A random primary key costs 2.8x on insert and nothing at all on a point read: a replication at 20 million rows

UUIDv4 against UUIDv7 and a bigint TSID on Postgres 16, identical rows, cold reads separated from warm ones

Dmitriy Semenkevich · ORCID 0009-0009-3013-4978

27 August 2026 · dimhold.by

ABSTRACT

The advice to swap a random UUID primary key for one ordered by time is everywhere and the published measurements behind it are single runs reported as prose. This is an independent replication with the disproof condition written before any data existed. 3 strategies load the same 20 million rows into Postgres 16 with fsync on: a random uuid, a uuid v7 whose top 48 bits are a timestamp and a bigint TSID. The random key and the v7 key share a column type, a width, an index and a payload, so the only variable between them is arrival order. Keys are laid out ahead of the timed section so the generator's own cost stays out of the measurement, strategy order alternates between passes and the server is restarted between strategies because shared_buffers is instance wide. The random key took 152.9 s against 54.5 s, wrote 1.32 times the WAL and ended 1.27 times larger at 49.8% leaf fragmentation against 0% on identical data. It does not start slow: the first 5 batches run at 1026 ms and the last 5 at 2496 ms, while the ordered key is flat from 681 ms to 665 ms. Point lookups show no difference at all. A 762 MiB index at 49.8% fragmentation answers in 236 us cold, the same as a 428 MiB index at zero, reading 1.07 index blocks per lookup against 1.04. Sizes are byte counts printed in powers of 1024.

1. The question

A random primary key is convenient. It can be generated anywhere, it does not leak the size of the table and it never needs a round trip to the database. The standing counterargument is that it is slower and the standing reply is that the difference is pennies.

Neither side usually brings 3 numbers: insert time, index size, WAL volume. This run brings those, plus a fourth that almost nobody measures, which is what the same damage

costs on a read.

This is a replication, not a discovery. The result that a random UUID degrades a Postgres B-tree and that a UUID ordered by time does not has been published and section 6 names who published it first. What this run adds is method: a disproof condition written before counting, keys generated so that a rerun produces the same keys, strategy order alternated between passes, cold reads separated from warm ones and every batch timing shipped as machine readable data rather than summarised in a paragraph.

2. Method

2.1 3 strategies, 1 variable

	KEY	HOW IT IS PRODUCED
R	`uuid`	128 bits spread uniformly across the key space, arriving in no order
T	`uuid`	RFC 9562 v7: the top 48 bits are milliseconds since the epoch, so values arrive strictly ascending
B	`bigint`	`(epoch_ms << 22) | counter`, 8 bytes, ascending

R and T use the same column type, the same 16 bytes, the same index and the same row payload. That is the point of the design. Comparing a `uuid` against a `bigint` would measure key width rather than key order, so the width advantage gets its own strategy and is reported separately instead of being blended into the ordering result.

2.2 Rules fixed before the first number

`CRITERIA.md` was committed before any data existed. The rules that shape the measurement:

1. **The payload is identical.** Row data is generated once and served to every strategy. Only the keys differ.
2. **Keys are laid out before the timed section.** `gen_random_uuid()` and a v7 generator cost different amounts of CPU and that difference would land inside an insert

measurement that is about index behaviour. All keys are written into a source table first, derived from the row number, so a rerun produces the same keys.

3. `fsync` **stays on**, along with `synchronous_commit` and `full_page_writes`. Turning any of them off removes exactly the cost under discussion.
4. **Strategy order alternates between passes.** The first load on a cold database is always faster. A fixed order would measure run order.
5. **The server is restarted between strategies.** `shared_buffers` is instance wide, so 1 database per strategy isolates the catalog and not the buffer cache. A `CHECKPOINT` precedes each measurement so that background writing from the previous strategy does not land inside the next one.
6. `shared_buffers` **is deliberately too small.** 256 MB was chosen before counting and is smaller than the final random index. A run where everything fits in cache measures memory rather than the key.

The disproof condition was written down in the same file: **under 1.5x on all 3 of last batch insert time, last batch WAL and end index size would have been the published result**, with the conclusion that at this table size key order decides nothing and the advice was sold for more than it is worth.

Postgres 16.14, `shared_buffers=256MB`, `max_wal_size=4GB`, `checkpoint_timeout=30min`, `work_mem=64MB`, `maintenance_work_mem=256MB`. The full server settings are recorded inside every row of the insert output, not only in the write up. The read rows carry the index statistics and the buffer counters but no settings block, which is a gap in the artifact rather than in the run.

