04 DECEMBER 2025

A deeper look at old UUIDv4 vs new UUIDv7 in PostgreSQL 18

In the past there have been many discussions about using UUID as a primary key in PostgreSQL. For some applications, even a BIGINT column does not have sufficient range: it is a signed 8-byte integer with range −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. Although these values look big enough, if we think about web services that collect billions or more records daily, this number becomes less impressive. Simple integer values can also cause conflicts of values in distributed system, in Data Lakehouses when combining data from multiple source databases etc.

However, the main practical problem with UUIDv4 as a primary key in PostgreSQL was not lack of range, but the complete randomness of the values. This randomness causes frequent B-tree page splits, a highly fragmented primary key index, and therefore a lot of random disk I/O. There have already been many articles and conference talks describing this problem. What many of these resources did not do, however, was dive deep into the on-disk structures. That's what I wanted to explore here.

What are UUIDs

UUID (Universally Unique Identifier) is a 16-byte integer value (128 bits), which has 2^128 possible combinations (approximately 3.4 × 10^38). This range is so large that, for most applications, the probability of a duplicate UUID is practically zero. Wikipedia shows a calculation demonstrating that the probability to find a duplicate within 103 trillion version-4 UUIDs is about one in a billion. Another often-quoted rule of thumb is that to get a 50% chance of one collision, you'd have to generate roughly 1 billion UUIDs every second for about 86 years.

Values are usually represented as a 36-character string with hexadecimal digits and hyphens, for example: **f47ac10b-58cc-4372-a567-0e02b2c3d479**. The canonical layout is 8-4-4-4-12 characters. The first character in the third block and the first character in the fourth block have special meaning: **xxxxxxxx-xxxx-Vxxx-Wxxx-xxxxxxxxxxxx** – **V** marks UUID version (4 for UUIDv4, 7 for UUIDv7, etc.), **W** encodes the variant in its upper 2 or 3 bits (the layout family of the UUID).

Until PostgreSQL 18, the common way to generate UUIDs in PostgreSQL was to use version-4 (for example via gen_random_uuid() or uuid_generate_v4() from extensions). PostgreSQL 18 introduces native support for the new time-ordered UUIDv7 via uuidv7() function, and also adds uuidv4() as a built-in alias for older gen_random_uuid() function. UUID version 4 is generated completely randomly (except for the fixed version and variant bits), so there is no inherent sequence in the values. UUID version 7 generates values that are time-ordered, because the first 48 bits contain a big-endian Unix epoch timestamp with roughly millisecond granularity, followed by additional sub-millisecond bits and randomness.

Test setup in PostgreSQL 18

I will show concrete results using a simple test setup – 2 different tables with column "id" containing generated UUID value (either v4 or v7), used as primary key, column "ord" with sequentially generated bigint, preserving the row creation order.

```
-- UUIDv4 (completely random keys)
CREATE TABLE uuidv4_demo (
    id uuid PRIMARY KEY DEFAULT uuidv4(), -- alias of gen_random_uuid()
    ord bigint GENERATED ALWAYS AS IDENTITY
);

-- UUIDv7 (time-ordered keys)
CREATE TABLE uuidv7_demo (
    id uuid PRIMARY KEY DEFAULT uuidv7(),
    ord bigint GENERATED ALWAYS AS IDENTITY
);

-- 1M rows with UUIDv4
INSERT INTO uuidv4_demo (id) SELECT uuidv4() FROM generate_series(1, 1000000);

-- 1M rows with UUIDv7
INSERT INTO uuidv7_demo (id) SELECT uuidv7() FROM generate_series(1, 1000000);

VACUUM ANALYZE uuidv4_demo;
VACUUM ANALYZE uuidv7_demo;
```

Query-level performance: EXPLAIN ANALYZE

As the first step, let's compare the costs of ordering by UUID for the two tables:

```
-- UUIDv4
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM uuidv4_demo ORDER BY id;

Index Scan using uuidv4_demo_pkey on uuidv4_demo (cost=0.42..60024.31 rows=1000000 width=24) (actual
time=0.031..301.163 rows=1000000.00 loops=1)
  Index Searches: 1
  Buffers: shared hit=1004700 read=30
Planning Time: 0.109 ms
Execution Time: 318.005 ms

-- UUIDv7
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM uuidv7_demo ORDER BY id;

Index Scan using uuidv7_demo_pkey on uuidv7_demo (cost=0.42..36785.43 rows=1000000 width=24) (actual
time=0.013..96.177 rows=1000000.00 loops=1)
  Index Searches: 1
  Buffers: shared hit=2821 read=7383
Planning Time: 0.040 ms
Execution Time: 113.305 ms
```

The exact buffer numbers depend on caching effects, but one thing is clear in this run: the index scan over UUIDv7 needs roughly 100 times less buffer hits and is around three times faster (113 ms vs 318 ms) for the same million-row ORDER BY id. This is the first sign that UUIDv7 is a very viable solution for a primary key when we need to replace a BIGINT column with something that has a much larger space and uniqueness, while still behaving like a sequential key from the point of view of the index.

