Member-only story

Optimizing PostgreSQL Queries: From 300 Seconds to 2 Seconds for Billions of Records

Code Geass · Following

5 min read · Aug 26, 2024

https://medium.com/codex/intro-to-postgresql-c8da31335c34

I was working on a huge volumes where I had this entire experience of optimizing the PostgreSQL, to make it support huge data. This article is designed based on our ideas, implementations that we had during the course of this process of reduction of the execution time that finally came down to 2 seconds from a mammothing 150 seconds for the worst case scenerios.

Disclaimer: GPT-4 assisted me in writing this article, combining its insights with my personal experience.

Table of Contents

Understanding the Problem

When working with massive datasets in PostgreSQL, poor table design, lack of indexing, and suboptimal query structures can lead to significant performance issues. In this scenario, we are optimizing a query that must process billions of records from a single table.

Without optimization, this query was initially taking more than 150 seconds to execute, which is far from acceptable in a production environment.

Generating Billions of Records

To simulate this scenario, let's first generate a table with billions of records using PostgreSQL's powerful data generation capabilities. The following table structure will be used:

```
CREATE TABLE transactions (
  transaction_id serial PRIMARY KEY,
  user_id int NOT NULL,
```

```
  transaction_date date NOT NULL,
  amount decimal(10,2) NOT NULL
);
```

Data Generation

The following stored procedure will generate 1 billion records:

```
DO $$
BEGIN
    FOR i IN 1..1000000 LOOP
        INSERT INTO transactions (user_id, transaction_date, amount)
        SELECT
            trunc(random() * 10000 + 1),
            '2020-01-01'::date + trunc(random() * 1000)::int,
            trunc(random() * 1000 + 1)::decimal(10,2)
        FROM generate_series(1, 1000);
    END LOOP;
END $$;
```

This script inserts 1 billion transaction records, simulating a real-world, large-scale database.

Original SQL Query

Our initial query attempts to calculate the total transaction amount for each user:

```
SELECT user_id, SUM(amount) AS total_amount
FROM transactions
GROUP BY user_id;
```

Without any indexes, this query takes over 150 seconds to complete. The *Explain* output shows a full table scan, which is highly inefficient.

First Optimization: Indexes

Adding Indexes

To improve performance, we start by adding an index to the *user_id* column, which is used in the *Group By* clause:

```
CREATE INDEX idx_user_id ON transactions (user_id);
```

Revised Query Execution

After adding the index, the query's execution time dropped to around 80 seconds. While this is an improvement, it's still not fast enough. The *Explain* output shows that the index is being used, but the process remains I/O-bound due to the sheer volume of data.

Second Optimization: Partitioning

Implementing Table Partitioning

To further optimize the query, we introduce partitioning based on *transcation_date*. Partitioning allows PostgreSQL to divide the table into smaller, more manageable pieces, thereby reducing the data each query needs to scan.

```
CREATE TABLE transactions_partitioned (
  transaction_id serial PRIMARY KEY,
  user_id int NOT NULL,
  transaction_date date NOT NULL,
  amount decimal(10,2) NOT NULL
) PARTITION BY RANGE (transaction_date)
;
CREATE TABLE transactions_2020 PARTITION OF transactions_partitioned
FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');

CREATE TABLE transactions_2021 PARTITION OF transactions_partitioned
FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');
```

Query Execution on Partitioned Tables

After partitioning, the query execution time decreased significantly to around 20 seconds. The *Explain* plan now shows that only the relevant partitions are being

scanned, which reduces the amount of data processed.

Third Optimization: Materialized Views

Creating a Materialized View

To push the optimization further, we introduce a materialized view to pre-calculate the sums for each user, reducing the need to repeatedly aggregate the entire dataset

```
CREATE MATERIALIZED VIEW user_transaction_sums AS
SELECT user_id, SUM(amount) AS total_amount
FROM transactions_partitioned
GROUP BY user_id;
```

Final Query Using Materialized View

The final query is now a simple *SELECT* from the materialized view:

```
SELECT * FROM user_transaction_sums;
```

This query executes in less than 2 seconds, achieving the performance target.

Handling Edge Cases

While the above optimizations significantly improved performance, there are several edge cases that need to be considered to ensure consistent results and avoid unexpected performance degradation.

1. Handling Skewed Data Distribution

In cases where the data is heavily skewed, with certain *user_ids* having a disproportionate number of transactions, partitioning and indexing strategies might not be sufficient. To address this, consider:

- **Hash Partitioning:** Use hash partitioning on *user_id* to evenly distribute data across partitions.

- **Partial Indexes:** Create partial indexes on frequently accessed users, e.g., users with the most transactions.

```
CREATE INDEX idx_high_activity_users ON transactions (user_id)
WHERE user_id IN (SELECT user_id FROM transactions GROUP BY user_id HAVING COUN
```

2. High Cardinality Columns

If the dataset includes high-cardinality columns (e.g., *transaction_id*), indexing these may lead to large index sizes that negatively impact performance. In such cases:

- **Composite Indexes:** Use composite indexes combining high-cardinality columns with lower-cardinality columns to improve selectivity.

```
CREATE INDEX idx_composite_user_date ON transactions (user_id, transaction_date
```

- **Clustered Indexing:** Consider clustering the table by *user_id*, allowing sequential reads, which are faster for grouped queries.

3. Frequent Data Updates

For databases with frequent updates or inserts, maintaining materialized views and indexes can become costly. In such scenarios:

- **Incremental Updates:** Use PostgreSQL's *Refresh Materialized view concurrently* to update materialized views without locking the table.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY user_transaction_sums;
```

- **Avoid Over-Indexing:** Balance the number of indexes with the write workload. Too many indexes can slow down inserts and updates.

4. Join Performance Issues

When dealing with joins between large tables, PostgreSQL's query planner may not always choose the most efficient plan, especially with complex queries. Solutions include:

- **Join Hints:** While PostgreSQL doesn't natively support optimizer hints like some other databases, restructuring queries to encourage nested loop joins or hash joins based on your dataset can improve performance.
- **Query Rewriting:** Rewrite queries to simplify joins, possibly using Common Table Expressions (CTEs) to break down complex queries into more manageable parts.

```
WITH user_sums AS (
  SELECT user_id, SUM(amount) AS total_amount
  FROM transactions
  GROUP BY user_id
)
SELECT u.user_id, u.name, us.total_amount
FROM users u
JOIN user_sums us ON u.user_id = us.user_id;
```

5. Query Timeouts

For particularly long-running queries, it's essential to handle potential timeouts gracefully:

- **Query Cancellation:** Implement logic to catch and handle query cancellations, especially in client applications.

```
BEGIN;
SET statement_timeout = '30s';
SELECT user_id, SUM(amount) FROM transactions GROUP BY user_id;
COMMIT;
```

- **Progressive Aggregation:** Break down large aggregations into smaller batches or stages, potentially using temporary tables to store intermediate results.

Final Results and Best Practices

The combination of indexing, partitioning, materialized views, and handling edge cases transformed a 150-second query into a sub-2-second query. Here are the key takeaways:

1. **Indexes**: Start with proper indexing on columns used in *Where*, *Join* , and *Group By* clauses.
2. **Partitioning**: Use table partitioning to break down large tables, reducing the data scanned by queries.
3. **Materialized Views**: Pre-compute complex aggregates to save query execution time.
4. **Edge Case Handling**: Be aware of and address edge cases like skewed data, high cardinality, frequent updates, and complex joins.

Next Steps

For even larger datasets, consider:

- Using *pg_partman* for automated partition management.
- Exploring parallel query execution and tuning PostgreSQL's settings for optimal performance.
- Leveraging caching layers like Redis for frequently accessed data.

By applying these strategies, PostgreSQL can handle even the most demanding datasets efficiently.

Postgresql | Database | Software Development | Software Engineering | Distributed Systems

Following

Written by Code Geass

210 Followers

Putting Software/ Tech in an Easy way

More from Code Geass

Code Geass

Tackling the Toughest: Top 10 Most Complicated Java Interview Questions

If you are preparing to clear out for SDE role in Java in some top companies, this is a good place to build up your knowledge, I have...

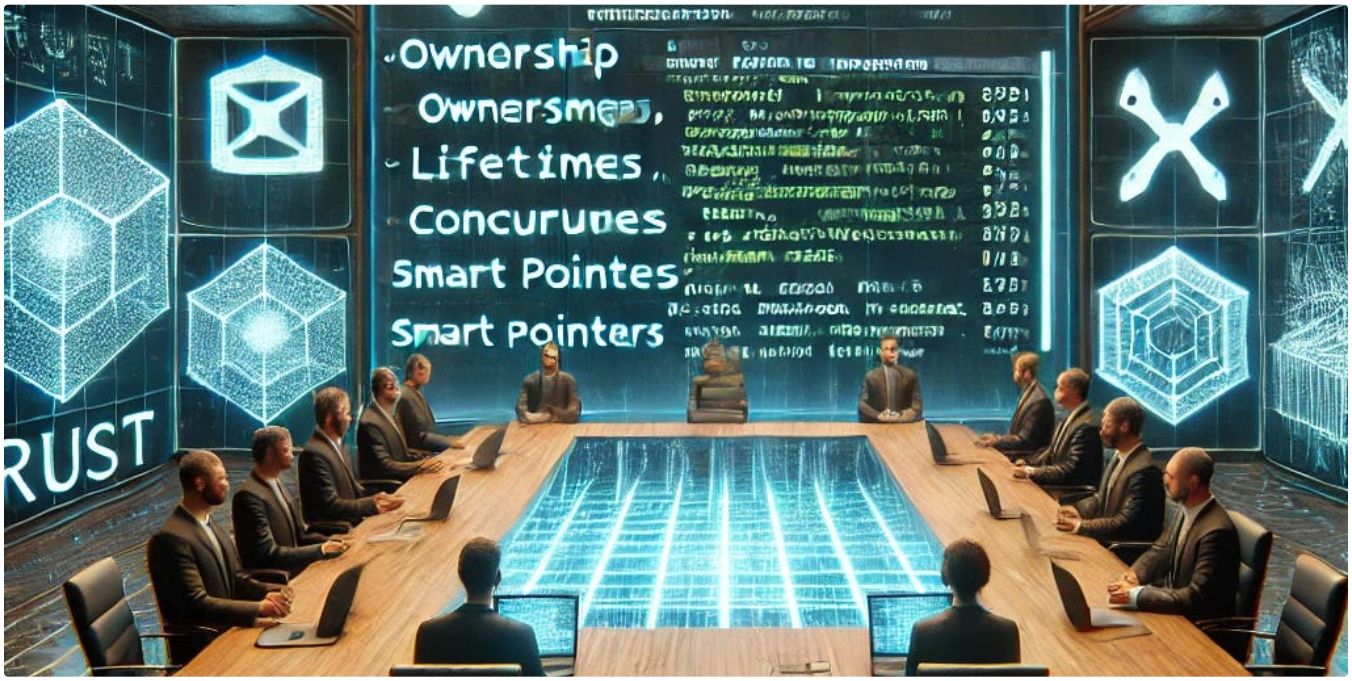
Aug 23 22

Code Geass

Top 20 Most Crucial Interview Questions on Swift

Disclaimer: gpt4 helped me writing this article and putting this article together.

Aug 26 42 2



Code Geass

Top 30 Most Crucial Coding Interview Questions on Rust

Disclaimer: gpt4 helped me writing this article

Aug 19 170 2

Code Geass

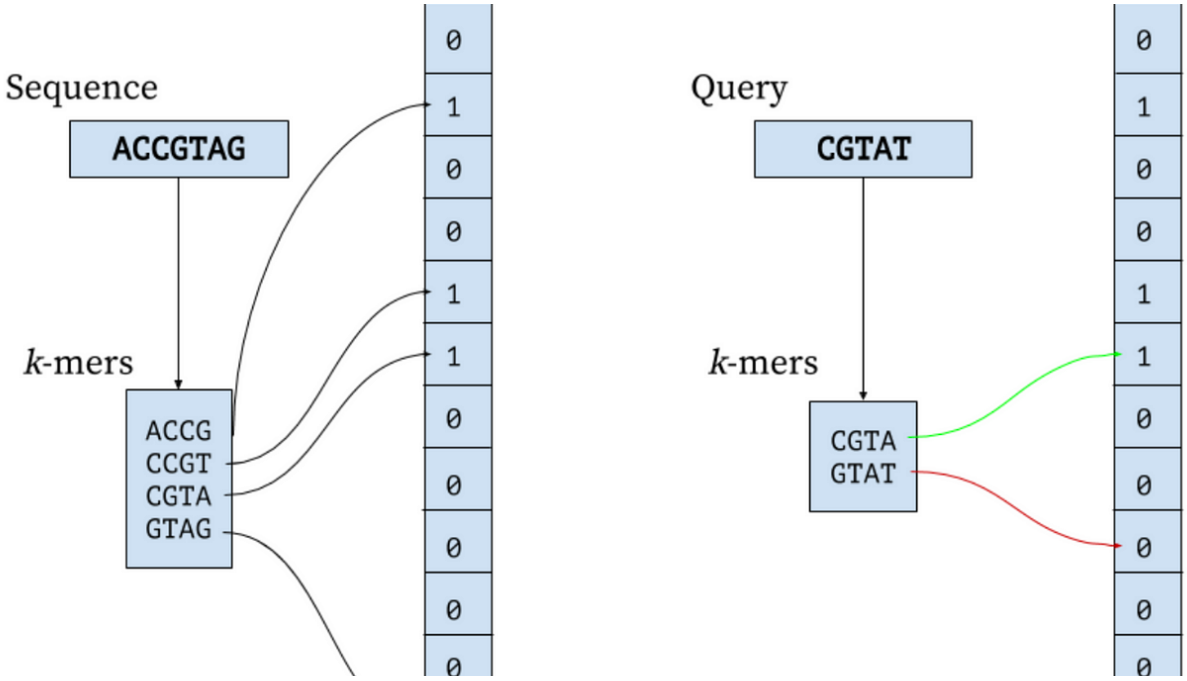
Top Golang Interview Questions Related to GoRoutines- Part 1

1) What is a goroutine in Go, and how does it differ from a traditional thread?

Sep 7, 2023 222 2

See all from Code Geass

Recommended from Medium



Aditi Mishra

How to Efficiently Check If a Username Exists Among Billions of Users

How to Efficiently Check If a Username Exists Among Billions of Users

Manjula Piyumal

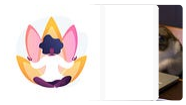
Mastering Kafka: Advanced Concepts Every Senior Software Engineer Should Know

Photo by Pietro Jeng on Unsplash

Lists

General Coding Knowledge

20 stories · 1598 saves



Stories to Help You Grow as a Software Developer

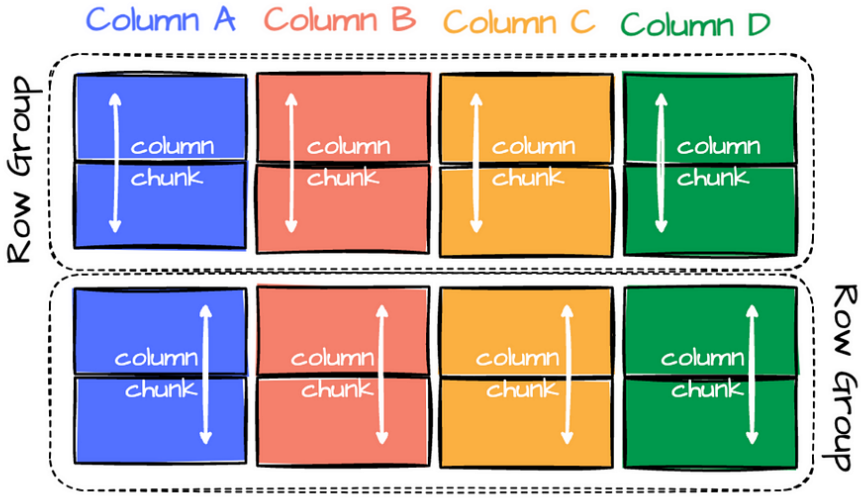
19 stories · 1378 saves

Leadership

59 stories · 442 saves

Coding & Development

