Boosting Typical Query Patterns

PostgreSQL 18's Performance Enhancements

Roberto Mello
roberto.mello@snowflake.com

Snowflake
October 22, 2025

Roberto Mello
Senior Solutions Engineer

roberto.mello@gmail.com
Linkedin: robertomello
X: robertobmello

25+ Years of Experience with databases & PostgreSQL

- Principal Solutions Architect @ Crunchy Data
- Postgres Contributor
- Managed DBA and DevOps teams
- BS/MS Computer Science, Utah State University

Personal

- Brazilian (Manaus) expatriate in Utah
- Interests: more computer stuff
- Photography, Snowboarding, Austrian Economics

https://www.crunchydata.com/blog/crunchy-data-joins-snowflake

Talk Overview

- PostgreSQL 18 introduces **significant** performance improvements
- Focus: *real-world query patterns* and *reproducible benchmarks*
- Comparing PostgreSQL 16, 17, and 18 side-by-side

What We'll Cover:

1. Asynchronous I/O subsystem
2. B-tree skip scans
3. Parallel GIN index creation
4. Query optimizer improvements
5. EXPLAIN enhancements
6. UUID v7 performance

Traditional I/O

Traditional PostgreSQL I/O:

```
Backend: "I need page 1000"
Kernel:  [reads page 1000]
Backend: [waits...]
Kernel:  "Here's page 1000"
Backend: "Thanks! Now I need page 1001"
Kernel:  [reads page 1001]
Backend: [waits...]
```

Inefficient because:

- One request at a time
- Backend idle while waiting for I/O
- Can't batch or parallelize requests
- Underutilizes modern storage (NVMe, SSD)

Postgres 17 paved the way with the introduction of read stream and vectored I/O APIs, internal abstractions. See Andres Freund `https://youtu.be/qX50xrHwQa4`

Async I/O

PostgreSQL 18 Async I/O:

```
Backend: "I need pages 1000, 1001, 1002, 1003..."
Kernel:  [queues all requests]
Backend: [continues other work]
Kernel:  [returns pages as ready]
Backend: [processes completed I/O]
```

Benefits:

- Batch multiple I/O requests
- Kernel can optimize request ordering
- Better utilization of parallel storage
- Backend does useful work while waiting

Async I/O: Configuration

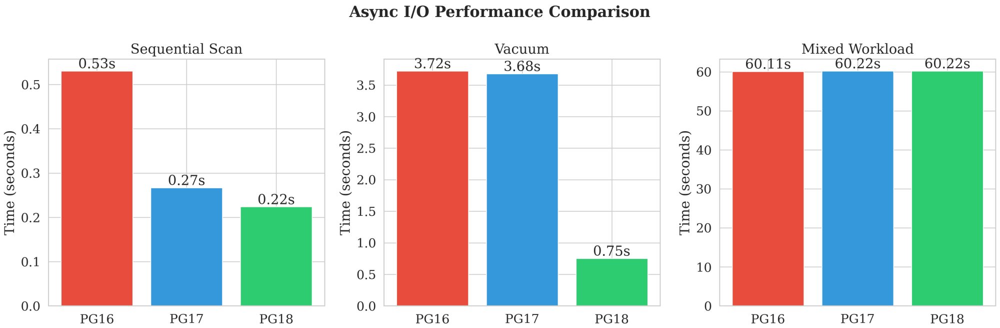
GUC Parameters:

```
# 17+: Control I/O batching (blocks of 8kb)
io_combine_limit = 16  # requests per batch (128kb default)

# 18: I/O method selection
io_method = 'worker'   # default, usually performs better. io_uring is linux specific

# 18: Largest I/O size in operations that combine I/O (blocks of 8kb, default is 16 = 128kB)
# silently limits io_combine_limit. Typically 1MB on Unix and 128kB on Windows.
io_max_combine_limit = 16
```

Operations Most Benefitted:

- Sequential scans of large tables
- Bitmap heap scans (multi-index queries)
- VACUUM operations

