20 JANUARY 2026 / POSTGRESQL, PERFORMANCE / 19 188

Unconventional PostgreSQL Optimizations

Creative ideas for speeding up queries in PostgreSQL

. . .

When it comes to database optimization, developers often reach for the same old tools: rewrite the query slightly differently, slap an index on a column, denormalize, analyze, vacuum, cluster, repeat. Conventional techniques are effective, but sometimes being creative can really pay off!

In this article, I present unconventional optimization techniques in PostgreSQL.

image by abstrakt design

Table of Contents

. . .

Eliminate Full Table Scans Based on Check Constraints

Imagine you have this table of users:

```
db=# CREATE TABLE users (
    id INT PRIMARY KEY,
    username TEXT NOT NULL,
    plan TEXT NOT NULL,
    CONSTRAINT plan_check CHECK (plan IN ('free', 'pro'))
);
CREATE TABLE
```

For each user you keep their name and which payment plan they're on. There are only two plans, "free" and "pro", so you add a check constraint.

Generate some data and analyze the table:

```
db=# INSERT INTO users
SELECT n, uuidv4(), (ARRAY['free', 'pro'])[ceil(random()*2)]
FROM generate_series(1, 100_000) AS t(n);
INSERT 0 100000

db=# ANALYZE users;
ANALYZE
```

You now have 100K users in the system.

Honest Mistakes

Now you want to let your analysts access this table in their reporting tool of choice. You give one of the analysts permission, and this is the first query they write:

```
db=# SELECT * FROM users WHERE plan = 'Pro';
 id │ username │ plan
────┼──────────┼──────
(0 rows)
```

The query returned no results, and the analyst is baffled. How come there are no users on the "Pro" plan?

The name of the plan is "pro" and not "Pro" (with a capital "P") as the analyst wrote it. This is an honest mistake really, anyone can make such a mistake! But what is the *cost* of this mistake?

Examine the execution plan of a query for a non-existing value:

```
db=# EXPLAIN ANALYZE SELECT * FROM users WHERE plan = 'Pro';
                QUERY PLAN
──────────────────────────────────────────────────────────────────
 Seq Scan on users  (cost=0.00..2185.00 rows=1 width=45)
                    (actual time=7.406..7.407 rows=0.00 loops=1)
   Filter: (plan = 'Pro'::text)
   Rows Removed by Filter: 100000
   Buffers: shared hit=935
 Planning:
   Buffers: shared hit=29 read=2
 Planning Time: 4.564 ms
 Execution Time: 7.436 ms
```

PostgreSQL scanned the entire table! However, there's a check constraint on the field - no row can ever have the value "Pro", the database makes sure of that! So if this condition always evaluates to false, why is PostgreSQL scanning the table?

Using `constraint_exclusion`

PostgreSQL is smart enough to skip a table scan when the query contains a condition that always evaluates to false, but not by default! To instruct PostgreSQL to look at constraints when generating a plan, you need to set the parameter constraint_exclusion:

```
db=# SET constraint_exclusion to 'on';
SET

db=# EXPLAIN ANALYZE SELECT * FROM users WHERE plan = 'Pro';
                                      QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────────
 Result  (cost=0.00..0.00 rows=0 width=0) (actual time=0.000..0.001 rows=0.00 loops=
   One-Time Filter: false
 Planning:
   Buffers: shared hit=5 read=4
 Planning Time: 5.760 ms
 Execution Time: 0.008 ms
(6 rows)
```

Nice! After turning `constraint_exclusion` on, PostgreSQL figured out based on the check constraint that the condition won't return any rows, and skipped the scan entirely.

When `constraint_exclusion` Makes Sense

So who are you `constraint_exclusion` and why are you not on by default?

> *Currently, constraint exclusion is enabled by default only for cases that are often used to implement table partitioning via inheritance trees. Turning it on for all tables imposes extra planning overhead that is quite noticeable on simple queries, and most often will yield no benefit for simple queries.*

The parameter `constraint_exclusion` is set to "partition" by default, where it's used to eliminate entire partitions when querying against a partitioned table - this is known as "partition pruning".

The documentation states that for simple queries the cost of evaluating all relevant conditions against all the relevant constraints might outweigh the benefit - you might end up spending more time planning than actually executing the query. It makes sense that queries executed by a system are less likely to query for invalid values or apply conditions that go against constraints. However, this is not the case for ad-hoc queries in reporting tools...

