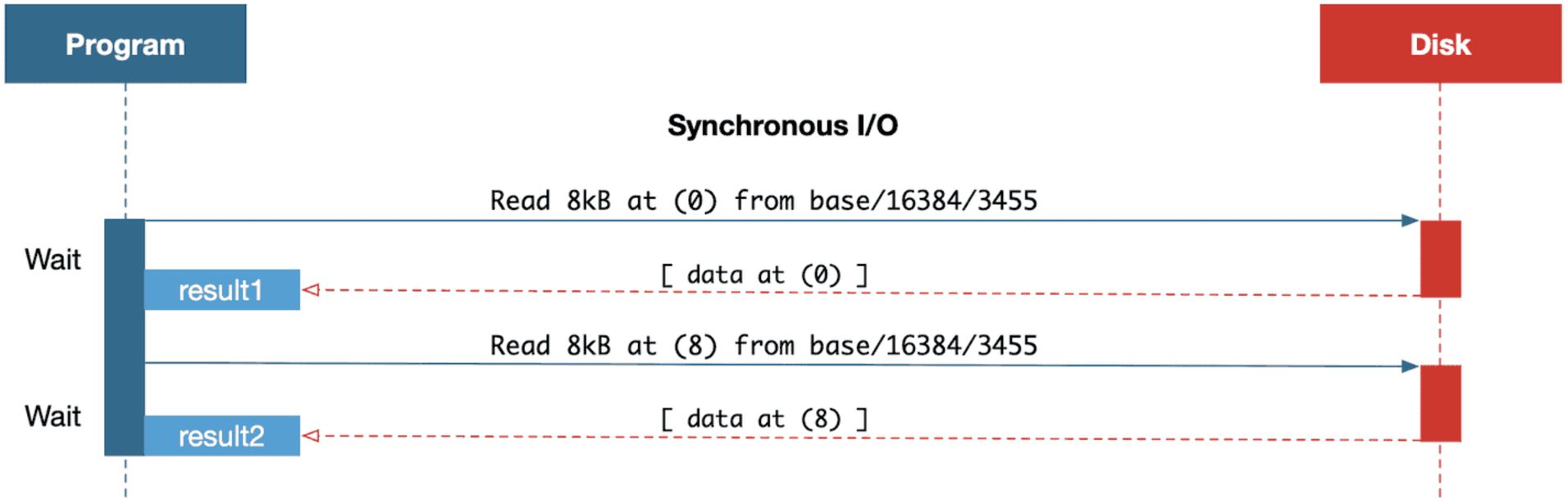
Hands-on Postgres 18

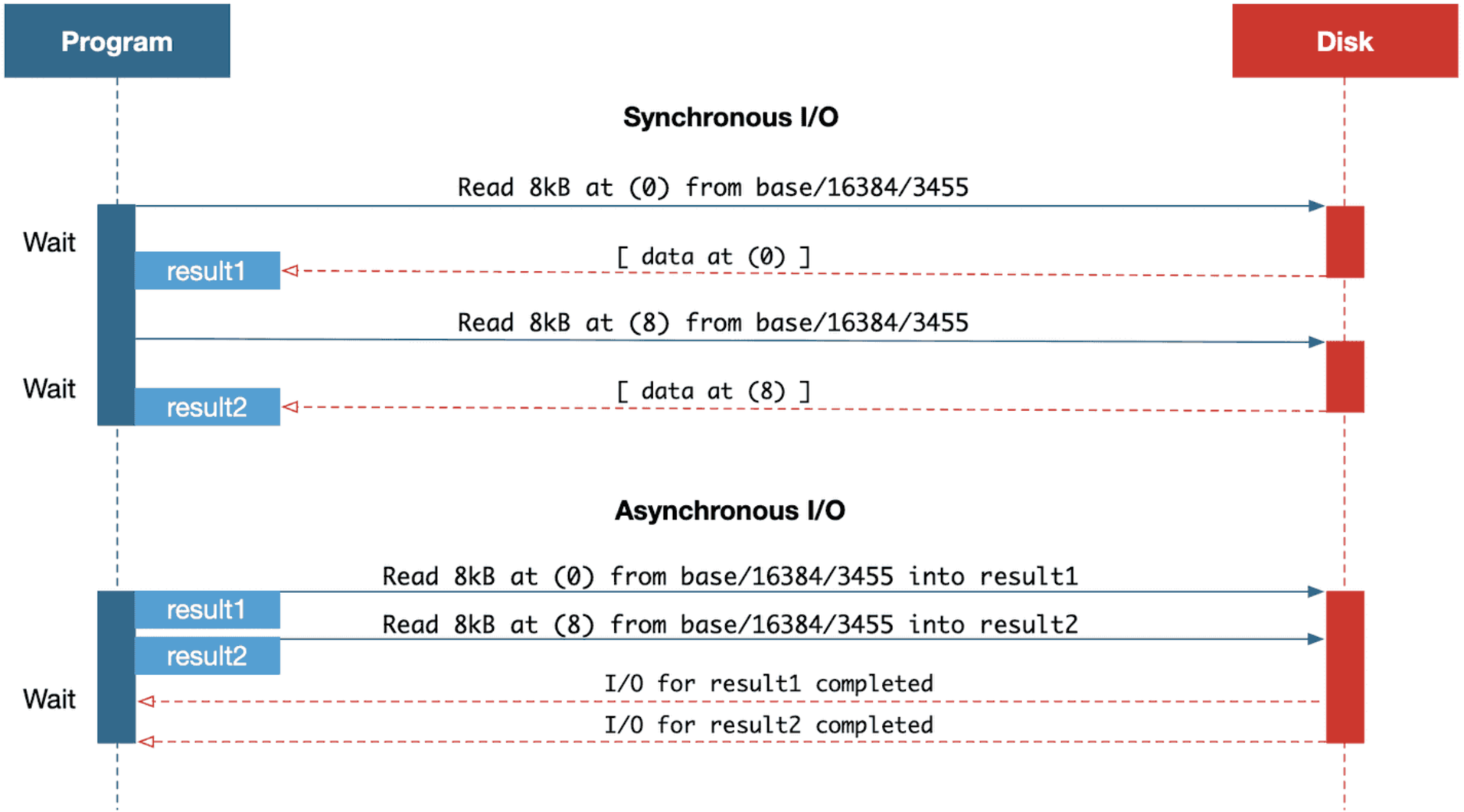
Async I/O, B-tree Skip Scan, UUIDv7, and More

lukas@pganalyze.com

1. Asynchronous Read I/O
2. B-Tree Skip Scan
3. UUIDv7
4. Planner changes
5. EXPLAIN changes
6. Plan ID tracking and pg_stat_plans
7. Other noteworthy changes

Asynchronous Read I/O


Program
Disk
Synchronous I/O
Read 8kB at (0) from base/16384/3455
Wait
result1
[data at (0)]
Read 8kB at (8) from base/16384/3455
Wait
result2
[data at (8)]