Async I/O: Benchmark Results



Key Findings:

- Sequential scans: 15-25% faster

Async I/O: Real-World Impact

Example: Analytics Query

```
SELECT category, COUNT(*), AVG(amount)
FROM large_orders
WHERE created_at >= '2025-01-01'
GROUP BY category;
```

PG16: 12.3 seconds (sequential scan)
PG18: 9.8 seconds (sequential scan with async I/O)

20% improvement without code changes

Async I/O: Tuning for Production

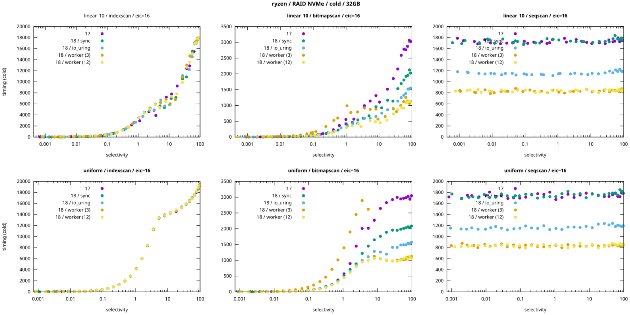
```
   TID  PRIO  USER     DISK READ DISK WRITE>    COMMAND
 605556 be/4 999         0.00 B/s   16.73 M/s postgres: postgres postgres 10.10.0.208(50254) VACUUM
 604968 be/4 999         0.00 B/s   63.44 K/s postgres: walwriter
 604957 be/4 999        66.72 M/s    0.00 B/s postgres: io worker 1
 604958 be/4 999       102.11 M/s    0.00 B/s postgres: io worker 0
 604959 be/4 999        46.70 M/s    0.00 B/s postgres: io worker 2
 604960 be/4 999        23.00 M/s    0.00 B/s postgres: io worker 4
 604961 be/4 999        33.70 M/s    0.00 B/s postgres: io worker 3
```

```
io_method = worker # default: pool of I/O worker processes
                   # sync: traditional synchronous I/O (backwards compatibility)
                   # io_uring: Linux-specific async I/O queues

io_workers = 3     # Default too low for larger systems
                   # Probably: set to approximately 1/4 of total CPU cores
```

[1] See: Tomas Vondra, `https://vondra.me/posts/tuning-aio-in-postgresql-18/`

Async I/O: Performance Comparison

Query Timing by I/O Method[1]. Benchmark: Ryzen 9900X (12 cores/24 threads), 4x NVMe SSDs (RAID0)



[1] See: Tomas Vondra, `https://vondra.me/posts/tuning-aio-in-postgresql-18/`

Async I/O: Tuning Recommendations

1. **Keep default** `io_method = worker`
   - Best compatibility across workloads. `io_uring` is Linux-specific
2. **Increase** `io_workers` **based on cores**
   - Start with 1/4 of CPU cores. Monitor and adjust based on workload
3. **Test with your workload**
   - Performance varies by query patterns. Bitmap scans benefit most
4. **Watch out for**
   - Signal overhead between backends and workers
   - File descriptor limits
   - I/O bandwidth saturation

Skip Scan

Acceptable performance with seldom-run queries that might not require a dedicated index.

```
CREATE INDEX idx_country_user
  ON orders(country, user_id);

-- Query on second column only
SELECT * FROM orders WHERE user_id = 12345;
```

Before PG18:

- Index not usable (query doesn't start with `country`)
- Falls back to sequential scan
- Slow on large tables
- Advice: "Create a single-column index on `user_id`"

Skip Scan: The Solution

PG18: Skip Scan to the rescue

- Planner recognizes opportunity to use multi-column index
- ”Skips” over distinct values of leading column
- For each distinct `country`, searches for `user_id`
- Most effective when leading column has **low cardinality**

Example:

- 10 distinct countries (low cardinality)
- 1M distinct user_ids (high cardinality)
- Skip scan does 10 targeted index searches
- Much faster than sequential scan