Speed of Inserts – simple benchmarking

Originally I wanted to make more sophisticated tests, but even very basic naive benchmark showed huge difference in speed of inserts. I compared time taken to insert 50 million rows into empty table, then again, into the table with 50 million existing rows.

```
INSERT INTO uuidv4_demo (id) SELECT uuidv4() FROM generate_series(1, 50000000);
INSERT INTO uuidv7_demo (id) SELECT uuidv7() FROM generate_series(1, 50000000);

-- UUID v4                                 -- UUID v7
                              Empty table
Insert time: 1239839.702 ms (20:39.840)    Insert time: 106343.314 ms (01:46.343)
Table size: 2489 MB                        Table size: 2489 MB
Index size: 1981 MB                        Index size: 1504 MB

                          Table with 50M rows
Insert time: 2776880.790 ms (46:16.881)    Insert time: 100354.087 ms (01:40.354)
Table size: 4978 MB                        Table size: 4978 MB
Index size: 3956 MB                        Index size: 3008 MB
```

As we can see, speed of inserts is radically different. Insertion of the first 50 million rows into empty table took only 1:46 minutes for UUIDv7, but already 20 minutes for UUIDv4. Second batch showed even 2 times bigger difference.

How values are distributed in the table

These results indicate huge differences in indexes. So let's analyze it. As next we will check how the values are distributed in the table, I use the following query for both tables (just switching the table name):

```
SELECT
    row_number() OVER () AS seq_in_uuid_order,
    id,
    ord,
    ctid
FROM uuidv4_demo
ORDER BY id
LIMIT 20;
```

Column seq_in_uuid_order is just the row number in UUID order, ord is the insertion order, ctid shows the physical location of each tuple in the heap: (block_number, offset_in_block).

UUIDv4: random UUID order ⇒ random heap access

How do the results look for UUIDv4?

```
 seq_in_uuid_order |                  id                  |  ord   |    ctid
-------------------+--------------------------------------+--------+------------
                 1 | 00000abf-cc8e-4cb2-a91a-701a3c96bd36 | 673969 | (4292,125)
                 2 | 00001827-16fe-4aee-9bce-d30ca49ceb1d | 477118 | (3038,152)
                 3 | 00001a84-6d30-492f-866d-72c3b4e1edff | 815025 | (5191,38)
                 4 | 00002759-21d1-4889-9874-4a0099c75286 | 879671 | (5602,157)
                 5 | 00002b44-b1b5-473f-b63f-7554fa88018d | 729197 | (4644,89)
                 6 | 00002ceb-5332-44f4-a83b-fb8e9ba73599 | 797950 | (5082,76)
                 7 | 000040e2-f6ac-4b5e-870a-63ab04a5fa39 | 160314 | (1021,17)
                 8 | 000053d7-8450-4255-b320-fee8d6246c5b | 369644 | (2354,66)
                 9 | 00009c78-6eac-4210-baa9-45b835749838 | 463430 | (2951,123)
                10 | 0000a118-f98e-4e4a-acb3-392006bcabb8 |  96325 | (613,84)
                11 | 0000be99-344b-4529-aa4c-579104439b38 | 454804 | (2896,132)
                12 | 00010300-fcc1-4ec4-ae16-110f93023068 |  52423 | (333,142)
                13 | 00010c33-a4c9-4612-ba9a-6c5612fe44e6 |  82935 | (528,39)
                14 | 00011fa2-32ce-4ee0-904a-13991d451934 | 988370 | (6295,55)
                15 | 00012920-38c7-4371-bd15-72e2996af84d | 960556 | (6118,30)
                16 | 00014240-7228-4998-87c1-e8b23b01194a |  66048 | (420,108)
                17 | 00014423-15fc-42ca-89bd-1d0acf3e5ad2 | 250698 | (1596,126)
                18 | 000160b9-a1d8-4ef0-8979-8640025c0406 | 106463 | (678,17)
                19 | 0001711a-9656-4628-9d0c-1fb40620ba41 | 920459 | (5862,125)
                20 | 000181d5-ee13-42c7-a9e7-0f2c52faeadb | 513817 | (3272,113)
```

Values are distributed completely randomly. Reading rows in UUID order practically does not make sense here and leads directly into random heap access for queries that use the primary key index.