Program
Disk
Synchronous I/O
Read 8kB at (0) from base/16384/3455
Wait
result1
[data at (0)]
Read 8kB at (8) from base/16384/3455
Wait
result2
[data at (8)]
Asynchronous I/O
Read 8kB at (0) from base/16384/3455 into result1
result1
Read 8kB at (8) from base/16384/3455 into result2
result2
Wait
I/O for result1 completed
I/O for result2 completed

Read Streams (PG17) paved the road for Async I/O

Provide API for streaming relation data.

```
author    Thomas Munro <tmunro@postgresql.org>
          Tue, 2 Apr 2024 11:17:06 +0000 (00:17 +1300)
committer Thomas Munro <tmunro@postgresql.org>
          Tue, 2 Apr 2024 11:49:46 +0000 (00:49 +1300)
commit    b5a9b18cd0bc6f0124664999b31a00a264d16913
tree      b992882fb23757da9e1d9f5794d8d7b5b866da1f      tree
parent    210622c60e1a9db2e2730140b8106ab57d259d15      commit | diff
```

```
Provide API for streaming relation data.

Introduce an abstraction allowing relation data to be accessed as a
stream of buffers, with an implementation that is more efficient than
the equivalent sequence of ReadBuffer() calls.

Client code supplies a callback that can say which block number it wants
next, and then consumes individual buffers one at a time from the
stream.  This division puts read_stream.c in control of how far ahead it
can see and allows it to read clusters of neighboring blocks with
StartReadBuffers().  It also issues POSIX_FADV_WILLNEED advice ahead of
time when random access is detected.

Other variants of I/O stream will be proposed in future work (for
example to support recovery, whose LsnReadQueue device in
```

The io_method setting controls how I/O is done:

- **sync:** Similar to Postgres 17, I/O is synchronous and OS prefetching is utilized to achieve async read-ahead on some systems

- **worker (default):** I/O is started by each process as needed, but completions are performed by a pool of I/O worker processes (default 3).

- **io_uring**: Linux-specific API that uses a shared ring buffer to minimize syscall overhead, and avoiding the need for workers. Linux 6.5+ recommended.

Benchmarking Async I/O

AWS c7i.8xlarge instance (32 vCPUs, 64 GB RAM), with a dedicated 100GB io2 EBS volume for Postgres, with 20,000 provisioned IOPS. The test table was 3.5GB:

```
CREATE TABLE test(id int);
INSERT INTO test SELECT * FROM generate_series(0, 100000000);

test=# \dt+
                                    List of relations
 Schema | Name | Type  |  Owner   | Persistence | Access method |  Size   | Description
--------+------+-------+----------+-------------+---------------+---------+-------------
 public | test | table | postgres | permanent   | heap          | 3458 MB |
(1 row)
```

Postgres 17

```
test=# SELECT COUNT(*) FROM test;
   count
-----------
 100000001
(1 row)

Time: 15830.880 ms (00:15.831)
```

Postgres 18 (sync)

```
test=# SELECT COUNT(*) FROM test;
   count
-----------
 100000001
(1 row)

Time: 15071.089 ms (00:15.071)
```

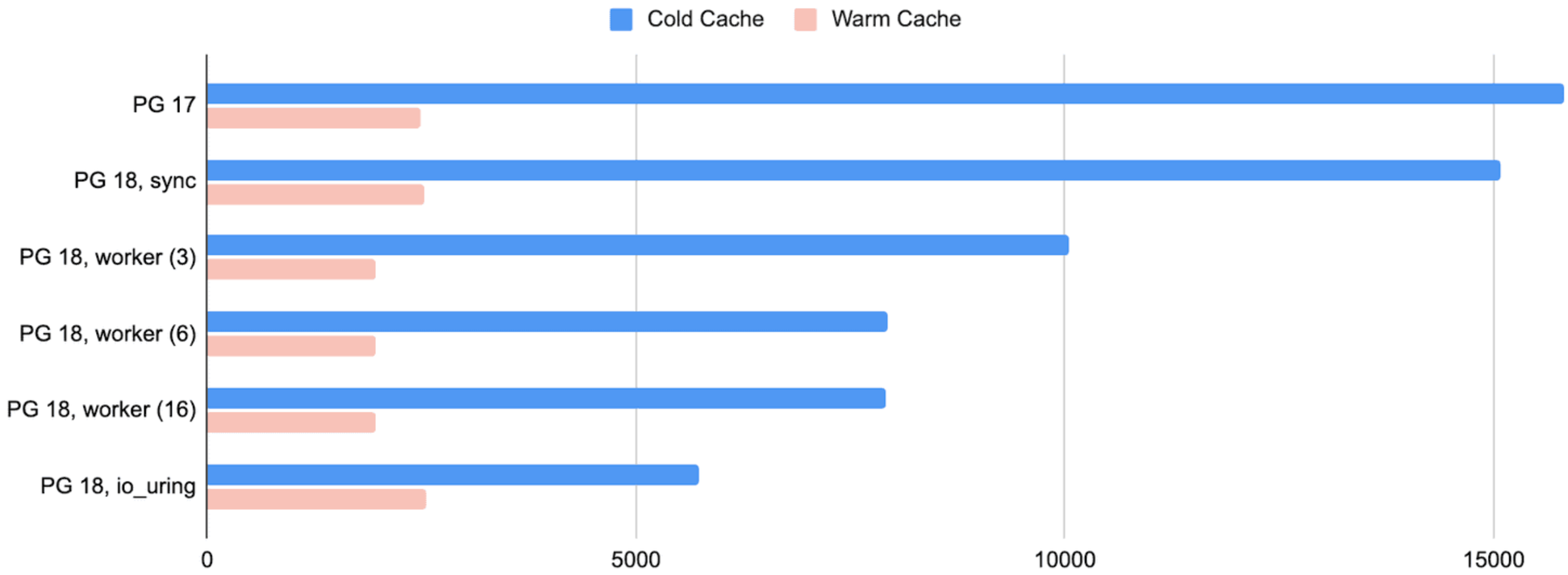
Postgres 18 (worker, 3)

```
test=# SELECT COUNT(*) FROM test;
   count
-----------
 100000001
(1 row)

Time: 10051.975 ms (00:10.052)
```

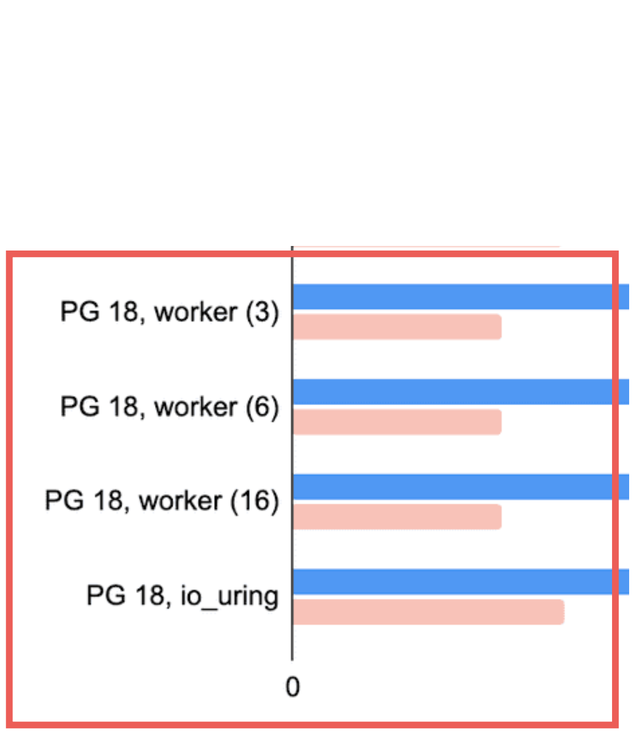
Postgres 18 (io_uring)

```
test=# SELECT COUNT(*) FROM test;
   count
-----------
 100000001
(1 row)

Time: 5723.423 ms (00:05.723)
```

Benchmarking Async I/O



Benchmarking Async I/O



Why is io_uring slower than worker for a warm cache?