In BI and reporting environments users can issue complicated queries that are often crafted by hand. In this type of environment, it's not unlikely that they'll make mistakes, just like our analyst did before. Setting `constraint_exclusion` to "on" in reporting and data warehouse environments where users can issue ad-hoc queries can potentially save time and resources by eliminating unnecessary full table scans.

. . .

Optimize for Lower Cardinality With Function Based Index

Imagine you have a sales table that looks like this:

```
db=# CREATE TABLE sale (
    id INT PRIMARY KEY,
    sold_at TIMESTAMPTZ NOT NULL,
    charged INT NOT NULL
);
CREATE TABLE
```

You keep track of when the sale was made and how much was charged. Create 10 million sales and analyze the table:

```
db=# INSERT INTO sale (id, sold_at, charged)
SELECT
    n AS id,
    '2025-01-01 UTC'::timestamptz + (interval '5 seconds') * n AS sold_at,
    ceil(random() * 100) AS charged
FROM generate_series(1, 10_000_000) AS t(n);
INSERT 0 10000000

db=# ANALYZE sale;
ANALYZE
```

Slapping a B-Tree on it

Your analysts often produce daily sales reports and their queries can look roughly like this:

```
db=# EXPLAIN (ANALYZE ON, BUFFERS OFF, COSTS OFF)
SELECT date_trunc('day', sold_at AT TIME ZONE 'UTC'), SUM(charged)
FROM sale
WHERE '2025-01-01 UTC' <= sold_at AND sold_at < '2025-02-01 UTC'
GROUP BY 1;
                               QUERY PLAN
-------------------------------------------------------------------------------------
 HashAggregate (actual time=626.074..626.310 rows=32.00 loops=1)
   Group Key: date_trunc('day'::text, sold_at)
   Batches: 1  Memory Usage: 2081kB
   ->  Seq Scan on sale (actual time=6.428..578.135 rows=535679.00 loops=1)
         Filter: (('2025-01-01 02:00:00+02'::timestamp with time zone <= sold_at)
                  AND (sold_at < '2025-02-01 02:00:00+02'::timestamp with time zone
         Rows Removed by Filter: 9464321
 Planning Time: 0.115 ms
 Execution Time: 627.119 ms
```

PostgreSQL scanned the entire table and the query completed in ~627ms. Your analysts are a bit spoiled and ~600ms is too slow for them, so you do what you always do in these cases and *"slap a B-Tree index on it"*:

```
db=# CREATE INDEX sale_sold_at_ix ON sale(sold_at);
CREATE INDEX
```

Execute the query with the index:

```
db=# EXPLAIN (ANALYZE ON, BUFFERS OFF, COSTS OFF)
SELECT date_trunc('day', sold_at AT TIME ZONE 'UTC'), SUM(charged)
FROM sale
WHERE '2025-01-01 UTC' <= sold_at AND sold_at < '2025-02-01 UTC'
GROUP BY 1;
                                    QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────────
HashAggregate (actual time=186.970..187.212 rows=32.00 loops=1)
  Group Key: date_trunc('day'::text, sold_at)
  Batches: 1  Memory Usage: 2081kB
  ->  Index Scan using sale_sold_at_ix on sale (actual time=0.038..137.067 rows=535
        Index Cond: ((sold_at >= '2025-01-01 02:00:00+02'::timestamp with time zone
                    AND (sold_at < '2025-02-01 02:00:00+02'::timestamp with time zo
        Index Searches: 1
Planning Time: 0.261 ms
Execution Time: 187.363 ms
```

Execution time reduced from ~627ms to 187ms and the analysts are happy, but at what cost?

```
db=# \di+ sale_sold_at_ix
List of indexes
─[ RECORD 1 ]─┬──────────────────
Schema        │ public
Name          │ sale_sold_at_ix
Type          │ index
Owner         │ haki
Table         │ sale
Persistence   │ permanent
Access method │ btree
Size          │ 214 MB
Description   │ ¤
```

The index is 214 MB! That's almost half the size of the entire table. So the analysts are happy, but you? Not so much...

Slapping a B-Tree index is very common, but DBAs and developers often ignore the storage cost and the maintenance burden that comes with it. Using simple measures, we can potentially save some space and money.

Rethinking the Problem

Let's step back and re-think what we were trying to optimize. Analysts wanted to produce *daily* reports, but we provided them with an index that can produce results at a millisecond precision. By indexing the date and the time, we gave the analysts a lot more than what they asked for!

