ETL tasks. It is an…

Nov 24 20

After exploring how to use DuckDB as an intelligent ETL tool for PostgreSQL, and how…

Nov 30 1



```
D SELECT
    u.username,
    u.email,
    o.order_date,
    o.total_amount,
    p.product_name,
    od.quantity,
    p.price,
    (od.quantity * p.price) AS total_price
  FROM
    pg13.users u
  JOIN
    pg15.orders o ON u.user_id = o.user_id
  JOIN
    pg15.order_details od ON o.order_id = od.order_id
  JOIN
    pg14.products p ON od.product_id = p.product_id
  ORDER BY
    u.username, o.order_date;
```

username varchar	email varchar	order_date date	total_amount decimal(10,2)	product_name varchar	quantity int32	price decimal(10,2)	total_price decimal(18,2)
Alice	alice@example.com	2024-11-20	150.00	Mouse	2	20.00	40.00
Alice	alice@example.com	2024-11-20	150.00	Keyboard	1	50.00	50.00

Josef Machytka

Easy Cross-Database Selects with DuckDB

DuckDB was created with simplicity and ease of use in mind. In my previous article, I…

Nov 26 3

Josef Machytka

How DuckDB handles data not fitting into memory?

In my previous article about DuckDB I described how to use this database as an…

Nov 13 7

See all from Josef Machytka

Recommended from Medium

Anass Guendef

Understanding Postgres Sql Plan

Introduction

Nov 14 2

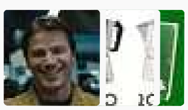
Manojkumar Vadivel

14 VS Code Extensions Every Data Engineer Should Swear By for...

As a data engineer, your toolbox is everything. The right set of tools can save you time,...

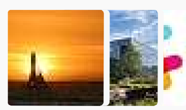
Nov 24 636 4

Lists



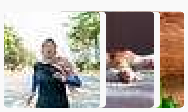
Staff picks

788 stories · 1502 saves



Stories to Help You Level-Up at Work

19 stories · 892 saves



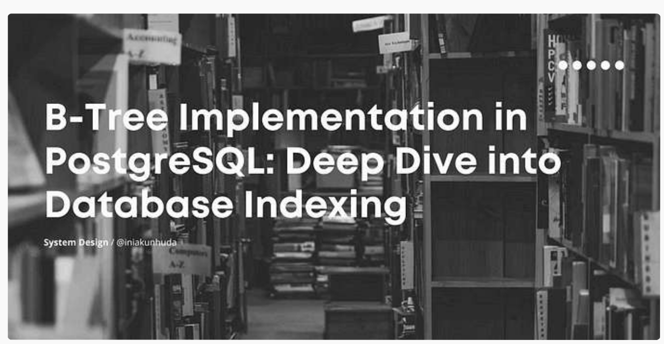
Self-Improvement 101

20 stories · 3135 saves

Productivity 101

20 stories · 2650 saves

In Towards Data Science by Eyal Trabelsi



Miftahul Huda

Query Optimization for Mere Humans in PostgreSQL

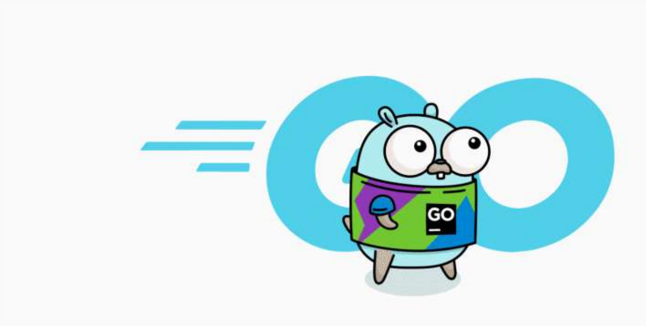
Understanding a PostgreSQL execution plan with practical examples

Dec 3 · 184

B-Tree Implementation in PostgreSQL: Deep Dive into...

In this article, we explore the B-Tree indexing implementation in PostgreSQL and its crucia...

Oct 28 · 3 · 1

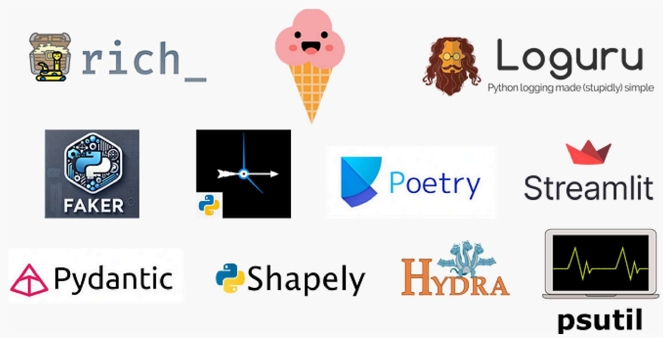


In Stackademic by Abhinav

Go: A Silent Killer in the Programming World

In a software development landscape dominated by trendy frameworks and...

Dec 6 · 183 · 7



In Level Up Coding by Md Arman Hossen

11 Python Libraries That Will 10x Your Development Speed in 2024:...

11 Game-Changing Python Libraries You've Been Missing in 2024

Dec 3 · 445 · 6

See more recommendations