Workers enable parallelism when moving data from the OS page cache to Postgres shared buffers.

Tuning Async I/O settings in Postgres 18

- **effective_io_concurrency:** New default of 16 (was 1), multiplied with io_combine_limit determines how many asynchronous read I/Os are issued:

- **io_combine_limit**: Controls the largest combined I/O size (default 128 kB)

maximum read-ahead = effective_io_concurrency × io_combine_limit

- **io_workers**: How many I/O workers to spawn for worker method. Default 3. Recommended to raise for larger machines (e.g. try between 10 and 20).

How would I know that I need more I/O workers?

Previously you may have seen the wait event "IO / DataFileRead" on I/O heavy systems. With the worker method you now see that on the I/O workers instead, with the new "AioIoCompletion" event on the connection itself:

```
  backend_type   | state  | wait_event_type |   wait_event
-----------------+--------+-----------------+-----------------
 client backend | active | IO              | AioIoCompletion
 io worker      |        | IO              | DataFileRead
 io worker      |        | IO              | DataFileRead
 io worker      |        | IO              | DataFileRead
(4 rows)
```

New monitoring view: pg_aios

See I/O requests in flight:

```
SELECT * FROM pg_aios;

  pid  | io_id | io_generation |    state     | operation |    off    | length | target | handl
-------+-------+---------------+--------------+-----------+-----------+--------+--------+------
 91452 |     1 |          4781 | SUBMITTED    | read      | 996278272 | 131072 | smgr   |
 91452 |     2 |          4785 | SUBMITTED    | read      | 996147200 | 131072 | smgr   |
 91452 |     3 |          4796 | SUBMITTED    | read      | 996409344 | 131072 | smgr   |
 91452 |     4 |          4802 | SUBMITTED    | read      | 996016128 | 131072 | smgr   |
 91452 |     5 |          3175 | COMPLETED_IO | read      | 995885056 | 131072 | smgr   |
(5 rows)
```

Heads up: I/O timing requires (more) interpretation in Postgres 18

Postgres 17:

```
test=# EXPLAIN (ANALYZE, BUFFERS, TIMING OFF) SELECT COUNT(*) FROM test;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1692478.40..1692478.41 rows=1 width=8) (actual rows=1 loops=1)
   Buffers: shared read=442478
   I/O Timings: shared read=14779.316
   ->  Seq Scan on test  (...)
         Buffers: shared read=442478
         I/O Timings: shared read=14779.316
...
 Execution Time: 18006.405 ms
(11 rows)
```

Heads up: I/O timing requires (more) interpretation in Postgres 18

Postgres 18 (worker):

```
test=# EXPLAIN (ANALYZE, BUFFERS, TIMING OFF) SELECT COUNT(*) FROM test;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1692478.40..1692478.41 rows=1 width=8) (actual rows=1.00 loops=1)
   Buffers: shared read=442478
   I/O Timings: shared read=7218.835
   ->  Seq Scan on test  (...)
         Buffers: shared read=442478
         I/O Timings: shared read=7218.835
...
 Execution Time: 10480.827 ms
(11 rows)
```

Heads up: I/O timing requires (more) interpretation in Postgres 18

"I/O Timings" is a measure of time spent waiting for the I/O to complete, *not* a measure of how much effort the I/O took.

Different than parallel query, the effective I/O wait time taken up by I/O workers does not get reported back to the original connection.

I/O time is still useful to understand OS page cache hits.

What about Direct I/O?

Technically possible with 18, but not recommended.

Direct I/O (not using the OS page cache) requires implementing Async I/O for write operations to avoid performance regressions.

Hopefully in Postgres 19!

B-Tree Skip Scan

Builds on improved handling of ScalarArrayOpExpr (SAOPs) in PG17

Enhance nbtree ScalarArrayOp execution.

```
author    Peter Geoghegan <pg@bowt.ie>
          Sat, 6 Apr 2024 15:47:10 +0000 (11:47 -0400)
committer Peter Geoghegan <pg@bowt.ie>
          Sat, 6 Apr 2024 15:47:10 +0000 (11:47 -0400)
```

```
Commit 9e8da0f7 taught nbtree to handle ScalarArrayOpExpr quals
natively.  This works by pushing down the full context (the array keys)
to the nbtree index AM, enabling it to execute multiple primitive index
scans that the planner treats as one continuous index scan/index path.
This earlier enhancement enabled nbtree ScalarArrayOp index-only scans.
It also allowed scans with ScalarArrayOp quals to return ordered results
(with some notable restrictions, described further down).

Take this general approach a lot further: teach nbtree SAOP index scans
to decide how to execute ScalarArrayOp scans (when and where to start
the next primitive index scan) based on physical index characteristics.
This can be far more efficient.  All SAOP scans will now reliably avoid
duplicative leaf page accesses (just like any other nbtree index scan).
SAOP scans whose array keys are naturally clustered together now require
far fewer index descents, since we'll reliably avoid starting a new
primitive scan just to get to a later offset from the same leaf page.

The scan's arrays now advance using binary searches for the array
element that best matches the next tuple's attribute value.  Required
```

Postgres 18 generates "skip arrays" for columns omitted in the query, that are in the index

Add nbtree skip scan optimization.

```
author    Peter Geoghegan <pg@bowt.ie>
          Fri, 4 Apr 2025 16:27:04 +0000 (12:27 -0400)
committer Peter Geoghegan <pg@bowt.ie>
          Fri, 4 Apr 2025 16:27:04 +0000 (12:27 -0400)
```

```
Teach nbtree multi-column index scans to opportunistically skip over
irrelevant sections of the index given a query with no "=" conditions on
one or more prefix index columns.  When nbtree is passed input scan keys
derived from a predicate "WHERE b = 5", new nbtree preprocessing steps
output "WHERE a = ANY(<every possible 'a' value>) AND b = 5" scan keys.
That is, preprocessing generates a "skip array" (and an output scan key)
for the omitted prefix column "a", which makes it safe to mark the scan
key on "b" as required to continue the scan.  The scan is therefore able
to repeatedly reposition itself by applying both the "a" and "b" keys.

A skip array has "elements" that are generated procedurally and on
demand, but otherwise works just like a regular ScalarArrayOp array.
Preprocessing can freely add a skip array before or after any input
ScalarArrayOp arrays.  Index scans with a skip array decide when and
where to reposition the scan using the same approach as any other scan
with array keys.  This design builds on the design for array advancement
```