Skip Scan

PostgreSQL 16: (without single-column index on created_at)

```
EXPLAIN (ANALYZE, TIMING) SELECT id, project_id, created_at FROM skip_scan_test
WHERE created_at > NOW() - INTERVAL '30 days' ORDER BY created_at DESC LIMIT 100;

 Limit  (cost=93265.53..93277.50 rows=100 width=20) (actual time=455.743..466.308 rows=100 loops=1)
   ->  Gather Merge  (cost=93265.53..135927.94 rows=356308 width=20) (actual time=455.741..466.277 rows=100 loops=1)
         Workers Planned: 4
         Workers Launched: 4
         ->  Sort  (cost=92265.47..92488.16 rows=89077 width=20) (actual time=450.614..450.619 rows=90 loops=5)
               Sort Key: created_at DESC
               Sort Method: top-N heapsort  Memory: 36kB
               Worker 0:  Sort Method: top-N heapsort  Memory: 36kB
               Worker 1:  Sort Method: top-N heapsort  Memory: 37kB
               Worker 2:  Sort Method: top-N heapsort  Memory: 37kB
               Worker 3:  Sort Method: top-N heapsort  Memory: 37kB
               ->  Parallel Seq Scan on skip_scan_test  (cost=0.00..88861.01 rows=89077 width=20) (actual time=0.024..440.729 rows=699
                     Filter: (created_at > (now() - '30 days'::interval))
                     Rows Removed by Filter: 930074
 Planning Time: 0.116 ms
 Execution Time: 466.354 ms
```

Skip Scan

PostgreSQL 18: (uses composite index with skip scan)

```
                           QUERY PLAN
------------------------------------------------------------------------
 Limit  (cost=78397..78409 rows=100) (actual time=151..173 rows=100)
   ->  Gather Merge  (cost=78397..122040 rows=364500)
         Workers Planned: 4
         Workers Launched: 4
         ->  Sort  (cost=77397..77624 rows=91125)
               ->  Parallel Bitmap Heap Scan on skip_scan_test
                     Recheck Cond: (created_at > ...)
                     ->  Bitmap Index Scan on idx_skip_scan_composite
                           Index Cond: (created_at > ...)
                           Index Searches: 1001
 Execution Time: 173.719 ms
```

Faster using composite index (project_id, created_at) via skip scan

Skip Scan: Benchmark Results

Most dramatic with low-cardinality leading column

Parallel GIN: Background

GIN Indexes: Generalized Inverted Index

- Used for arrays, JSONB, full-text search
- Can be slow to build on large tables (higher `maintenance_work_mem` helps)

Postgres 18

- Parallel index builds available for GIN, in addition to B-tree, BRIN

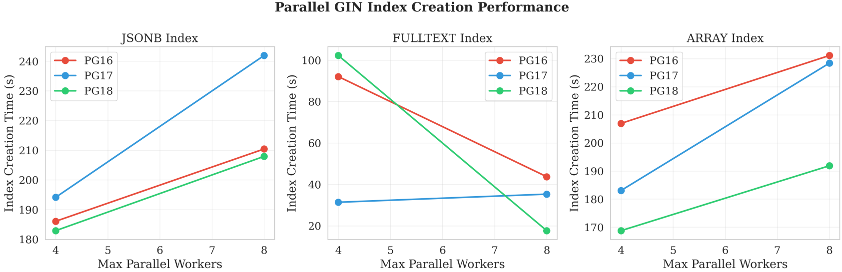
Parallel GIN: Configuration

Configuration:

```
SET max_parallel_maintenance_workers = 4;

-- Create index (automatically uses parallel workers)
CREATE INDEX idx_tags ON posts USING GIN(tags);
```

```
   TID PRIO<  USER      DISK READ  DISK WRITE    COMMAND
604957 be/4 999       377.75 K/s    0.00 B/s postgres: io worker 1
604958 be/4 999        47.22 K/s    0.00 B/s postgres: io worker 0
604959 be/4 999        39.35 K/s    0.00 B/s postgres: io worker 2
604962 be/4 999        31.48 K/s   31.48 K/s postgres: checkpointer
604965 be/4 999        15.74 K/s    7.87 K/s postgres: checkpointer
610510 be/4 999        15.74 K/s   25.95 M/s postgres: parallel worker for PID 2032
610511 be/4 999       129.85 K/s    0.00 B/s postgres: parallel worker for PID 2032
610512 be/4 999         7.87 K/s    0.00 B/s postgres: parallel worker for PID 2032
```