Units. The harness records byte counts and this paper prints them in powers of 1024, labelled MiB and GiB. The 762 MiB index is 799,064,064 bytes, which a tool counting in powers of 1000 would call 799 MB. Figures quoted from other people's write ups keep whatever units they used. A Postgres setting such as `shared_buffers=256MB` is the string the server takes rather than a measurement.

Dates. The insert passes ran on 26 August 2026 and every batch row carries its own timestamp. The read passes ran on 27 August 2026 against the tables the insert passes left behind.

2.3 What is counted

Time per batch, kept per batch rather than averaged, so that degradation as the table grows is visible. WAL volume as the difference of `pg_current_wal_lsn()` around each batch. `pg_relation_size` for the index and the table. `avg_leaf_density` and `leaf_fragmentation` from `pgstatindex`.

2.4 The read measurement and the debt that produced it

The first attempt at reads was wrong and was published as a debt rather than as a number. It joined 2,000 known keys against the table in 1 query, which left Postgres free to plan a bulk scan instead of 2,000 independent index lookups. That run is still in the artifact, flagged as broken.

`read.mjs` is that debt paid, against the same 20 million row tables so that the read numbers sit beside the write numbers. Every key is looked up by its own query inside plpgsql and timed individually. The keys are materialised before the timed section. The server is restarted so that `shared_buffers` is empty, a cold series runs, then a warm series over the same keys. Buffer reads are counted next to the clock, because a cache hit and a disk read differ by 2 orders of magnitude and must never be averaged together. Percentiles are `percentile_cont`, which interpolates between the 2 neighbouring observations rather than picking a stored one.

3. Results

At 20 million rows the random key took 152.9 s against 54.5 s for the key ordered by time, wrote 1.32 times the WAL and finished 1.27 times larger with 49.8% leaf fragmentation against 0% on identical data.

	TOTAL INSERT	FIRST 5 BATCHES	LAST 5 BATCHES	WAL	INDEX	LEAF FRAGMENTATION	LEAF DENSITY
R random uuid	**152.9 s**	1026 ms	**2496 ms**	5.23 GiB	762 MiB	**49.8%**	71.2%
T uuid v7	54.5 s	681 ms	665 ms	3.97 GiB	602 MiB	0%	90.0%
B bigint TSID	51.2 s	627 ms	640 ms	3.67 GiB	428 MiB	0%	90.1%

80 batches of 250,000 rows. **The table prints run totals for insert time and WAL while the preregistered threshold names something narrower.** `CRITERIA.md` asks for at least 1.5x on 1 of 3 quantities: insert time on the last batch, WAL on the last batch and index size at the end. On those 3 the random key comes in at 3.73x, 1.18x and 1.27x, so insert time clears the bar by a wide margin and the other 2 do not clear it at all. On the totals the same 3 read 2.80x, 1.32x and 1.27x. The criterion passes either way and the 2 sets of numbers are printed side by side rather than swapped for each other, because a total is what a person filling a table pays while the last batch is what the registration asked for.

The shape matters more than the ratio. The random key does not start slow, it becomes slow. 1026 ms across the first 5 batches, 2496 ms across the last 5, a decay of 2.4x inside a single load. The ordered key is flat end to end, 681 ms at the start against 665 ms at the finish. Anyone extrapolating from a small table will understate the cost, because at a small table there is barely a cost to see.

3.1 The same run at 3 million rows, in both orders

PASS	ORDER		INSERT	WAL	INDEX	FRAGMENTATION
1	R, T, B	R	14.7 s	0.53 GiB	122 MiB	49.82%
1		T	7.8 s	0.46 GiB	90 MiB	0%
1		B	6.9 s	0.40 GiB	64 MiB	0%
2	B, T, R	B	6.6 s	0.41 GiB	64 MiB	0%
2		T	7.6 s	0.47 GiB	90 MiB	0%
2		R	13.4 s	0.52 GiB	122 MiB	49.82%

Reversing the order moves the timings by a few percent and moves nothing else. Index sizes and `pgstatindex` output are identical to the decimal between the 2 passes, which is what the deterministic keys are for.

The same 3 million row pass also ran on a second machine, Postgres 16 in Docker on Windows, where R took 25.9 s and 28.7 s against 13.1 s and 10.5 s for T. Different absolute numbers, the same ratio band, identical index sizes and identical fragmentation.

3.2 The one number the rest follows from

49.8% leaf fragmentation against 0%, on identical data.

A random key arrives in a leaf page that is already full. The page splits and half of each split page stays empty, which gives 71.2% leaf density against 90.0%, which gives an index a quarter larger, which means more pages touched per insert and more full page images written to WAL. An ordered key always lands at the right edge of the tree, so nothing splits in the middle and nothing fragments.

3.3 Reads: the fragmentation costs nothing

4,999 lookups per strategy, 2 passes in opposite order, against the same 20 million row tables.

	COLD P50	COLD P90	COLD P99	WARM P50	INDEX BLOCKS READ	TREE LEVELS	INDEX SIZE	LEAF FRAGMENTATION
R random uuid	236 / 254 us	427 / 462	611 / 700	6 / 7 us	5,340	3	762 MiB	**49.8%**
T uuid v7	236 / 247 us	429 / 350	1193 / 556	5 / 5 us	5,379	3	602 MiB	0%
B bigint TSID	242 / 223 us	415 / 313	563 / 527	5 / 5 us	5,193	2	428 MiB	0%

There is no difference. A 762 MiB index at 49.8% leaf fragmentation answers a point lookup as fast as a 428 MiB index at zero fragmentation and it reads essentially the same number of blocks doing it: 1.07 index blocks per lookup against 1.04.

The cold and warm split deserves its own line. 236 us against 5 us is a factor of 47, so a read benchmark that does not say which of the 2 it measured has not said anything.

4. Where the damage is paid

The result is narrower than the advice usually given: **a random primary key costs you on write, on WAL and on disk and costs you nothing on a point read.**

The null on reads is not a weak probe failing to find an effect. It is what a B-tree does. A lookup by key descends a fixed number of levels and fragmentation changes how leaves are packed and how they are ordered on disk rather than how deep the tree is. The `bigint` index is a whole level shallower than either uuid index here, 2 levels against 3, yet it is still not faster on a point lookup. Density and ordering are paid back in range scans and on the write path, which is exactly where the 2.8x sits.

The practical consequence for a schema decision is that the write cost is real and compounding while the read cost is absent, so the trade is between insert throughput and

whatever the random key was bought for. A random key hides the neighbouring record from anyone who can guess an identifier. That is a genuine advantage and this measurement does not weigh it at all.

5. Threats to validity

The operating system page cache is not cleared between strategies. Restarting the server clears `shared_buffers`, but the host cache stays warm for all 3 strategies equally, so the absolute times are optimistic across the board while the comparison between strategies survives.

This is 1 Postgres version and 1 B-tree implementation. Page splits are how this index works rather than a property of random keys in general.

The read series is 4,999 lookups per strategy per pass over existing keys. It does not cover range scans, which is where ordering and density should pay off and where this measurement predicts a gap it did not try to find.

The 20 million row figures come from a single pass in a single order. The alternating order check was run at 3 million rows, where it moved the timings by a few percent and left the index statistics identical, so the direction is not an artefact of run order. The exact 20 million row timings are 1 observation each.

Building the source table with a single `INSERT ... generate_series` over 20 million rows crashed the backend and filled a disk on the first attempt. The table is now filled in chunks of 1 million rows. That is outside every timed section and it is recorded because the first attempt at this scale produced no data at all.

6. Prior work

The UUIDv4 against UUIDv7 comparison on Postgres is well covered and the central finding here was published before this run. This section was written after the measurement rather than before it, which is the wrong order and it is recorded as such in the criteria file rather than presented as though the check had come first.