This optimization works best for low cardinality omitted columns

e.g. there are 10,000 distinct "a" values that are skipped, vs 10 million "b" values.

```
CREATE TABLE test(a int, b int);
INSERT INTO test SELECT val / 1000, val FROM generate_series(0, 10000000) _(val);

CREATE INDEX ON test(a, b);

SELECT * FROM test WHERE b = 100;
```

Postgres 17

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE b = 100;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..169247.74 rows=1 width=8) (actual time=0.108..235.780 rows=1
loops=1)
   Filter: (b = 100)
   Rows Removed by Filter: 10000000
 Planning Time: 0.118 ms
 Execution Time: 235.794 ms
(5 rows)
```

Postgres 17 (with seqscan = off)

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE b = 100;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using test_a_b_idx on test  (cost=0.43..184688.29 rows=1 width=8) (actual
time=0.087..136.433 rows=1 loops=1)
   Index Cond: (b = 100)
   Heap Fetches: 0
 Planning Time: 0.354 ms
 Execution Time: 136.458 ms
(5 rows)
```

Postgres 18

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE b = 100;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using test_a_b_idx on test  (cost=0.43..35388.90 rows=1 width=8) (actual
time=0.059..22.238 rows=1.00 loops=1)
   Index Cond: (b = 100)
   Heap Fetches: 0
   Index Searches: 10001
   Buffers: shared hit=30004
 Planning Time: 0.178 ms
 Execution Time: 22.272 ms
(7 rows)
```

136 ms to 22ms!

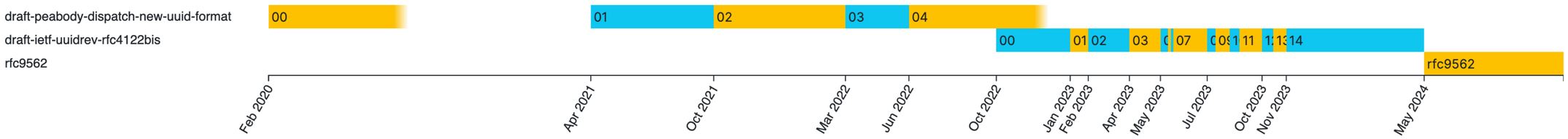
UUIDv7

UUIDv7 is part of a new UUID RFC that was finalized in 2024

Universally Unique IDentifiers (UUIDs)

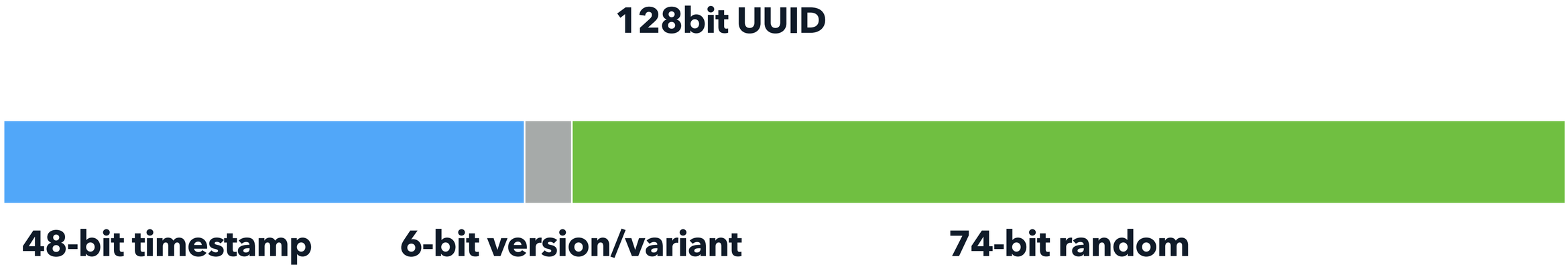
RFC 9562

Status | Email expansions | History



Document	**Type**	RFC - Proposed Standard (May 2024) Errata IPR Obsoletes RFC 4122 Was draft-ietf-uuidrev-rfc4122bis (uuidrev WG)
	Authors	Kyzer R. Davis ✉, Brad Peabody ✉, P. Leach ✉

UUIDv7 are timestamp-prefixed, partially random UUIDs



You could already use UUIDv7 before Postgres 18, but now the server can assign them!

```
select uuidv7();

                uuidv7
--------------------------------------
 0199815b-e874-706a-9612-34f33fe0bc33
(1 row)
```

UUIDv7 is a significantly better choice for indexed columns over UUIDv4

```
CREATE TABLE test (id uuid DEFAULT uuidv7());
CREATE TABLE test2 (id uuid DEFAULT uuidv4());

CREATE INDEX ON test(id);
CREATE INDEX ON test2(id);

INSERT INTO test SELECT FROM generate_series(0, 10_000_000);
INSERT INTO test2 SELECT FROM generate_series(0, 10_000_000);
```

UUIDv7 is a significantly better choice for indexed columns over UUIDv4

```
postgres=# \di+ test_id_idx
                                                 List of relations
 Schema |    Name     | Type  |  Owner   | Table | Persistence | Access method |  Size  | Descr
--------+-------------+-------+----------+-------+-------------+---------------+--------+------
 public | test_id_idx | index | postgres | test  | permanent   | btree         | 301 MB |
(1 row)

postgres=# \di+ test2_id_idx
                                                  List of relations
 Schema |     Name     | Type  |  Owner   | Table | Persistence | Access method |  Size  | Desc
--------+--------------+-------+----------+-------+-------------+---------------+--------+-----
 public | test2_id_idx | index | postgres | test2 | permanent   | btree         | 386 MB |
(1 row)
```

UUIDv4: 386 MB, UUIDv7: 301 MB

You can extract timestamps from UUIDv7 values

```
SELECT uuid_extract_timestamp('0199815b-e874-706a-9612-34f33fe0bc33');

   uuid_extract_timestamp
----------------------------
 2025-09-25 14:51:48.212+00
(1 row)
```

You can use UUIDv7 for time-based partitioning, if you really want to

```
CREATE TABLE test (id uuid PRIMARY KEY, data text) PARTITION BY RANGE(id);
CREATE INDEX ON test(data);

CREATE TABLE test_p20250924 PARTITION OF test FOR VALUES
FROM (overlay(uuidv7('2025-09-24' - now())::text placing '0000-0000-0000-000000000000' from
10)::uuid)
TO (overlay(uuidv7('2025-09-25' - now())::text placing '0000-0000-0000-000000000000' from
10)::uuid);

CREATE TABLE test_p20250925 PARTITION OF test FOR VALUES
FROM (overlay(uuidv7('2025-09-25' - now())::text placing '0000-0000-0000-000000000000' from
10)::uuid)
TO (overlay(uuidv7('2025-09-26' - now())::text placing '0000-0000-0000-000000000000' from
10)::uuid);
```

Values are placed in partitions based on their UUIDv7 timestamp

```
INSERT INTO test VALUES(uuidv7(), 'example1');

SELECT * FROM test WHERE data = 'example1';
                  id                  |   data
--------------------------------------+----------
 01998169-bc13-7422-819e-b47c60e540a5 | example1
(1 row)