What if instead of indexing the entire datetime, we index just the date, without the time?

```
db=# CREATE INDEX sale_sold_at_date_ix ON sale((date_trunc('day', sold_at AT TIME ZO
CREATE INDEX
```

This creates a function-based index on the date part of the sale date. We make sure to set the time zone before we truncate the date to match the one used by the analysts in their reports.

First, check the size of the indexes:

```
db=# \di+ sale_sold_at_*
                  List of indexes
        Name          │ Table │ Access method │  Size
──────────────────────┼───────┼───────────────┼────────
 sale_sold_at_date_ix │ sale  │ btree         │ 66 MB
 sale_sold_at_ix      │ sale  │ btree         │ 214 MB
```

The function-based index is just 66MB, that's more than 3 times smaller than the full index. While a `date` is smaller than a `timestamptz` -- 4 bytes vs. 8 bytes -- this is actually not where the majority of the savings come from. The function-based index has *fewer distinct values*, so PostgreSQL can optimize its size using deduplication.

To check if we can make better use of the smaller index, we start by dropping the full index:

```
db=# DROP INDEX sale_sold_at_ix;
DROP INDEX
```

To allow our query to use the function-based index we make some adjustments to the query (we'll tackle that later!):

```
db=# EXPLAIN (ANALYZE ON, BUFFERS OFF, COSTS OFF)
SELECT date_trunc('day', sold_at AT TIME ZONE 'UTC'), SUM(charged)
FROM sale
WHERE date_trunc('day', sold_at AT TIME ZONE 'UTC')::date BETWEEN '2025-01-01' AND
GROUP BY 1;

                                                                 QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────
 GroupAggregate (actual time=6.499..145.889 rows=31.00 loops=1)
   Group Key: date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text))
   ->  Index Scan using sale_sold_at_date_ix on sale (actual time=0.015..119.832 rov
         Index Cond: ((date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text)):
                  AND (date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text)):
         Index Searches: 1
 Planning Time: 0.151 ms
 Execution Time: 145.913 ms
```

The index was used and the query completed in just 145ms, that's ~20ms faster then using the full index, and x4.5 time faster than the full table scan.

The Discipline Problem

Using a function based index can be fragile. If we make even the slightest adjustment to the expression, the database won't be able to use the index:

```
db=# EXPLAIN (ANALYZE OFF, COSTS OFF)
SELECT (sold_at AT TIME ZONE 'UTC')::date, SUM(charged)
FROM sale
WHERE (sold_at AT TIME ZONE 'UTC')::date BETWEEN '2025-01-01' AND '2025-01-31'
GROUP BY 1;
                                 QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────
 HashAggregate
   Group Key: ((sold_at AT TIME ZONE 'UTC'::text))::date
   ->  Seq Scan on sale
         Filter: ((((sold_at AT TIME ZONE 'UTC'::text))::date >= '2025-01-01'::date
         AND (((sold_at AT TIME ZONE 'UTC'::text))::date <= '2025-01-31'::date))
```

The query is the same as the previous, but we changed the expression from using `date_trunc` to using `::date`, so the database was unable to use the function-based index.

Using the exact same expression requires a certain level of discipline that doesn't realistically exist in any organization. It's borderline naive to expect this to be useful this way - we need to come up with a way to force the use of this exact expression.

The old way of doing this involved a view:

```
db=# CREATE VIEW v_sale AS
SELECT *, date_trunc('day', sold_at AT TIME ZONE 'UTC')::date AS sold_at_date
FROM sale;
CREATE VIEW
```

The view adds a new calculated column called "sold_at_date" that uses the exact same expression we used to define the function-based index. Using the view, we can guarantee that we allow the database to use the index:

```
db=# EXPLAIN (ANALYZE OFF, COSTS OFF)
SELECT sold_at_date, SUM(charged)
FROM v_sale
WHERE sold_at_date BETWEEN '2025-01-01' AND '2025-01-31'
GROUP BY 1;
                                         QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────
 GroupAggregate
   Group Key: (date_trunc('day'::text, (sale.sold_at AT TIME ZONE 'UTC'::text)))::da
   ->  Index Scan using sale_sold_at_date_ix on sale
         Index Cond: (((date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text))
         AND ((date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text)))::date <
```

The index is used and the query is fast! Cool, but...

Views are definitely a viable solution here, but they suffer from the same discipline problem - the analysts can still use the table directly (and they will!). We can revoke access from the table or do some magic tricks with `search_path` to fool them into using the view, but there is an easier way.

Using Virtual Generated Columns

Starting at version 14, PostgreSQL supports generated columns - these are columns that are automatically populated with an expression when we insert the row. Sounds exactly like what we need but there is a caveat - the result of the expression is materialized - this means additional storage, which is what we were trying to save in the first place!

Lucky for us, starting with version 18, PostgreSQL supports *virtual* generated columns. A virtual column looks like a regular column, but it's actually an expression that's being evaluated every time it is accessed. Basically, what we tried to achieve before with a view!

First, add a virtual generated column to the table with the same expression we indexed:

```
db=# ALTER TABLE sale ADD sold_at_date DATE
GENERATED ALWAYS AS (date_trunc('day', sold_at AT TIME ZONE 'UTC'));
ALTER TABLE
```

Next, execute the query using the virtual generated column:

```
db=# EXPLAIN (ANALYZE ON, COSTS OFF, BUFFERS OFF)
SELECT sold_at_date, SUM(charged)
FROM sale
WHERE sold_at_date BETWEEN '2025-01-01' AND '2025-01-31'
GROUP BY 1;
                                          QUERY PLAN
──────────────────────────────────────────────────────────────────────────────────────
 GroupAggregate (actual time=7.047..162.965 rows=31.00 loops=1)
   Group Key: (date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text)))::date
   ->  Index Scan using sale_sold_at_date_ix on sale (actual time=0.015..134.795 ro
         Index Cond: (((date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text))
         AND ((date_trunc('day'::text, (sold_at AT TIME ZONE 'UTC'::text)))::date <
         Index Searches: 1
 Planning Time: 0.128 ms
 Execution Time: 162.989 ms
```

Using the virtual generated column we can make sure the expression used in the query is the exact same expression we indexed. PostgreSQL is then able to use the index, and the query is fast.

There are several advantages to this approach:

1. **Smaller index**: fewer distinct values means the database can use deduplication to make the index smaller.

2. **Faster query**: the small and specific index requires less resources so the query is faster.

3. **No discipline**: using the generated column is straight forward and the index is guaranteed to be useable.

4. **No ambiguity**: making sure anyone on the team uses the same exact expression is prone to errors and discrepancies, especially when time zones are involved. Using a virtual generated column eliminates this ambiguity.

Indexing Virtual Generated Columns

The next logical step would be to create the index directly on the virtual column. Unfortunately, as of writing this article, PostgreSQL 18 does not support indexes on virtual generated columns:

```
db=# CREATE INDEX sale_sold_at_date_ix ON sale(sold_at_date);
ERROR:  indexes on virtual generated columns are not supported
```

Hopefully indexes on virtual generated columns will make it to PostgreSQL 19.

. . .

Enforce Uniqueness with Hash Index

Imagine you have a system that extracts information from URLs. You create a table to keep track:

```
CREATE TABLE urls (
    id INT PRIMARY KEY,
    url TEXT NOT NULL,
    data JSON
);
```

Create some entries:

```
db=# INSERT INTO urls (id, url)
SELECT n, 'https://' || uuidv4() || '.com/ ' || uuidv4() || '?p=' || uuidv4()
FROM generate_series(1, 1_000_000) AS t(n);
INSERT 0 1000000
```

Processing web pages can be resource intensive, time consuming and expensive, so you want to make sure you don't process the same page more than once.

Slap a Unique B-Tree on it

To make sure URLs are not processed more than once you add a unique constraint on the `url` column:

```
db=# CREATE UNIQUE INDEX urls_url_unique_ix ON urls(url);
```

You can now rest assured that you don't process the exact URL more than once:

```
db=# INSERT INTO urls(id, url) VALUES (1_000_001, 'https://hakibenita.com');
INSERT 0 1

db=# INSERT INTO urls(id, url) VALUES (1_000_002, 'https://hakibenita.com');
ERROR:  duplicate key value violates unique constraint "urls_url_unique_ix"
DETAIL:  Key (url)=(https://hakibenita.com) already exists.
```

The unique constraint is enforced using a unique B-Tree index, so you also get the nice perk of being able to search for a specific URL very quickly:

```
db=# EXPLAIN (ANALYZE ON, BUFFERS OFF, COSTS OFF)
SELECT * FROM urls WHERE url = 'https://hakibenita.com';
                                        QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────
 Index Scan using urls_url_unique_ix on urls (actual time=0.018..0.018 rows=1.00 lo
   Index Cond: (url = 'https://hakibenita.com'::text)
   Index Searches: 1
```

```
  Planning Time: 0.173 ms
  Execution Time: 0.046 ms
```

Web pages these days can have pretty big URLs. Some web apps even go as far as storing the entire application state in the URL. This is great for users, but not so great if you need to store these URLs.

Check the size of the table and the B-Tree index used to enforce the unique constraint:

```
db=# \dt+ urls
List of tables
─[ RECORD 1 ]─┬────────────
Schema        │ public
Name          │ urls
Type          │ table
Owner         │ haki
Persistence   │ permanent
Access method │ heap
Size          │ 160 MB
Description   │ ¤

db=# \di+ urls_url_unique_ix
List of indexes
─[ RECORD 1 ]─┬───────────────────
Schema        │ public
Name          │ urls_url_unique_ix
Type          │ index
Owner         │ haki
Table         │ urls
Persistence   │ permanent
Access method │ btree
Size          │ 154 MB
Description   │ ¤
```

The size of the table is 160MB and the size of the index is a staggering 154MB!

Unique Hash Index

A B-Tree index stores the indexed values themselves in the leaf blocks, so when indexing large values, the B-Tree index can get very large.

PostgreSQL offers another type of index called a Hash index. This type of index does not store the actual values. Instead, it stores the hash values which can be much smaller. I wrote about Hash indexes in the past so I wont repeat myself. I would just say that indexing large values with very few repetition is where the Hash index truly shines!

A Hash index sounds like a reasonable way to enforce a unique constraint, so let's try to create a unique hash index:

```
db=# CREATE UNIQUE INDEX urls_url_unique_hash ON urls USING HASH(url);
ERROR:  access method "hash" does not support unique indexes
```

Oh no! PostgreSQL does not support unique hash indexes, but this doesn't mean we can't still enforce uniqueness using a Hash index...

Enforcing Uniqueness Using a Hash Index

PostgreSQL offers a special type of constraint called an exclusion constraint. This lesser-known and not-so-widely-used constraint is often mentioned in combination with a GIN or GiST index as a way to prevent overlapping ranges. However, using an exclusion constraint we can effectively enforce uniqueness using a Hash index:

```
db=# ALTER TABLE urls ADD CONSTRAINT urls_url_unique_hash EXCLUDE USING HASH (url W
ALTER TABLE
```

This adds an exclusion constraint on the table that prevents two rows with the same URL - this guarantees uniqueness. The exclusion constraint is enforced using a Hash index - this means we effectively enforce uniqueness with a Hash index!

First, verify that uniqueness is indeed enforced:

```
db=# INSERT INTO urls (id, url) VALUES (1_000_002, 'https://hakbenita.com/postgresq
INSERT 0 1

db=# INSERT INTO urls (id, url) VALUES (1_000_003, 'https://hakbenita.com/postgresq
ERROR:  conflicting key value violates exclusion constraint "urls_url_unique_hash"
```

```
DETAIL:  Key (url)=(https://hakbenita.com/postgresql-hash-index) conflicts with
existing key (url)=(https://hakbenita.com/postgresql-hash-index).
```

Attempting to add a row with a URL that already exists failed with an exclusion constraint violation. Good.

Next, can this Hash index be useful for queries that filter for specific urls?

```
db=# EXPLAIN (ANALYZE ON, BUFFERS OFF, COSTS OFF)
SELECT * FROM urls WHERE url = 'https://hakibenita.com';
                                        QUERY PLAN
─────────────────────────────────────────────────────────────────────────────────────
 Index Scan using urls_url_unique_hash on urls (actual time=0.010..0.011 rows=1.00
   Index Cond: (url = 'https://hakibenita.com'::text)
   Index Searches: 1
 Planning Time: 0.178 ms
 Execution Time: 0.022 ms
```

Yes it can, and in this case it's even faster than using the B-Tree index (0.022ms vs 0.046ms).

Finally, compare the size of the B-Tree and the Hash index:

```
db=# \di+ urls_url_*
                List of indexes
        Name          │ Access method │  Size
──────────────────────┼───────────────┼────────
urls_url_unique_hash │ hash          │ 32 MB
urls_url_unique_ix   │ btree         │ 154 MB
```

Amazing! The Hash index is x5 smaller than the corresponding B-Tree index. Instead of storing those large URLs in the B-Tree leaf blocks, the Hash index stores only the hash values which results in a significantly smaller index.