11 stories · 820 saves



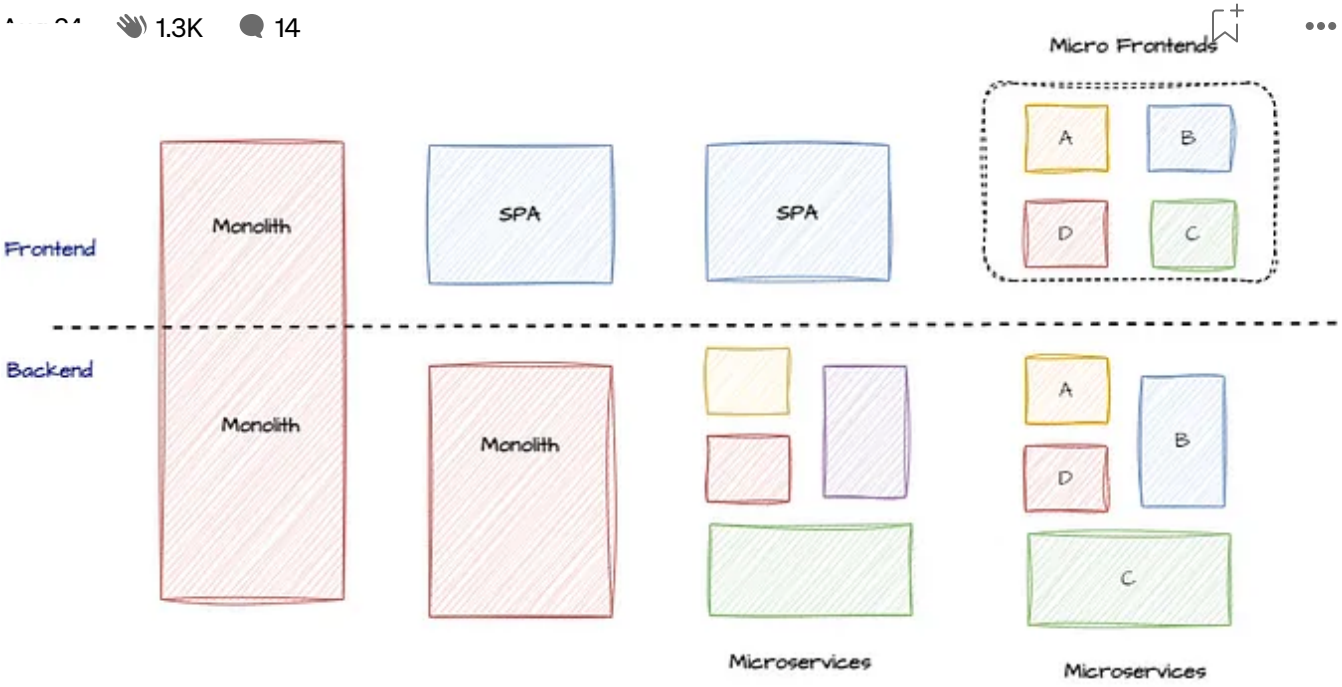
I spent 8 hours learning Parquet. Here's what I discovered

Vu Trinh in Data Engineer Things

I spent 8 hours learning Parquet. Here's what I discovered

I finally sat down and learned about it.

1.3K 14

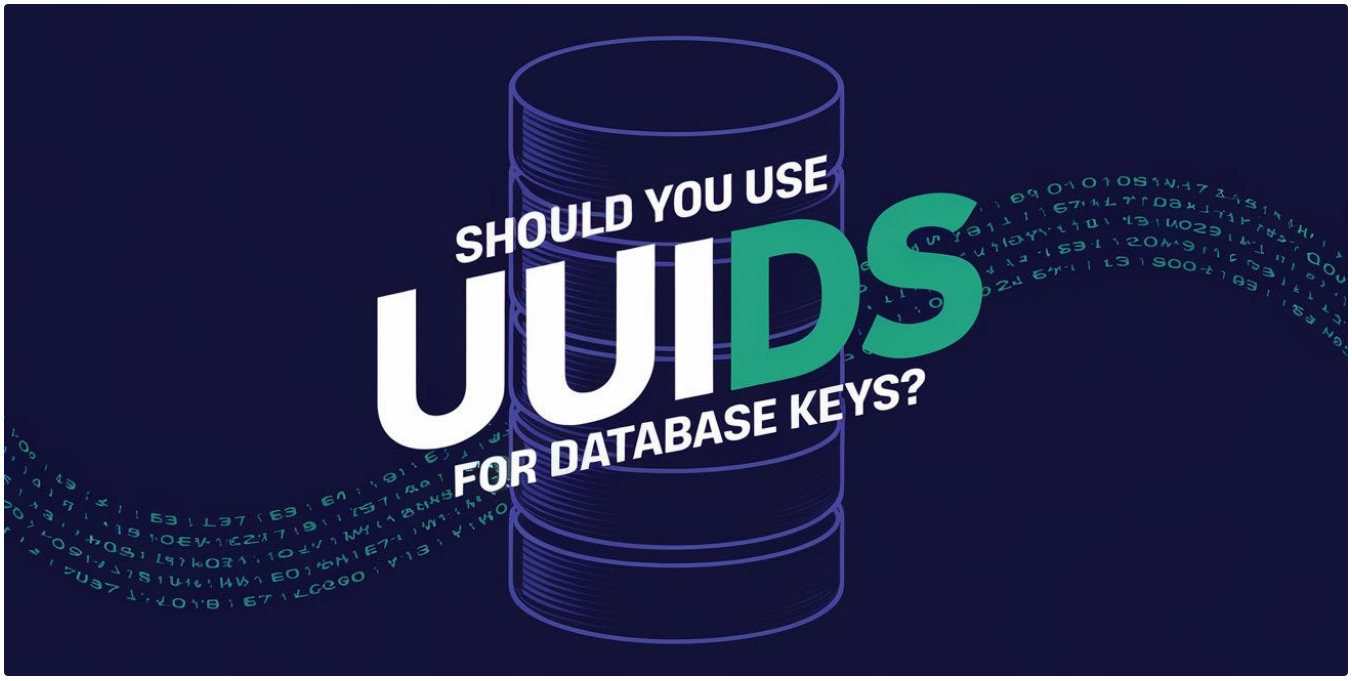


Rohit S in Level Up Coding

Micro Frontend Architecture

As web applications grow in complexity, teams seek scalable, modular approaches to frontend development. One such approach, Micro Frontend...

Sep 16 262 5



Rabinarayan Patra

Should You Use UUIDs for Database Keys?

UUIDs offer uniqueness in databases but come with performance costs. Learn when to use them, their drawbacks, and alternatives loved by...

Sep 14 293 5

Rahul Sharma in AWS in Plain English

I have Asked This SSH Question in Every AWS Interview—And Here's the Catch

When I interview people, I always ask questions about problems that people face in the real world.

Sep 16 515 25