UUIDv7: UUID order follows insertion order

On the other hand, UUIDv7 values are generated in a clear sequence:

```
 seq_in_uuid_order |                  id                  | ord | ctid
-------------------+--------------------------------------+-----+--------
                 1 | 019ad94d-0127-7aba-b9f6-18620afdea4a |   1 | (0,1)
                 2 | 019ad94d-0131-72b9-823e-89e41d1fad73 |   2 | (0,2)
                 3 | 019ad94d-0131-7384-b03d-8820be60f88e |   3 | (0,3)
                 4 | 019ad94d-0131-738b-b3c0-3f91a0b223a8 |   4 | (0,4)
                 5 | 019ad94d-0131-7391-ab84-a719ca98accf |   5 | (0,5)
                 6 | 019ad94d-0131-7396-b41d-7f9f27a179c4 |   6 | (0,6)
                 7 | 019ad94d-0131-739b-bdb3-4659aeaafbdd |   7 | (0,7)
                 8 | 019ad94d-0131-73a0-b271-7dba06512231 |   8 | (0,8)
                 9 | 019ad94d-0131-73a5-8911-5ec5d446c8a9 |   9 | (0,9)
                10 | 019ad94d-0131-73aa-a4a3-0e5c14f09374 |  10 | (0,10)
                11 | 019ad94d-0131-73af-ac4b-3710e221390e |  11 | (0,11)
                12 | 019ad94d-0131-73b4-85d6-ed575d11e9cf |  12 | (0,12)
                13 | 019ad94d-0131-73b9-b802-d5695f5bf781 |  13 | (0,13)
                14 | 019ad94d-0131-73be-bcb0-b0775dab6dd4 |  14 | (0,14)
                15 | 019ad94d-0131-73c3-9ec8-c7400b5c8983 |  15 | (0,15)
                16 | 019ad94d-0131-73c8-b067-435258087b3a |  16 | (0,16)
                17 | 019ad94d-0131-73cd-a03f-a28092604fb1 |  17 | (0,17)
                18 | 019ad94d-0131-73d3-b4d5-02516d5667b5 |  18 | (0,18)
                19 | 019ad94d-0131-73d8-9c41-86fa79f74673 |  19 | (0,19)
                20 | 019ad94d-0131-73dd-b9f1-dcd07598c35d |  20 | (0,20)
```

Here, seq_in_uuid_order, ord, and ctid all follow each other nicely – ord increases by 1 for each row, ctid moves sequentially through the first heap page, and UUIDs themselves are monotonic because of the timestamp prefix. For index scans on the primary key, this means Postgres can walk the heap in a much more sequential way than with UUIDv4.

How sequential are these values statistically?

After VACUUM ANALYZE, I ask the planner what it thinks about the correlation between id and the physical order:

```
SELECT
    tablename,
    attname,
    correlation
FROM pg_stats
WHERE tablename IN ('uuidv4_demo', 'uuidv7_demo')
AND attname = 'id'
ORDER BY tablename, attname;
```

Result:

```
   tablename | attname | correlation
-------------+---------+---------------
 uuidv4_demo |      id | -0.0024808696
 uuidv7_demo |      id |             1
```

The statistics confirm what we just saw:

- **For uuidv4_demo.id, the correlation is essentially 0 ⇒ values are random with respect to heap order.**
- **For uuidv7_demo.id, the correlation is 1 ⇒ perfect alignment between UUID order and physical row order in this test run.**

That high correlation is exactly why UUIDv7 is so attractive as a primary key for B-tree indexes.

Primary key indexes: size, leaf pages, density, fragmentation

Next I look at the primary key indexes – their size, number of leaf pages, density, and fragmentation – using pgstatindex:

```
SELECT 'uuidv4_demo_pkey' AS index_name, (pgstatindex('uuidv4_demo_pkey')).*;

        index_name | uuidv4_demo_pkey
           version | 4
        tree_level | 2
        index_size | 40026112
     root_block_no | 295
    internal_pages | 24
        leaf_pages | 4861
       empty_pages | 0
     deleted_pages | 0
  avg_leaf_density | 71
leaf_fragmentation | 49.99

SELECT 'uuidv7_demo_pkey' AS index_name, (pgstatindex('uuidv7_demo_pkey')).*;

        index_name | uuidv7_demo_pkey
           version | 4
        tree_level | 2
        index_size | 31563776
     root_block_no | 295
    internal_pages | 20
        leaf_pages | 3832
       empty_pages | 0
     deleted_pages | 0
  avg_leaf_density | 89.98      -- i.e. standard 90% fillfactor
leaf_fragmentation | 0
```