Limitation of "Unique" Exclusion Constraints

Using an exclusion constraint to enforce uniqueness with a Hash index can potentially save storage and make queries faster. However, there are a few caveats to consider with this approach:

⚠️ Column cannot be referenced by foreign keys

PostgreSQL requires that a foreign key reference a unique constraint. Since we can't define a unique hash constraint, we can't point a foreign key to it:

```
db=# CREATE TABLE foo (url TEXT REFERENCES urls(url));
ERROR:  there is no unique constraint matching given keys for referenced table "url
```

⚠️ Limitations on `INSERT ... ON CONFLICT`

The `ON CONFLICT` clause in an `INSERT` command is common and very useful for syncing data. Unfortunately, using exclusion constraints with this clause can have some rough edges.

Attempting to use an exclusion constraint with a list of fields in an `ON CONFLICT ... DO NOTHING` clause can fail:

```
db=# INSERT INTO urls (id, url) VALUES (1_000_004, 'https://hakibenita.com')
ON CONFLICT (url) DO NOTHING;
ERROR:  there is no unique or exclusion constraint matching the ON CONFLICT specifi
```

The message suggests that it should be possible to use an exclusion, and it is, but using the `ON CONFLICT ON CONSTRAINT` clause instead:

```
db=# INSERT INTO urls (id, url) VALUES (1_000_004, 'https://hakibenita.com')
ON CONFLICT ON CONSTRAINT urls_url_unique_hash DO NOTHING;
INSERT 0 0
```

Trying the same with `ON CONFLICT ... DO UPDATE` is not possible at all, even when using `ON CONFLICT ON CONSTRAINT`:

```
db=# INSERT INTO urls (id, url) VALUES (1_000_004, 'https://hakibenita.com')
ON CONFLICT ON CONSTRAINT urls_url_unique_hash DO UPDATE SET id = EXCLUDED.id;
ERROR:  ON CONFLICT DO UPDATE not supported with exclusion constraints
```

I'm not a big fan of using the constraint names in SQL, so to overcome both limitations I'de use MERGE instead:

```
db=# MERGE INTO urls t
USING (VALUES (1000004, 'https://hakibenita.com')) AS s(id, url)
ON t.url = s.url
WHEN MATCHED THEN UPDATE SET id = s.id
WHEN NOT MATCHED THEN INSERT (id, url) VALUES (s.id, s.url);
MERGE 1
```

Finally, check the execution plan to verify that the statement is capable of using the Hash index:

```
                                        QUERY PLAN
──────────────────────────────────────────────────────────────────────────────────────
Merge on urls t  (cost=0.00..8.04 rows=0 width=0)
  ->  Nested Loop Left Join  (cost=0.00..8.04 rows=1 width=6)
      ->  Result  (cost=0.00..0.01 rows=1 width=0)
      ->  Index Scan using urls_url_unique_hash on urls t  (cost=0.00..8.02 rows=1 w
          Index Cond: (url = 'https://hakibenita.com'::text)
```

It can, and it did!

Despite these minor limitations and inconveniences, a Hash index is a good candidate for enforcing uniqueness of large values that don't need to be referenced by foreign keys.

. . .

Want me to send you an email when I publish something new?

Yes Please!

. . .

Share to show you care

. . .

DISCUSSIONS

Y Unconventional PostgreSQL Optimizations	9 102
Unconventional PostgreSQL Optimizations	7 63
L Unconventional PostgreSQL Optimizations	3 17
Unconventional PostgreSQL Optimizations	0 3
Unconventional PostgreSQL Optimizations	0 2

Show 1 more discussions...

. . .

SIMILAR ARTICLES

27 JULY 2023 / POSTGRESQL, SQL, PERFORMANCE / 7 71
When Good Correlation is Not Enough
How outliers can trick the optimizer into the wrong plan

08 NOVEMBER 2021 / POSTGRESQL / 216 1315
Lesser Known PostgreSQL Features
Features you already have but may not know about!

11 JANUARY 2021 / POSTGRESQL, PERFORMANCE, SQL / 5 131
Re-Introducing Hash Indexes in PostgreSQL
The Ugly Duckling of index types

20 OCTOBER 2020 / SQL, PERFORMANCE, POSTGRESQL / 66 579
The Surprising Impact of Medium-Size Texts on PostgreSQL Performance
Why TOAST is the best thing since sliced bread

21 NOVEMBER 2019 / POSTGRESQL, SQL, PERFORMANCE / 186 678
12 Common Mistakes and Missed Optimization Opportunities in SQL
Made by Developers and Non-Developers