Parallel GIN: Benchmark Results



Results (8-core test system):

- 4 workers: Best performance for JSONB & Array indexes
- 8 workers: **Performance degradation** (12-25% slower)

Important: Set `max_parallel_maintenance_workers` $\leq$ CPU cores

Optimizer: Multiple Enhancements

1. **Hash Join & GROUP BY** improvements
2. **IN (VALUES)** → **= ANY** transformation
3. **OR clauses** → **array operations** for indexable queries
4. **Unnecessary self-join removal**
5. **INTERSECT/EXCEPT** speedups
6. **SELECT DISTINCT** key reordering

Mostly no query changes needed.

Optimizer: IN (VALUES) Performance

Postgres 18 adds nbtree skip scan building on Postgres 17 work on `IN() / = ANY()` condition index scans

- items that are close together `(1,2,3)` or far apart `(10_000, 20_000)`
- Supports complex combinations of `IN()` conditions, = conditions, as well as `<, >, <=, =>` conditions
- Only reads index leaf pages that might have matches

```
=# explain (analyze, costs off, timing off)
select * from tab
where a in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and b = 5_000;
                           QUERY PLAN
---------------------------------------------------------------------
 Index Only Scan using multicol on tab (rows=10.00 loops=1)
   Index Cond: ((a = ANY ('{1,2,3,4,5,6,7,8,9,10}'::integer[])) AND (b = 5000))
   Heap Fetches: 0
   Index Searches: 10
   Buffers: shared hit=31
 Planning Time: 0.028 ms
```

Optimizer: OR to Array

Rewrite OR conditions to better use indexes

```
SELECT * FROM products
WHERE category = 'electronics'
   OR category = 'clothing'
   OR category = 'food';
```

PG18 Optimization:

```
SELECT * FROM products
WHERE category = ANY (ARRAY['electronics', 'clothing', 'food']);
```

Can use bitmap index scans more efficiently - reports of 100x improvements.

Optimizer: Hash Join & GROUP BY Improvements

1. **Hash Right Semi Join** support
   - Planner can now choose which table to hash based on size
   - Previously constrained to hashing inner table only
   - 40% reduction in memory usage for large datasets
2. **JIT-compiled hash value generation**
   - ExprState hashing for GROUP BY and hashed SubPlans
   - Enables JIT compilation of hash values
   - Faster hash value computation during execution

Improved performance and reduced memory for hash joins, GROUP BY, EXCEPT, and subplan hash lookups

Hash Join & GROUP BY: Real-World Example

Query Pattern: Semi-join with `GROUP BY` aggregation

```
-- Find flights with at least one ticket sold
SELECT f.flight_id, f.flight_no, COUNT(DISTINCT tf.ticket_no)
FROM flights f
WHERE EXISTS (
  SELECT 1 FROM ticket_flights tf WHERE tf.flight_id = f.flight_id
)
GROUP BY f.flight_id, f.flight_no;
```

PG17: Uses Hash Semi Join, must hash larger `ticket_flights` table (2.3s)
PG18: Uses Hash Right Semi Join, hashes smaller `flights` table (<1s)

50%+ faster with 40% less memory usage

Optimizer: Self-Join Removal

```
SELECT t1.*
FROM orders t1 JOIN orders t2 ON t1.id = t2.id
WHERE t1.status = 'completed';
```