SELECT * FROM test_p20250924 WHERE data = 'example1';
(0 rows)

SELECT * FROM test_p20250925 WHERE data = 'example1';
                  id                  |   data
--------------------------------------+----------
 01998169-bc13-7422-819e-b47c60e540a5 | example1
(1 row)
```

Searches without the ID look at all partitions

```
EXPLAIN (ANALYZE, BUFFERS OFF, COSTS OFF, TIMING OFF) SELECT * FROM test WHERE data = 'example1
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Append (actual rows=1.00 loops=1)
   ->  Bitmap Heap Scan on test_p20250924 test_1 (actual rows=0.00 loops=1)
         Recheck Cond: (data = 'example1'::text)
         ->  Bitmap Index Scan on test_p20250924_data_idx (actual rows=0.00 loops=1)
               Index Cond: (data = 'example1'::text)
               Index Searches: 1
   ->  Bitmap Heap Scan on test_p20250925 test_2 (actual rows=1.00 loops=1)
         Recheck Cond: (data = 'example1'::text)
         Heap Blocks: exact=1
         ->  Bitmap Index Scan on test_p20250925_data_idx (actual rows=1.00 loops=1)
               Index Cond: (data = 'example1'::text)
               Index Searches: 1
 Planning Time: 0.469 ms
 Execution Time: 0.084 ms
```

Searches with the ID can perform partition pruning

```
EXPLAIN (ANALYZE, BUFFERS OFF, COSTS OFF, TIMING OFF)
SELECT * FROM test WHERE id = '01998169-bc13-7422-819e-b47c60e540a5';

                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Index Scan using test_p20250925_pkey on test_p20250925 test (actual rows=1.00 loops=1)
   Index Cond: (id = '01998169-bc13-7422-819e-b47c60e540a5'::uuid)
   Index Searches: 1
 Planning Time: 0.364 ms
 Execution Time: 0.070 ms
(5 rows)
```

Timestamps leaking to the client can be a problem, UUIDv47 can be a solution:

README MIT license

UUIDv47 — UUIDv7-in / UUIDv4-out (SipHash-masked timestamp)

`uuidv47` lets you store sortable UUIDv7 in your database while emitting a UUIDv4-looking façade at your API boundary. It XOR-masks *only* the UUIDv7 timestamp field with a keyed SipHash-2-4 stream derived from the UUID's own random bits. The mapping is deterministic and exactly invertible.

- Header-only C (C89) · zero deps
- Deterministic, invertible mapping (exact round-trip)
- RFC-compatible version/variant bits (v7 in DB, v4 on the wire)
- Key-recovery resistant (SipHash-2-4, 128-bit key)
- Full tests provided
- Optional PostgreSQL extension (UUID type + operators/opclasses)

Table of contents

Contributors 4


aabbdev Lucas


sylph01 Ryo Kajiwara


taiseiue Taisei Uemura


hillac

Deployments 2

github-pages 1 hour ago

Languages

C 52.0% **HTML** 35.5%
PLpgSQL 8.1% **Makefile** 4.4%

Planner changes

Self-join elimination

```
CREATE TABLE tt(a int PRIMARY KEY, b text);

EXPLAIN
SELECT p.*
  FROM tt p
  JOIN (SELECT * FROM tt WHERE b ~~ 'a%') q ON p.a = q.a;
```

Self-join elimination

Postgres 17

```
                QUERY PLAN
-----------------------------------
 Nested Loop  (...)
   ->  Seq Scan on tt  (...)
         Filter: (b ~~ 'a%'::text)
   ->  Index Scan using tt_pkey on tt p (...)
         Index Cond: (a = tt.a)
(5 rows)
```

Postgres 18

```
                              QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on tt  (...)
   Filter: (b ~~ 'a%'::text)
(2 rows)
```

OR to Array transformations

```
EXPLAIN SELECT * FROM test WHERE id = 2 OR id = 5 OR id = 10;
```

Postgres 17

```
                          QUERY PLAN
-----------------------------------------------------
 Seq Scan on test  (cost=0.00..220.02 rows=3 width=4)
   Filter: ((id = 2) OR (id = 5) OR (id = 10))
(2 rows)
```

Postgres 18

```
                                  QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using test_id_idx on test  (cost=0.29..16.91 rows=3 width=8)
   Index Cond: (id = ANY ('{2,5,10}'::integer[]))
(2 rows)
```

Other planner changes:

1. DISTINCT key reordering
2. Right Semi Join
3. Incremental Sorts for Merge Joins
4. Partitionwise Join improvements
5. Transformation of IN (VALUES ...) to ANY
6. Improve the efficiency of planning queries accessing many partitions

EXPLAIN changes

Turning off planner settings uses a new "disabled" marker

Postgres 17

```
SET enable_seqscan = off;
EXPLAIN SELECT * FROM test;

                        QUERY PLAN
-----------------------------------------------------
Seq Scan on test  (cost=10000000000.00
..10000000035.50 rows=2550 width=4)
(1 row)
```

Postgres 18

```
SET enable_seqscan = off;
EXPLAIN SELECT * FROM test;

                        QUERY PLAN
-----------------------------------------------------
 Seq Scan on test  (cost=0.00..35.50 ...)
   Disabled: true
(2 rows)
```

BUFFERS is on by default when using EXPLAIN (ANALYZE)

Postgres 17

```
EXPLAIN ANALYZE SELECT * FROM test;

QUERY PLAN
-----------
 Seq Scan on test  (...)
 Planning Time: 0.074 ms
 Execution Time: 2.591 ms
(3 rows)
```

Postgres 18

```
EXPLAIN ANALYZE SELECT * FROM test;

QUERY PLAN
-----------
 Seq Scan on test  (...)
  Buffers: shared hit=45
 Planning Time: 0.089 ms
 Execution Time: 2.796 ms
(4 rows)
```

Row Counters are now shown as floats

Postgres 17

```
EXPLAIN ANALYZE SELECT * FROM test JOIN test2
...;

QUERY PLAN
-----------
 Nested Loop  (...)
 ->  Bitmap Heap Scan on test  (...)
     (actual time=... rows=1 loops=1)
 ->  Index Only Scan  ... on test2  (...)
     (actual time=... rows=0 loops=1)
(3 rows)
```

Postgres 18

```
EXPLAIN ANALYZE SELECT * FROM test JOIN test2
...;

QUERY PLAN
-----------
 Nested Loop  (...)
 ->  Bitmap Heap Scan on test  (...)
     (actual time=... rows=1.00 loops=1)
 ->  Index Only Scan ... on test2  (...)
     (actual time=... rows=0.50 loops=1)
(3 rows)
```

Index Searches show the efficiency of Index Scans (less = better)

Postgres 18

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE search_id IN (1, 2, 3, 10000);

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using test_search_id_idx on test  (cost=0.29..27.40 rows=301 width=8) (actual
time=0.094..0.182 rows=300.00 loops=1)
   Index Cond: (search_id = ANY ('{1,2,3,10000}'::integer[]))
   Index Searches: 2
   Buffers: shared hit=5 read=1
 Planning Time: 0.112 ms
 Execution Time: 0.230 ms
(6 rows)
```

Plan ID tracking and pg_stat_plans

PG18: Allow plugins to set a 64-bit plan identifier in PlannedStmt

projects / postgresql.git / commit

summary | shortlog | log | commit | commitdiff | tree
(parent: 8a3e401) | patch

Allow plugins to set a 64-bit plan identifier in PlannedStmt

```
author    Michael Paquier <michael@paquier.xyz>
          Mon, 24 Mar 2025 04:23:42 +0000 (13:23 +0900)
committer Michael Paquier <michael@paquier.xyz>
          Mon, 24 Mar 2025 04:23:42 +0000 (13:23 +0900)
commit    2a0cd38da5ccf70461c51a489ee7d25fcd3f26be
tree      000fe6d92b36523695dcb368d699ecf2ecd0f191      tree
parent    8a3e4011f02dd2789717c633e74fefdd3b648386      commit | diff
```

```
Allow plugins to set a 64-bit plan identifier in PlannedStmt

This field can be optionally set in a PlannedStmt through the planner
hook, giving extensions the possibility to assign an identifier related
to a computed plan.  The backend is changed to report it in the backend
entry of a process running (including the extended query protocol), with
semantics and APIs to set or get it similar to what is used for the
existing query ID (introduced in the backend via 4f0b0966c8).  The plan
ID is reset at the same timing as the query ID.  Currently, this
information is not added to the system view pg_stat_activity; extensions
can access it through PgBackendStatus.

Some patches have been proposed to provide some features in the planning
area, where a plan identifier is used as a key to know the plan involved
(for statistics, plan storage and manipulations, etc.), and the point of
this commit is to provide an anchor in the backend that extensions can
rely on for future work.   The reset of the plan identifier is
controlled by core and follows the same pattern as the query identifier
added in 4f0b0966c8.

The contents of this commit are extracted from a larger set proposed
originally by Lukas Fittl, that Sami Imseih has proposed as an
independent change, with a few tweaks sprinkled by me.

Author: Lukas Fittl <lukas@fittl.com>
Author: Sami Imseih <samimseih@gmail.com>
Reviewed-by: Bertrand Drouvot <bertranddrouvot.pg@gmail.com>
Reviewed-by: Michael Paquier <michael@paquier.xyz>
Discussion: https://postgr.es/m/CAP53Pkyow59ajFMHGpmb1BK9WHDypaWtUsS_5DoYUEfsa_Hktg@mail.gmail.com
Discussion: https://postgr.es/m/CAA5RZ0vyWd4r35uUBUmhngv8XqeiJUkJDDKkLf5LCoWxv-t_pw@mail.gmail.com
```

```
typedef struct PlannedStmt
{
    pg_node_attr(no_equal, no_query_jumble)

    NodeTag     type;

    /* select|insert|update|delete|merge|utility */
    CmdType     commandType;

    /* query identifier (copied from Query) */
    uint64      queryId;

    /* plan identifier (can be set by plugins) */
    uint64      planId;
```

In Postgres 18, you can now write an extension that sets **PlannedStmt.planId** in a **planner_hook,** and then uses it in **ExecutorFinish_hook** to track statistics.

Query ID

Differentiates by query structure. Set by core Postgres but extensions could define their own mechanism.

Plan ID

Differentiates by plan shape. Not set by core Postgres (as of 18), but extensions can set it.

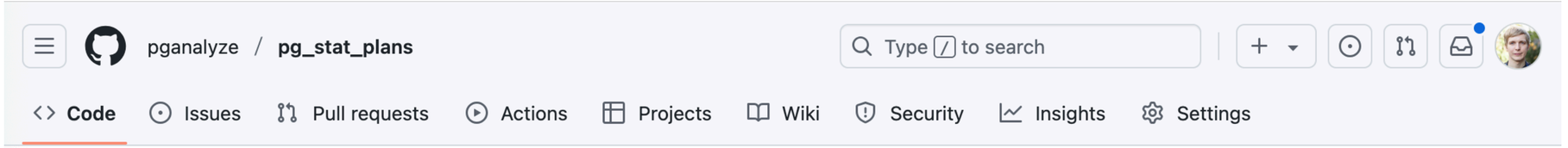
Hypothesis: A Plan ID should represent the "Plan Shape"

~ EXPLAIN (COSTS OFF)

```
Seq Scan on users
   Filter: (lower((email)::text) = '...'::text)
```

vs

```
 Bitmap Heap Scan on users
   Recheck Cond: (lower((email)::text) = '...'::text)
   ->  Bitmap Index Scan on index_users_lower_email
         Index Cond: (lower((email)::text) = '...'::text)
```



github.com/pganalyze/pg_stat_plans

Preview | Code | Blame | 207 lines (165 loc) · 16.3 KB | Raw

pg_stat_plans 2.0 - Track per-plan call counts, execution times and EXPLAIN texts in Postgres

`pg_stat_plans` is designed for low overhead tracking of aggregate plan statistics in Postgres, by relying on hashing the plan tree with a plan ID calculation. It aims to help identify plan regressions, and get an example plan for each Postgres query run, slow and fast. Additionally, it allows showing the plan for a currently running query.

Plan texts are stored in shared memory for efficiency reasons (instead of a local file), with support for `zstd` compression to compress large plan texts.

Plans have the same plan IDs when they have the same "plan shape", which intends to match `EXPLAIN (COSTS OFF)`. This extension is optimized for tracking changes in plan shape, but does not aim to track execution statistics for plans, like auto_explain can do for outliers.

This project is inspired by multiple Postgres community projects, including the original pg_stat_plans extension (unmaintained), with a goal of upstreaming parts of this extension into the core Postgres project over time.

Experimental. May still change in incompatible ways without notice. Not (yet) recommended for production use.

PG18: Introduce pluggable APIs for Cumulative Statistics

```
author    Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
committer Michael Paquier <michael@paquier.xyz>
          Sun, 4 Aug 2024 10:41:24 +0000 (19:41 +0900)
commit    7949d9594582ab49dee221e1db1aa5401ace49d4
tree      ad74385fbb0ef9f8b8d5a125d4b6e7ddc87ab20b      tree
parent    365b5a345b2680615527b23ee6befa09a2f784f2      commit | diff
```

```
Introduce pluggable APIs for Cumulative Statistics

This commit adds support in the backend for $subject, allowing
out-of-core extensions to plug their own custom kinds of cumulative
statistics.  This feature has come up a few times into the lists, and
the first, original, suggestion came from Andres Freund, about
pg_stat_statements to use the cumulative statistics APIs in shared
memory rather than its own less efficient internals.  The advantage of
this implementation is that this can be extended to any kind of
statistics.

The stats kinds are divided into two parts:
- The in-core "builtin" stats kinds, with designated initializers, able
to use IDs up to 128.
- The "custom" stats kinds, able to use a range of IDs from 128 to 256
(128 slots available as of this patch), with information saved in
TopMemoryContext.  This can be made larger, if necessary.

There are two types of cumulative statistics in the backend:
- For fixed-numbered objects (like WAL, archiver, etc.).  These are
attached to the snapshot and pgstats shmem control structures for
efficiency, and built-in stats kinds still do that to avoid any
redirection penalty.  The data of custom kinds is stored in a first
array in snapshot structure and a second array in the shmem control
structure, both indexed by their ID, acting as an equivalent of the
builtin stats.
- For variable-numbered objects (like tables, functions, etc.).  These
are stored in a dshash using the stats kind ID in the hash lookup key.

Internally, the handling of the builtin stats is unchanged, and both
fixed and variabled-numbered objects are supported.  Structure
definitions for builtin stats kinds are renamed to reflect better the
differences with custom kinds.
```

```
SELECT * FROM pg_stat_plans;


-[ RECORD 1 ]---+-----------------------------------------------------------------------------------------------
userid          | 10
dbid            | 16391
toplevel        | t
queryid         | -2322344003805516737
planid          | -1865871893278385236
calls           | 1
total_exec_time | 0.047708
plan            | Limit
                |   ->  Sort
                |         Sort Key: database_stats_35d.frozenxid_age DESC
                |         ->  Bitmap Heap Scan on database_stats_35d_20250514 database_stats_35d
```