We can immediately see that the primary key index on UUIDv4 is about 26–27% bigger:

- index_size is ~40 MB vs ~31.6 MB
- leaf_pages are 4861 vs 3832 (again about 26–27% more)
- leaf pages in the v4 index have lower average density (71 vs ~90)
- leaf_fragmentation for v4 is about 50%, while for v7 it is 0

So UUIDv4 forces the B-tree to allocate more pages and keep them less full, and it fragments the leaf level much more.

Deeper index analysis with bt_multi_page_stats

To go deeper, I examined the B-tree indexes page by page and built some statistics. I used the following query for both indexes (just changing the index name in the CTE). The query calculates the minimum, maximum, and average number of tuples per index leaf page, and also checks how sequentially leaf pages are stored in the index file:

```
WITH leaf AS (
    SELECT *
    FROM bt_multi_page_stats('uuidv4_demo_pkey', 1, -1) -- from block 1 to end
    WHERE type = 'l'
)
SELECT
    count(*) AS leaf_pages,
    min(blkno) AS first_leaf_blk,
    max(blkno) AS last_leaf_blk,
    max(blkno) - min(blkno) + 1 AS leaf_span,
    round( count(*)::numeric / (max(blkno) - min(blkno) + 1), 3) AS leaf_density_by_span,
    min(live_items) AS min_tuples_per_page,
    max(live_items) AS max_tuples_per_page,
    avg(live_items)::numeric(10,2) AS avg_tuples_per_page,
    sum(CASE WHEN btpo_next = blkno + 1 THEN 1 ELSE 0 END) AS contiguous_links,
    sum(CASE WHEN btpo_next <> 0 AND btpo_next <> blkno + 1 THEN 1 ELSE 0 END) AS non_contiguous_links
FROM leaf;
```

Results for UUIDv4:

```
-- uuidv4_demo_pkey
          leaf_pages | 4861
      first_leaf_blk | 1
       last_leaf_blk | 4885
           leaf_span | 4885
leaf_density_by_span | 0.995
 min_tuples_per_page | 146
 max_tuples_per_page | 291
 avg_tuples_per_page | 206.72
    contiguous_links | 0
non_contiguous_links | 4860
```

Results for UUIDv7:

```
-- uuidv7_demo_pkey
          leaf_pages | 3832
      first_leaf_blk | 1
       last_leaf_blk | 3852
           leaf_span | 3852
leaf_density_by_span | 0.995
 min_tuples_per_page | 109
 max_tuples_per_page | 262
 avg_tuples_per_page | 261.96
    contiguous_links | 3812
non_contiguous_links | 19
```

As we can see- the UUIDv4 index has more leaf pages, spread over a larger span of blocks, and although it has higher minimum and maximum tuples per page, its average number of tuples per leaf page (206.72) is significantly lower than for UUIDv7 (261.96).

But these numbers can obscure the whole pictures. So, let’s look at histograms visualizing count of tuples in leaf pages. For this I will use following query with buckets between 100 and 300 and will list only non empty results:

```
WITH leaf AS (
    SELECT live_items
    FROM bt_multi_page_stats('uuidv4_demo_pkey', 1, -1)
    WHERE type = 'l'
),
buckets AS (
    -- bucket lower bounds: 100, 110, ..., 290
    SELECT generate_series(100, 290, 10) AS bucket_min
)
SELECT
    b.bucket_min AS bucket_from,
    b.bucket_min + 9 AS bucket_to,
    COUNT(l.live_items) AS page_count
FROM buckets b
LEFT JOIN leaf l
    ON l.live_items BETWEEN b.bucket_min AND b.bucket_min + 9
GROUP BY b.bucket_min HAVING count(l.live_items) > 0
ORDER BY b.bucket_min;
```

Result for UUIDv4:

```
 bucket_from | bucket_to | page_count
-------------+-----------+------------
         140 |       149 |        159
         150 |       159 |        435
         160 |       169 |        388
         170 |       179 |        390
         180 |       189 |        427
         190 |       199 |        466
         200 |       209 |        430
         210 |       219 |        387
         220 |       229 |        416
         230 |       239 |        293
         240 |       249 |        296
         250 |       259 |        228
         260 |       269 |        214
         270 |       279 |        171
         280 |       289 |        140
         290 |       299 |         21
```

Result for UUIDv7:

```
 bucket_from | bucket_to | page_count
-------------+-----------+------------
         100 |       109 |          1
         260 |       269 |       3831
```

There results nicely demonstrate huge fragmentation of UUIDv4 index and stable compact structure of UUIDv7 index. The lowest buckets in UUIDv4 histogram show cases of half empty leaf index pages, on the other hand pages with more than 270 tuples exceed 90% fillfactor, because PostgreSQL uses remaining free space to avoid split. In the UUIDv7 index all leaf pages except for one (the very last one in the tree) are filled up to 90% standard fillfactor.

Another important result is in the last two columns of index statistics:

- For UUIDv4: contiguous_links = 0, non_contiguous_links = 4860
- For UUIDv7: contiguous_links = 3812, non_contiguous_links = 19

btpo_next = blkno + 1 means the next leaf page in the logical B-tree order is also the next physical block. With UUIDv4, that never happens in this test – the leaf pages are completely fragmented, randomly distributed over the index structure. With UUIDv7, almost all leaf pages are contiguous, i.e. nicely follow each other.

Also, when we examine the actual content of leaf pages, we can immediately see the randomness of UUIDv4 versus the sequential behavior of UUIDv7: UUIDv4 leaf pages point to heap tuples scattered all over the table, while UUIDv7 leaf pages tend to point into tight ranges of heap pages. The result is the same pattern we saw earlier when looking at ctid directly from the table, so I won't repeat the raw dumps here.

A small gotcha: embedded timestamp in UUIDv7

There is one small gotcha with UUIDv7 values: they expose a timestamp of creation. PostgreSQL 18 exposes this explicitly via uuid_extract_timestamp():

```
SELECT
    id,
    uuid_extract_timestamp(id) AS created_at_from_uuid
FROM uuidv7_demo
ORDER BY ord
LIMIT 5;
```

Sample results:

```
                   id                 |       created_at_from_uuid
--------------------------------------+---------------------------
 019ad94d-0127-7aba-b9f6-18620afdea4a | 2025-12-01 09:44:53.799+00
 019ad94d-0131-72b9-823e-89e41d1fad73 | 2025-12-01 09:44:53.809+00
 019ad94d-0131-7384-b03d-8820be60f88e | 2025-12-01 09:44:53.809+00
 019ad94d-0131-738b-b3c0-3f91a0b223a8 | 2025-12-01 09:44:53.809+00
 019ad94d-0131-7391-ab84-a719ca98accf | 2025-12-01 09:44:53.809+00
```

If we look at the whole sequence of values, we can analyze the time deltas between record creations directly from the UUIDs, without any separate timestamp column. For some applications this could be considered a potential information leak (for example, revealing approximate creation times or request rates), while many others will most likely not care.

Summary

- UUIDs provide an enormous identifier space (128 bits, ~3.4 × 10^38 values) where the probability of collision is negligible for real-world workloads.
- Traditional UUIDv4 keys are completely random. When used as primary keys in PostgreSQL, they tend to:
  - fragment B-tree indexes
  - lower leaf page density

  - cause highly random heap access patterns and more random I/O
- UUIDv7, introduced natively in PostgreSQL 18 as uuidv7(), keeps the 128-bit space but reorders the bits so that:
  - the most significant bits contain a Unix timestamp with millisecond precision (plus sub-millisecond fraction)
  - the remaining bits stay random
- In practical tests with 1M rows per table:
  - The UUIDv7 primary key index was about 26–27% smaller, with fewer leaf pages and much higher average leaf density
  - Leaf pages in the UUIDv7 index were overwhelmingly physically contiguous, whereas the UUIDv4 leaf pages were completely fragmented
  - An ORDER BY id query over UUIDv7 was roughly three times faster in my run than the same query over UUIDv4, thanks to better index locality and more sequential heap access

The trade-off is that UUIDv7 embeds a timestamp, which might expose approximate creation times, but for most use cases this is acceptable or even useful. So, UUIDv7 significantly improves the performance and physical layout of UUID primary keys in PostgreSQL, not by abandoning randomness, but by adding a time-ordered prefix. In PostgreSQL 18, that gives us the best of both worlds: the huge identifier space and distributed generation benefits of UUIDs, with index behavior much closer to a classic sequential BIGINT primary key.

JM

ABOUT THE AUTHOR

Josef Machytka

View posts