- **Josef Machytka,** ***A deeper look at old UUIDv4 vs new UUIDv7 in PostgreSQL 18*** (credativ.de, 5 December 2025) is the closest work and it published this study's headline 9 months earlier. On PostgreSQL 18 at 1 million rows per table it gives `avg_leaf_density` of 71 for v4 against 89.98 for v7, `leaf_fragmentation` of 49.99 against 0 and an index of about 40 MB with 4,861 leaf pages against 31.6 MB with 3,832. It also counts 0 contiguous leaf links for v4 against 3,812 for v7. On inserts of 50 million rows it reports 20 minutes 39 seconds for v4 against 1 minute 46 seconds for v7 into an empty table, then 46 minutes 17 seconds against 1 minute 40 seconds into a table already holding 50 million rows. The density and fragmentation figures here, 71.2 against 90.0 and 49.8% against 0%, reproduce that on a different major version at 20 times the row count.
- **Umang Sinha,** ***Benchmarking Random (v4) and Time-based (v7) UUIDs*** (dev.to, 23 May 2025) loads 10 million rows in batches of 10,000 and reports v7 inserting about 34.8% faster, with an index of roughly 793 MB against 619 MB, a reduction of about 22%. On point lookups it reports v7 with significantly lower planning and execution times than v4. **That is the one place where this replication disagrees with its neighbour. The asymmetry is the reason the disagreement is worth naming: n = 1 against 4,999 cold lookups per strategy per pass.** The point lookup comparison there is a single pair of `EXPLAIN ANALYZE` statements typed into pgAdmin, 1 query per version, with no repeats and no percentiles. The reported figures are planning 0.316 ms and execution 0.167 ms for v4 against planning 0.068 ms and execution 0.038 ms for v7. An execution time of 0.038 ms is a buffer cache hit rather than a read from disk. Under per query timing with the buffer cache emptied and cold reads separated from warm ones, the 2 strategies are indistinguishable at every percentile measured here.
  The planning half of that gap cannot be caused by the thing it is offered as evidence for. Planning happens before the index is descended, so leaf fragmentation cannot reach it: 0.316 ms against 0.068 ms, a factor of 4.6, is the cost of the first statement in a session against a later one. A lookup measured through `EXPLAIN` timings on a warm instance and a lookup measured cold, one query at a time, are different measurements.
- **Kakolaki,** ***A Comparative Analysis of Identifier Schemes: UUIDv4, UUIDv7 and ULID for Distributed Systems*** (arXiv:2509.08969, September 2025) compares the same identifier families on collision probability, generation speed and

network transmission overhead. It is the academic neighbour and it measures a different axis: nothing about B-tree behaviour or index size.

- **equenum/postgre_uuid_performance** and **mikeblum/pg-uuidv7-benchmark** are existing public harnesses for the same comparison, created in 2023 and 2024 respectively.
- **RFC 9562** defines UUID version 7 and the timestamp layout used by strategy T here.

What is left that belongs to this run is execution rather than discovery: keys generated deterministically from the row number so that the generator's cost stays out of the timed section and a rerun produces the same keys; buffer accounting alongside the clock so that a cache hit and a disk read are never averaged; cold and warm series reported separately, a factor of 47 apart; and a disproof condition written down before counting. Searched arXiv, Semantic Scholar and GitHub on 29 August 2026. Semantic Scholar rate limited most queries and general web search was unavailable that day, so the open web outside those sources is not claimed as checked. The general web search that surfaced the 2 blog posts above was run on 27 August 2026, after the insert passes had already produced their numbers rather than before them.

7. Availability

Every insert pass writes 1 JSON line per strategy carrying every batch timing, every WAL delta, the `pgstatindex` output, the sizes and the full server settings that produced them. The read passes write cold and warm percentiles with their per phase buffer deltas, the same `pgstatindex` output and the sizes, but no settings block. Both machines are in the repository, along with the harness and the criteria file with its dated amendments. The archived release carries the DOI above.

8. References

1. Josef Machytka. A deeper look at old UUIDv4 vs new UUIDv7 in PostgreSQL 18. credativ blog, 5 December 2025. https://www.credativ.de/en/blog/postgresql-en/a-deeper-look-at-old-uuidv4-vs-new-uuidv7-in-postgresql-18/

2. Umang Sinha. PostgreSQL UUID performance: benchmarking random (v4) and time based (v7) UUIDs. dev.to, 23 May 2025. https://dev.to/umangsinha12/postgresql-uuid-performance-benchmarking-random-v4-and-time-based-v7-uuids-n9b
3. Nima Karimian Kakolaki. A comparative analysis of identifier schemes: UUIDv4, UUIDv7 and ULID for distributed systems. arXiv preprint, 2025. arXiv:2509.08969. https://arxiv.org/abs/2509.08969
4. K. Davis, B. Peabody, P. Leach. Universally Unique IDentifiers (UUIDs). RFC 9562, IETF, May 2024. doi:10.17487/RFC9562. https://www.rfc-editor.org/rfc/rfc9562
5. equenum. `postgre_uuid_performance`. GitHub, 2024. https://github.com/equenum/postgre_uuid_performance
6. mikeblum. `pg-uuidv7-benchmark`. GitHub, 2023. https://github.com/mikeblum/pg-uuidv7-benchmark
7. Dmitriy Semenkevich. `key-order`: harness, preregistration and raw measurements. GitHub, 2026. https://github.com/dimhold/key-order

---

CITE THIS

Semenkevich, D. (2026). A random primary key costs 2.8x on insert and nothing at all on a point read: a replication at 20 million rows. dimhold.by. https://doi.org/10.5281/zenodo.22128828

PostgreSQL · UUID · database indexes · replication study · B-tree page splits · benchmarking · write amplification