Cumulative statistics on **which query ID used which plan, how often (calls)**, and **how long it took (total_exec_time).**

```
SELECT * FROM pg_stat_plans_activity;

  pid  |       plan_id        |                                   plan
-------+----------------------+------------------------------------------------------------------------------------------
 83994 | -5449095327982245076 | Merge Join
       |                      |   Merge Cond: ((a.datid = p.dbid) AND (a.usesysid = p.userid) AND (a.query_id = p.queryid) AND (a
       |                      |   ->  Sort
       |                      |         Sort Key: a.datid, a.usesysid, a.query_id, a.plan_id
       |                      |         ->  Function Scan on pg_stat_plans_get_activity a
       |                      |   ->  Sort
       |                      |         Sort Key: p.dbid, p.userid, p.queryid, p.planid
       |                      |         ->  Function Scan on pg_stat_plans p
       |                      |               Filter: (toplevel IS TRUE)
 87168 |  4721228144609632390 | Sort
       |                      |   Sort Key: q.id
       |                      |   ->  Nested Loop
       |                      |         ->  Index Scan using index_query_runs_on_server_id on query_runs q
```

Get the plan for a currently running query

(no progress tracking, just the plan that's being used)

Other noteworthy changes

Other new features:

1. OAuth authentication
2. Virtual generated columns
3. Temporal constraints / constraints over ranges

Other noteworthy improvements:

1. Better locking performance of queries that access many relations
2. Normal vacuums freeze some pages, even though they are all-visible
3. autovacuum_vacuum_max_threshold helps autovacuum scheduling for large tables
4. NUMA information in pg_buffercache

Two more things to know:

1. Checksums are now on by default
2. MD5 authentication is deprecated
3. Planner statistics can be preserved during upgrades (PG18 => PG19)

Thank you!

Try out pganalyze:

PGANALYZE.COM

Reach out for any questions:

lukas@pganalyze.com

Additional Q&A

Asynchronous Read I/O

Ebubekir B.:
Can you please shortly highlight the main difference of the io_method parameter options worker and io_uring? Can we say that in v18 starting with io_uring option is safe and better performance gain?

You can see a detailed explanation of the different I/O method settings in our Async I/O blog post from a few weeks ago: https://pganalyze.com/blog/postgres-18-async-io#new-io_method-setting-in-postgres-18

I would say that as a default, the “worker” setting is the right choice (as it has been made by upstream Postgres), because io_uring will not work for all systems. However if you have a modern Linux kernel, and no restrictions on using io_uring, it seems like a good choice to keep CPU overhead related to I/O operations the lowest.

Ozano N.:
The setting io_uring has correlation with effective_io_concurrency?

Yes - the effective_io_concurrency setting determines how much read ahead is done by Postgres (i.e. I/O operations that are scheduled that haven’t finished yet). This applies to all I/O methods, including io_uring.

Asynchronous Read I/O

Thomas H.
How do you determine how many worker should be for io_method? Is it following the total CPU count? Or?

Ideally I would tune it based on the workload, i.e. when you see all I/O workers consistently busy (based on pg_stat_activity), you increase it. If you're looking for a general measure, Tomas Vondra suggested that ¼ of total CPU cores might be reasonable: https://vondra.me/posts/tuning-aio-in-postgresql-18/#io_workers

Nikhil S.
Can io_uring be used for RHEL8?

Jakub Wartak has reviewed this on the PostgreSQL mailinglists: https://www.postgresql.org/message-id/CAKZiRmz+NSnQNLZR9CBKd6zctNs22ae48TcDNnh7M2g+SpOmiA@mail.gmail.com

Per the mailing list post, RHEL8 does not support io_uring, and RHEL9 discourages it. The RHEL9 kernel version (5.14.x) likely experiences the performance issues with io_uring + high max_connections described in that thread, which go away on Linux kernel 6.5+.

Asynchronous Read I/O

Joe L.
Do we have any idea/guess on limitations on async io on RDS?

[will be updated later today]

Rohit R.
Should we use Direct I/O on read replicas which are Read Only Workloads?

I haven't found any specific notes in this regard, so I'm not sure to be honest. This may be reasonable, or there might be a reason it's a bad idea. Please write a post about this if you do testing! As discussed, the reason Direct I/O (debug_io_direct) should not be used in PG18 is because there is no asynchronicity for write operations yet.

Nenad N.
Any concerns about WAL writer when using Direct I/O on read replicas?

Potentially - I have not researched this sufficiently to provide a good answer here.

B-Tree skip scan

Guy G.
Will a index-skip-scan also work with (a,b,c) and WHERE c=<value>?

Yes, skip scan also supports this situation, though because of the additional index data that has to be skipped over, it may often times not work as well, or the planner might not choose a plan that uses the index due to the costing not expecting it to work well.

David C.
How does b-tree skip scan help if there are large indexes?

Skip scan itself won't affect the index size (i.e. large indexes will still be large), since it's a change in how the Index Scans work.

However, if you have a large index where previously Postgres was slow to scan them because a lot of indexed values were not included in the Index condition, that may be faster with skip scan. Skip scan also allows you to consolidate indexes more efficiently (if you have a lot of similar indexes), so you could reduce your total number of indexes potentially.

B-Tree skip scan

Jan S.

Any idea what is the speed difference between the index with skip scan vs. normal index with just the leftmost column matching the where constraints?

It certainly depends on the specific case, but in the test case shown on the earlier slides, it is still a significant difference (15ms => <1ms) when you have an index that directly matches. Also note the difference in index searches:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE b = 100;

 Bitmap Heap Scan on test  (cost=184700.94..231677.71 rows=50000 width=8) (actual time=14.380..14.381 rows=1.00 loops=1)
   ...
   ->  Bitmap Index Scan on test_a_b_idx  (cost=0.00..184688.44 rows=50000 width=0) (actual time=14.367..14.368 rows=1.00 loops=1)
         Index Cond: (b = 100)
         Index Searches: 10001
 Planning Time: 0.054 ms
 Execution Time: 14.395 ms
(10 rows)

postgres=# CREATE INDEX ON test(b);
postgres=# EXPLAIN ANALYZE SELECT * FROM test WHERE b = 100;


 Bitmap Heap Scan on test  (cost=939.93..47916.70 rows=50000 width=8) (actual time=0.039..0.040 rows=1.00 loops=1)
   ...
   ->  Bitmap Index Scan on test_b_idx  (cost=0.00..927.43 rows=50000 width=0) (actual time=0.031..0.031 rows=1.00 loops=1)
         Index Cond: (b = 100)
         Index Searches: 1
...
 Execution Time: 0.055 ms
(12 rows)
```

UUIDv7

Lucio C.
Will the btree index the best/unique choice to index UUIDv7 values?

Yes, assuming you are mostly using equality comparisons (= operator) when searching for values, a B-tree index is going to be the best choice for UUIDv7.

David T.
You mention tool or extension that split UUIDv7 right part for client to hide time part, but what happend if client send some query back to server with just right part of UUIDv7, i think btree wouldn't be happy :) Or is there any other magic?

The mentioned UUIDv47 (https://github.com/stateless-me/uuidv47, which I have not used myself, but found interesting), does not just use the random portion, but actually keeps the full UUIDv7 information, but transforms it using an SipHash so the client does not have visibility into the timestamp portion (it instead looks like a random UUIDv4 type value). In my understanding the server side will transform such UUIDs back into a proper UUIDv7 before doing the database lookup.

UUIDv7

Marcelo F.
Isn't it faster to look up for a value in a UUIDv7-indexed column given it's ordered?

Correct, it would be faster to look up values in a UUIDv7 based column, because the internal index structure would be kept in a more efficient way (which will allow faster lookups as well).

Rick J.
If you change a column default from uuid_v4 to v7, would that really help your indexes going forward? Won't the uuid_v7 values end up somewhere in the middle with some uuid_v4 values greater and some less than, so you'll still have all the same messy page splits?

You are certainly correct that it will be less efficient to have a mix of UUIDv4 and UUIDv7, since you still have a change of page splits when the two meet (causing index bloat). However, if the choice is between keeping UUIDv4, or using a mix of the two, you could still see better index performance after switching (if the mix is not a problem for the application), and doing a REINDEX after sufficient new values have been added (to fix up some of the inefficient index structure).

(continued on next slide)

UUIDv7

(continued from last slide)

Here is a similar example shown on earlier slides, but comparing UUIDv4 vs UUIDv4 for the first 5 million values, and then switching to UUIDv7:

```
CREATE TABLE test2 (id uuid DEFAULT uuidv4());
CREATE TABLE test_mix (id uuid DEFAULT uuidv4());
INSERT INTO test_mix SELECT FROM generate_series(0, 5_000_000);
ALTER TABLE test_mix ALTER COLUMN id DEFAULT uuidv7();
INSERT INTO test_mix SELECT FROM generate_series(0, 5_000_000);

postgres=# \di+ test_mix_id_idx
                                             List of relations
 Schema |      Name       | Type  |  Owner   |  Table   | Persistence | Access method |  Size  |
Description
--------+-----------------+-------+----------+----------+-------------+---------------+--------
+-------------
 public | test_mix_id_idx | index | postgres | test_mix | permanent   | btree         | 467 MB |
(1 row)
```

(continues on next slide)

UUIDv7

(continued from last slide)

Note at this point the index actually behaves worse than either UUIDv4 or UUIDv7 alone. However, if we now reindex, and insert more values, the mixed table will perform better than UUIDv4:

```
REINDEX INDEX test_mix_id_idx;
REINDEX INDEX test2_id_idx;

INSERT INTO test_mix SELECT FROM generate_series(0, 5_000_000);
INSERT INTO test2 SELECT FROM generate_series(0, 5_000_000);

postgres=# \di+ test_mix_id_idx
...
 public | test_mix_id_idx | index | postgres | test_mix | permanent   | btree          | 572 MB |

postgres=# \di+ test2_id_idx
                                                                  ...
 public | test2_id_idx | index | postgres | test2 | permanent   | btree          | 602 MB |
```

Note how the index for the table where its ⅓ UUIDv4, ⅔ UUIDv7 values is now smaller (572MB) vs larger for the one that is only UUIDv4 (602MB).

Plan ID / pg_stat_plans

Martin v. O.
Should I log the plan_id, when log_min_duration triggers?

Currently there is no mechanism to log the Plan ID in Postgres (like exists for the Query ID), so you cannot have this logged yet in Postgres 18. This may be added in future releases, especially if the Plan ID generation is done in-core (as I proposed for 18, but that patch was not accepted).

Balaji G.
Do you think pg_stat_plans extension will make way for us to set plan baseline in future releases?

Potentially - the Plan ID mechanism will help plan management use cases as well, although I am not aware of any open-source extensions for this yet. If you're using Amazon Aurora, this is something available through Aurora Query Plan Management (QPM).

Ebubekir B.
Do you expect any performance degradation if the plan storing enabled via pg_stat_plans extension especially in a high volume transaction system?

From our performance testing, it should work well for high volume systems (and we developed a low overhead mechanism to generate the Plan ID), but it's also early days for the extension, so I would recommend testing it on non-critical systems first.

Other questions

Sixtus A.
If I upgrade from 17 to 18, is checksum automatically enabled or it's only for new clusters?

Checksums are only enabled by default for newly provisioned clusters (new initdb run). Clusters that are upgraded with pg_upgrade will keep the existing choice in terms of checksums, as rewriting all the data pages to have checksums would be too expensive during an upgrade. If you wanted to enable checksums on an existing cluster that does not have them enabled, you would need to do a dump and restore in my understanding.

Göran S.
*Have you seen an extension for reading postgres errorlog from a table... imagine `pg_stat_errorlog` (this is *especially* good/important/enhancement for PG that are hosted by Cloud vendors)*

I have no first hand experience with it, but I saw the new "pg_stat_errors" extension released a few weeks ago, that I believe addresses that: https://github.com/fabriziomello/pg_stat_errors

As you note, extensions are not always available with managed providers, but if sufficient customers request it I've certainly seen new community developed extensions be supported.

Other questions

Joe L.
Would self join elimination offer any improvement on recursive queries?

Researching this, I am not 100% sure, but what I've found leads me to believe it's likely not supported. I would encourage you to test specific queries and reach out on the Postgres mailing lists for clarification.

Lauro O.
Do you have any info about the waited partition pruning improvement which actually allows pruning to happen with similar datatypes and with subqueries?

The release notes says it was fixed but I tested partition pruning on PG18rc1 but I saw no difference in the execution plan :(

It looks like you opened a mailing list thread after asking this question (good idea!), and this was well answered by David Rowley on the pgsql-performance mailinglist: https://www.postgresql.org/message-id/flat/CAApHDvrydoBmh5MJQUMpvqaxNOioB2U-jTGahQevYjT_YFfDCw%40mail.gmail.com#0fea41664e4af9905f4c2e3db0deccab