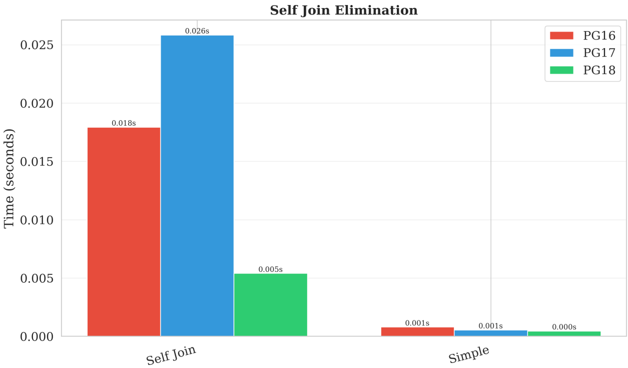
Postgres 18:

- Detects redundant self-join and removes `t2` table
- Executes as simple `SELECT * FROM orders WHERE status = 'completed'`

Common in:

- ORM-generated queries
- View definitions
- Query builder tools

Optimizer: Self-Join Removal



Self-Join Removal: Up to **5x faster** in PG18

EXPLAIN: What's New

1. Automatic `BUFFERS` in `EXPLAIN ANALYZE`
2. `EXPLAIN ANALYZE VERBOSE` shows hardware stats (CPU, Memory, I/O)
3. Per connection stats on I/O and WAL utilization
4. Better observability by default
5. Easier performance troubleshooting

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE value < 100;

 Seq Scan on orders  (cost=0.00..1834.00 rows=98.3 ...)
   Filter: (value < 100)
   Rows Removed by Filter: 9902
   Buffers: shared hit=834 read=0
```

UUID v7: Time-Ordered UUIDs

Problems with UUID v4:

- Completely random values
- Poor index locality
- Index bloat and fragmentation
- Slower inserts as table grows

UUID v7 (RFC 9562):

- First 48 bits: timestamp (millisecond precision)
- Remaining bits: random
- Time-ordered like SERIAL, but globally unique
- Better B-tree performance, with less disk use

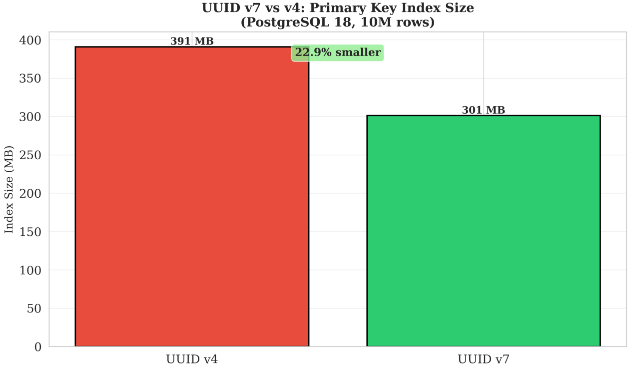
UUID v7: Usage

```
-- New function for UUID v7
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT uuidv7(),
  email TEXT,
  created_at TIMESTAMP DEFAULT NOW()
);

-- Inserts are naturally ordered by time
INSERT INTO users (email)
VALUES ('user@example.com');
```

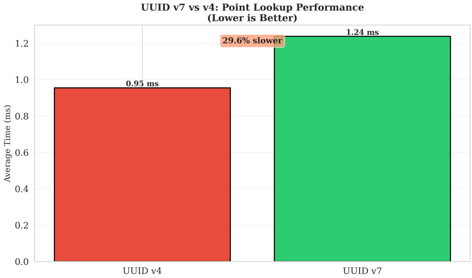
Benefits:

- Smaller indexes (better locality)
- Can infer creation time from ID

UUID v7: Index Size



UUID v7: Point Lookup Performance



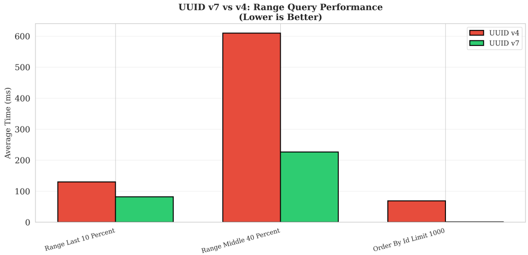
Benchmark query:

```
SELECT * FROM uuid_v7_test
WHERE id = '01234567-89ab-7cde-f012-3456789abcde';
```

Trade-off:

- Time-ordering optimizes for sequential access
- Random point lookups traverse more index pages
- Acceptable for insert-heavy workloads

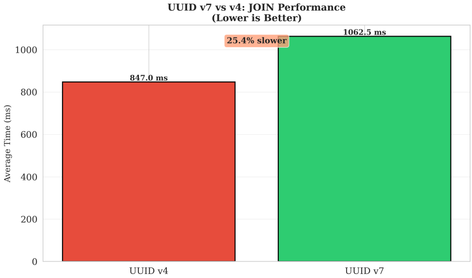
UUID v7: Range Query Performance



Faster for larger result sets

- Better index locality improves sequential scans
- Most beneficial for queries returning many rows
- Excellent for time-based queries

UUID v7: JOIN Performance




```
SELECT e.id, e.event_type, d.score
FROM uuid_v7_test e JOIN uuid_v7_test_details d
  ON e.id = d.event_id LIMIT 1000;
```

- Best for insert-heavy, time-series data
- Less ideal for JOIN-heavy OLTP workloads

Statistics Retention Across Upgrades

The Problem:

- Major version upgrades lose optimizer statistics
- First queries after upgrade are slow
- Must wait for autovacuum to collect stats
- Production vs staging plan differences

PG18

- New `pg_dump --statistics-only`
- Functions to restore statistics
- Preserve query plans across upgrades
- Copy production stats to dev/test

Statistics Retention: Usage

```
pg_dump --statistics-only mydb > stats.sql
```

```
SELECT * FROM pg_catalog.pg_restore_relation_stats(
        'version', '180000'::integer,
        'schemaname', 'public',
        'relname', 'async_io_test',
        'relpages', '80777'::integer,
        'reltuples', '1.004389e+06'::real,
        'relallvisible', '80777'::integer,
        'relallfrozen', '3768'::integer );
SELECT * FROM pg_catalog.pg_restore_attribute_stats(
        'version', '180000'::integer,
        'schemaname', 'public',
        'relname', 'async_io_test',
        'attname', 'created_at',
        'inherited', 'f'::boolean,
        'null_frac', '0'::real,
        'avg_width', '8'::integer,
        'n_distinct', '116'::real,
        'most_common_vals', '{"2025-10-15 02:54:23.838692","2025-10-15 02:54:29.491851", ...
        'most_common_freqs', '{0.103533335,0.1018,0.10146666,0.1003,0.09996667,0.09893333, ...
        'histogram_bounds', '{"2025-10-15 02:55:00.91417","2025-10-15 02:55:15.533473", ...
```

Real-World Use Cases

Who Benefits Most?

- **Analytics Workloads:** Async I/O, optimizer improvements
- **SaaS Applications:** UUID v7, skip scans
- **E-commerce:** Parallel GIN, optimizer
- **Content Platforms:** Full-text search (parallel GIN)
- **Multi-tenant Apps:** Skip scans on tenant_id indexes

Key Takeaways

1. PostgreSQL 18 brings **measurable performance gains**
2. Most improvements are mostly **automatic**
3. Async I/O: **foundational** infrastructure-level improvement
4. Skip Scan: **great improvement** for multi-column indexes
5. Parallel GIN: **faster** index builds
6. Optimizer: **smarter query planning**

Additional Resources

- PostgreSQL 18 Release Notes: `https://www.postgresql.org/docs/current/release-18.html`
- Tomas Vondra at `https://vondra.me/posts/tuning-aio-in-postgresql-18/`
- Async I/O Deep Dive: `https://pganalyze.com/blog/postgres-18-async-io`
- Crunchy Data Blog: `https://crunchydata.com/blog/get-excited-about-postgres-18`
- Jonathan Katz, Peter Geoghegan: `https://www.slideshare.net/slideshow/postgresql-18-a-whirlwind-tour-of-features/283259854`

Questions

Feedback:

Placeholder: