International Journal of Computer (IJC)

ISSN 2307-4523 (Print & Online)

https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

PostgreSQL JSONB-based vs. Typed-column Indexing: A Benchmark for Read Queries

Gennadii Turutin M.S.Statistics(CUNY)[a*], Mikita Puzevich, M.S. Statistics (CUNY)[b]

[a]*Vice President of Distributed Ledger Technology, iCapital, Member of the IEEE and the American Statistical Association (ASA)*

[b]*Financial Analytics Associate, Reach Financial, Independent Researcher in Fintech*

[a]*Email: gennadii.turutin@gmail.com, Sec-ry email: gennadii.turutin@baruchmail.cuny.edu*

[b]*Email: nikita.puzevich@gmail.com, Sec-ry email: mikita.puzevich@baruchmail.cuny.edu*

Abstract

A document storage pattern is used in a wide range of business applications, particularly in systems with strong requirements for immutability and auditability. Using the JSONB data type is a common approach to achieving document storage in PostgreSQL, which supports indexing for JSONB. However, building an efficient system for such structures remains a challenge in terms of read latency, mainly due to payload size, recheck cost and value extraction. In this paper, the comparative performance of regular typed-column indexes and JSONB-based indexes is evaluated across ten queries typical of a production application. Generated metrics are used to visualize latency of indexes on JSONB expressions and typed-column indexes and for a one-sided statistical test. The findings show that JSONB-based indexes are not an optimal solution in terms of read performance, and overall typed-column indexes are expected to be at least 20% faster for tables with 1 million records or more. This benchmark is important to consider when designing an efficient data storage for documents.

I. Introduction

The advancement of blockchain systems [1], Internet of Things [2], and Artificial Intelligence (in the context of Retrieval-Augmented Generation [3]) is likely to accelerate the usage of document storage patterns.

For instance, JSONB [4] columns can prove crucial in systems with high security requirements, that operate on the append-only basis, where records are archived instead of being deleted. In such systems the most recent record is treated as the current state and archived records stay immutable and represent a chain of events that an entity went through. Preserving event timestamps ensures ordering and can be used in hash generation for encryption and other security measures. At any point in time a user should be able to see the state of an entity by fetching either historical or current records.

Received: 10/14/2025
Accepted: 12/1/2025
Published: 12/10/2025

----

* *Corresponding author.*

Example:

id	payload	created_at	archived
1	'{amount:12,status:pending}'	2025-09-22	True
2	'{amount:12,status:paid}'	2025-09-23	False

1. TOAST (The Oversized-Attribute Storage Technique) is the mechanism for handling values that do not fit in a standard page (8KB).

This design simplifies version control, and in combination with encryption makes it computationally impossible to tamper with original data.

The other advantage of JSONB is high flexibility of schema changes even if payloads have a complex nested structure. However this design comes with tradeoffs, and the read performance is worse in comparison with regular typed-column indexes. It is expected due to a larger payload size, TOASTing[1], rechecking[2], filtering[3], index evaluation and selection, as well as other types of overheads.

Despite all these additional operations, software companies often build servers working primarily with JSONB data types [5]. The focus of the study is on establishing a reproducible benchmark that quantifies this performance difference between JSONB indexes and typed-column indexes.

II. Methodology

A. Experiment design

In this study [6], a PostgreSQL (version 17) [7] database was created in a Docker container. With a seeding script the same data but in different formats was created in table inv_rel and inv_jsonb. Both tables contain indexed (prefixed ind_) and unindexed columns or typed expressions (prefixed unind_). The second table, inv_jsonb, contains only the payload JSONB column. All indexes were built on identical typed expressions.

Parallel planning and execution were turned off to exclude the effect of parallelism, since JSONB queries are more likely to trigger parallel execution due to heavy value extraction and formatting logic. In addition to that, the autovacuum process was turned off during tests to remove any potential effect on latency metrics. After the seeding stage, the vacuum process was run to remove visibility checks[4], because the goal of the study is to estimate the true costs of index searches, independent of heap visibility overhead. Queries with indexed typed columns or indexed typed expressions (JSONB) were executed with datasets of sizes (N) 1,000, 10,000, 100,000, and 1,000,000 rows, following a separate warm-up procedure to minimize caching effects and ensure the observations are independent. This experiment was repeated 30 times to obtain sufficient statistical power and reduce anomalous effects. The collected metrics were used to produce data visualizations and a formal one-sided test with a null hypothesis, stating that typed-column indexes are at least 20% faster than JSONB indexes for N=1,000,000. The p95 metric was used for visual inspection, and the per-run execution times were used for the statistical test.

2. Rechecking is the additional step of confirming the search condition PostgreSQL takes after fetching tuples from the selected indexes. It can happen if the index does not have enough information to fully verify the condition
3. Filtering is the step of confirming that all search criteria have been met which can include properties that were not used in indexes or rechecking.
4. Visibility check is the step that confirms that a record is visible to the active transaction. It is not tied to any search parameter and uses xmin and xmax to define what transactions a record is visible to.

B. Scenarios

Each scenario was designed to explore different aspects of PostgreSQL's query planner, rather than to optimize for the fastest performance in any query.

S1 – Equality + Numeric Inequality
payload->>ind_text_1=A AND payload->>ind_num_1>100

S2 – LIKE Prefix
payload->>ind_text_2 LIKE INV00012%

S3 – Substring Contains
payload->>ind_text_3 ILIKE %priority%

S4 – Timestamp Range
payload->>ind_ts_1 >= 2025-01-01T00:00:00 AND payload->>ind_ts_1 < 2025-02-01T00:00:00

S5 – Array AND (contains both)
payload->ind_text_arr_1 @> [aml,priority]

S6 – Array OR (overlap)
payload->ind_text_arr_1 @> [aml] OR payload->ind_text_arr_2 @> [priority]

S7 - Multi-key AND (2 keys)

payload->>ind_text_1 = A AND payload->>ind_bool_1 IS TRUE

S8 – Multi-key AND (3 keys)
payload->>ind_text_1 = A AND

payload->>ind_bool_1 IS TRUE AND payload->>ind_num_1 > 100

S9 – OR Across Keys

payload->>ind_text_1 = A OR payload->>ind_bool_1 IS TRUE

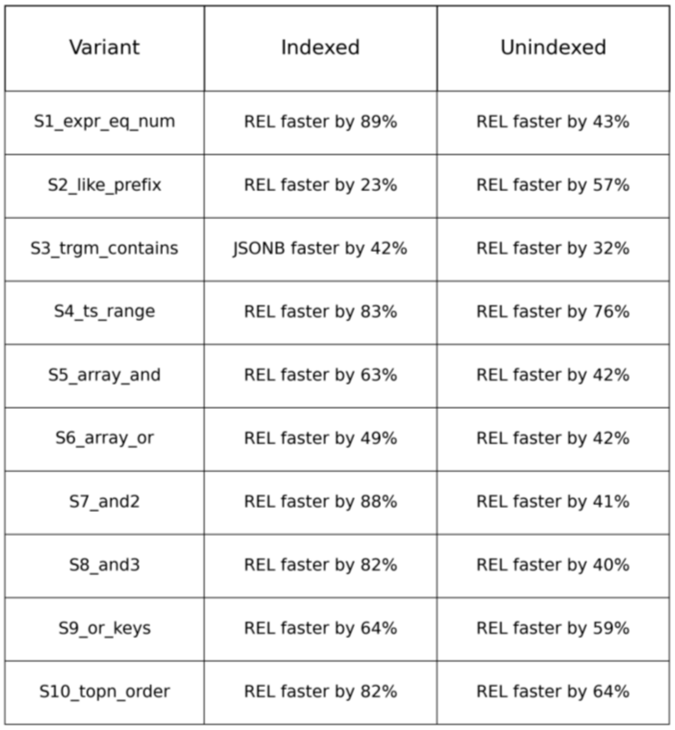
S10 – Top-N by Timestamp
payload->>ind_text_1 = A

III. Results and Discussion

A. Visual Analysis

Based on the graphs (Fig. 1) displaying p95 metrics, REL[1] indexes outperform JSONB indexes in 9/10 scenarios. Table 1 has a summary of comparisons in percentage terms which allows to set the approximate target for a formal test.

Table 1: Relative performance in p95 for different scenarios.



Variant	Indexed	Unindexed
S1_expr_eq_num	REL faster by 89%	REL faster by 43%
S2_like_prefix	REL faster by 23%	REL faster by 57%
S3_trgm_contains	JSONB faster by 42%	REL faster by 32%
S4_ts_range	REL faster by 83%	REL faster by 76%
S5_array_and	REL faster by 63%	REL faster by 42%
S6_array_or	REL faster by 49%	REL faster by 42%
S7_and2	REL faster by 88%	REL faster by 41%
S8_and3	REL faster by 82%	REL faster by 40%
S9_or_keys	REL faster by 64%	REL faster by 59%
S10_topn_order	REL faster by 82%	REL faster by 64%

Even though unindexed searches are not the focus of the study, they have been included in the graphs to establish the context and raise the importance of the right index structure. The difference between JSONB and REL is significant for both indexed and unindexed searches.

Introducing indexes improves the performance of a query in most cases. The graphs were generated for N=1,000,000 to achieve settings that are close to real-world applications.

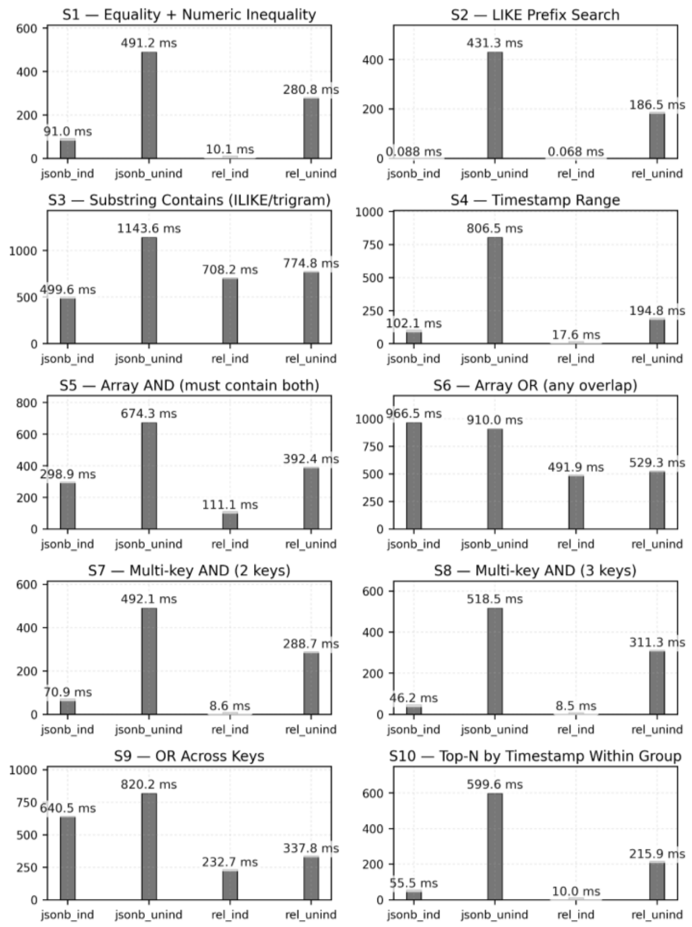


Figure 1: P95 for tables with N=1,000,000 records.

Legend:

jsonb_ind - query using ind. JSONB expression

jsonb_unind - query using unindexed JSONB expression

rel_ind - query on a typed column index

rel_unind - query on a typed column without index

B. Query Plan Analysis

The recorded query plans (N=1,000,000) provide information about execution paths and behaviour of the indexed queries for each of the scenarios. The specifics related to a particular run were removed from the plans to simplify representation and comparison. In 9/10 scenarios, p95 latency of REL indexed was substantially lower. The only exception was scenario 3, where JSONB indexed performed faster, as the REL query plan relied on a BTREE[1] index which required filtering a significantly larger number of records compared to the trigram[2] index used by JSONB.

In most queries, REL and JSONB used similar paths that relied on either Bitmap Index Scan or Index Scan. Despite the similarities in planning, the JSONB queries incurred a higher cost mostly due to scanning a greater number of heap pages to retrieve the same payload as in REL.

For each scenario a query plan for JSONB indexed and REL indexed were generated with the help of EXPLAIN ANALYZE statements:

S1 – Equality + Numeric Inequality

JSONB
Bitmap Index Scan on text_1_trgm → Bitmap Heap Scan

Recheck: payload->>ind_text_1 = A

Filter: payload->>ind_number_1::numeric > 100

Buffers: 38,492

REL
Index Only Scan using text_1_boolean_1_num_1

Index Cond: ind_text_1 = A AND ind_number_1>100

Buffers: 18,036

S2 – LIKE Prefix

JSONB
Index Scan on text_2_like

Range check: ~>=~ INV00012 and ~<~ INV00013

Recheck/Filter: payload->>ind_text_2 LIKE INV00012%

Buffers: 22

REL

Index Only Scan on text_2_like

Range check: ~>=~ INV00012 and ~<~ INV00013

Recheck/Filter: ind_text_2 LIKE INV00012%

Buffers: 5

S3 – Substring Contains

JSONB

Bitmap Index Scan on text_3_trgm → Bitmap Heap Scan

Recheck/Filter: payload->>ind_text_3 ILIKE %priority% Buffers: 166,874

REL

Index Only Scan using text_3_like

Filter: ind_text_3 ILIKE %priority%

Buffers: 3,777

S4 – Timestamp Range

JSONB

Bitmap Index Scan on ts_1_txt → Bitmap Heap Scan

Recheck/Filter: payload->>ind_ts_1 >= 2025-01-01T00:00:00.000Z and < 2025-02-01T00:00:00.000Z Buffers: 69,440

REL

Index Only Scan on ts_1

Index Cond: ind_ts_1 >= 2025-01-01 00:00:00 and < 2025-02-01 00:00:00 Buffers : 39,821

S5 – Array AND (contains both)

1. A BTREE index is an ordered tree data structure with sorted data. It is the default index type in PostgreSQL.
2. A trigram index is an index for fast substring searches that is based on trigrams (three-character sequences). More efficient for LIKE statements.

JSONB

Bitmap Index Scan on text_arr_1_gin → Bitmap Heap Scan
Rech./Filter: payload->ind_text_arr_1 @> [aml,priority]::jsonb
Buffers: 91,019

REL

Bitmap Index Scan on text_arr_1 → Bitmap Heap Scan
Recheck/Filter: ind_text_arr_1 @> ARRAY[aml,priority]::text[]
Buffers: 47,747

S6 – Array OR (overlap)

JSONB

Seq Scan & Filter: payload->ind_text_arr_1 @> [aml]::jsonb OR payload->ind_text_arr_2 @> [grpA]::jsonb
Buffers: 166,339

REL

Seq Scan & Filter: ind_text_arr_1 @> ARRAY[aml]::text[] OR ind_text_arr_2 @> ARRAY[grpA]::text[]
Buffers: 51,613

S7 – Multi-key AND (2 keys)

JSONB

Bitm. Ind scan on text_1_bool_1_num_1_str→Bitm. Heap Scan
Rech./Filter: payload->>ind_text_1=A AND payload->>ind_bool_1::boolean IS TRUE
Buffers: 38,692

REL

Index Only Scan using text_1_bool_1_num_1
Ind. Cond: ind_text_1=A AND ind_boolean_1= TRUE

Buffers:

1. Association does not imply causality
2. Shared hit blocks is the number of pages read from the shared buffer cache.

S8 – Multi-key AND (3 keys)

JSONB

Index Scan on text_1_bool_1_num_1_str
Index Cond: payload->>ind_text_1 = A AND payload->>ind_bool_1::boolean = TRUE AND payload->>ind_number_1::numeric > 100
Buffers: 38,698

REL
Index Only Scan using text_1_bool_1_num_1
Ind. Cond: ind_text_1=A AND ind_bool_1=TRUE AND ind_num_1 > 100
Buffers: 18,025

S9 – OR Across Keys

JSONB
Bitmap Index Scan on text_1_trgm OR Bitmap Index Scan on inv_jsonb_idx_bool_1 → BitmapOr → Bitmap Heap Scan
Recheck/Filter: payload->>ind_text_1=A OR payload->>ind_bool_1::boolean IS TRUE
Buffers: 168,179

REL
Index Only Scan using text_1_bool_1_num_1
Filter: ind_text_1=A OR ind_bool_1 IS TRUE
Buffers: 470,816

S10 – Top-N by Timestamp

JSONB
Index Scan using text_1_ts_1_txt
Index Cond: payload->>ind_text_1=A
Order: payload->>ind_ts_1
Buffers: 38,827

REL
Index Only Scan using text_1_ts_1
Index Cond: ind_text_1 = 'A'
Order: ind_ts_1
Buffers: 18,030

C. Scalability Analysis

Figure 2 shows p95 metrics for REL indexed and JSONB indexed for different sample sizes (N: 1,000; 10,000; 100,000; 1,000,000). It is expected that the JSONB key-value design increases the cost of overheads as the number of rows grows due to wider tuples, value extraction and typecasting.

The difference between REL and JSONB grows over N in absolute terms in most scenarios, however there is no consistent trend in relative terms (the dashed line in fig. 2).

There is a possible positive association[1] (no correlation coefficient calculated) between latency and shared hit blocks based on the visual inspection of Figure 3, but there are a few exceptions too.

- *S3 (substring contains)*: REL uses index-only BTREE scan and ILIKE '%…%' operator to filter to remove ~833k; it touches fewer pages but does a lot of per-tuple work. JSONB uses a trigram index to pre-filter to ~166k matches via Bitmap Index Scan → Bitmap Heap Scan, which is faster despite more heap pages.
- *S9 (OR across keys):* JSONB visits less heap pages, though it does not select the most efficient index and uses two indexes, which leads to expensive rechecking and filtering operations across a very large set of data.

No formal test for association between latency and shared hit blocks was made as retrieving data from buffers can be cheap (e.g. S9) due to the underlying structure and types and have an insignificant effect on the latency despite a high number of hits.

Considering that the row content is identical for JSONB and REL, higher shared-hit counts are mainly driven by larger payloads and access-path differences - JSONB frequently uses Bitmap Heap Scan.

D. Formal Statistical Test

In order to statistically verify the observable difference in execution time between REL indexed and JSONB indexed, we conducted a series of tests. As shown in Table 1, for most variants, REL outperformed JSONB. Therefore, we designed a test that would not only confirm that REL has lower latency than JSONB but also verify that REL is faster by a fixed percentage—in our case, 20%—compared to JSONB.

We formulated our null and alternative hypotheses: H0: $\partial < 20\%$ Ha: $\partial >= 20\%$

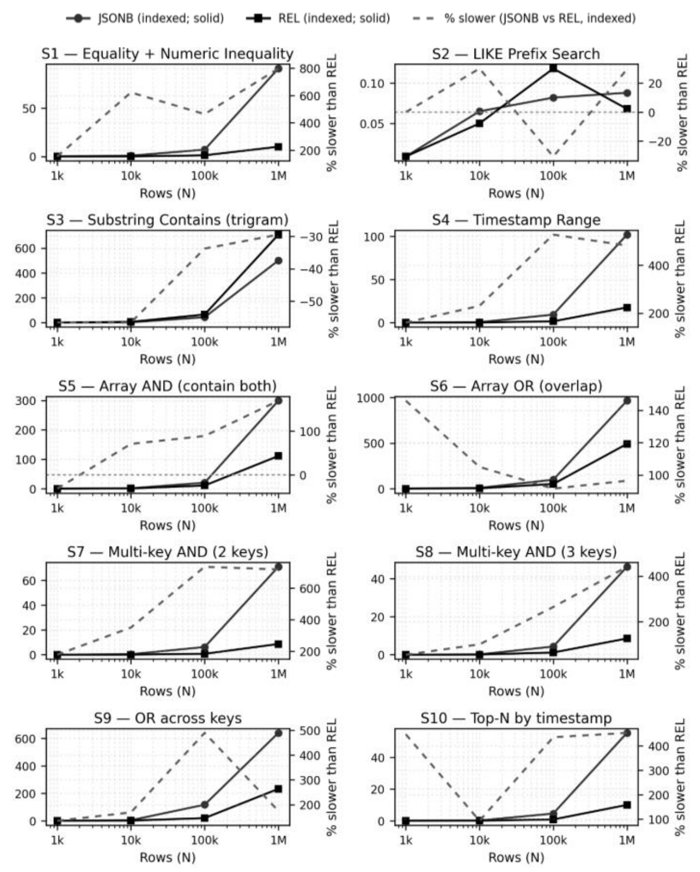


Figure 2: P95 for N 1,000; 10,000; 100,000; 1,000,000 records. X-axis was log-transformed for even spacing and better representation (N ∈ {10^3, 10^4, 10^5, 10^6}).

Where µ_jsonb – mean exec. time for JSONB and µ_rel – mean exec. time for REL ∂ is calculated as:

$$\partial = \frac{\mu_{jsonb} - \mu_{rel}}{\mu_{jsonb}}$$

To test our alternative hypothesis, we used a one-sided nonparametric bootstrap test [8]. For each query (scenario) and table, 100,000 bootstrap samples were generated by resampling execution times with replacement from the original dataset. Each bootstrap sample contained 30 observations for each table, JSONB and REL. Next, we calculated the observed relative mean difference (∂) for each bootstrap sample. Finally, we computed the p-value as the proportion of simulated values of ∂ below the threshold relative to the total number of bootstrap samples. Additionally, we calculated a 95% confidence interval for ∂ to show the variability of ∂ around its expected value. Observed difference, confidence bounds, p-value, and decision for each group are described in Table 2.

For the statistical test we use only those metrics for the indexed searches with N = 1,000,000 because we are interested in performance at scale.

As shown in Table 2, REL outperforms JSONB by at least 20% for 8 out of 10 variants. Additionally, the lower bound for some variants reaches as high as 88%, suggesting that in some scenarios the relative performance of REL is significantly better than 20%.

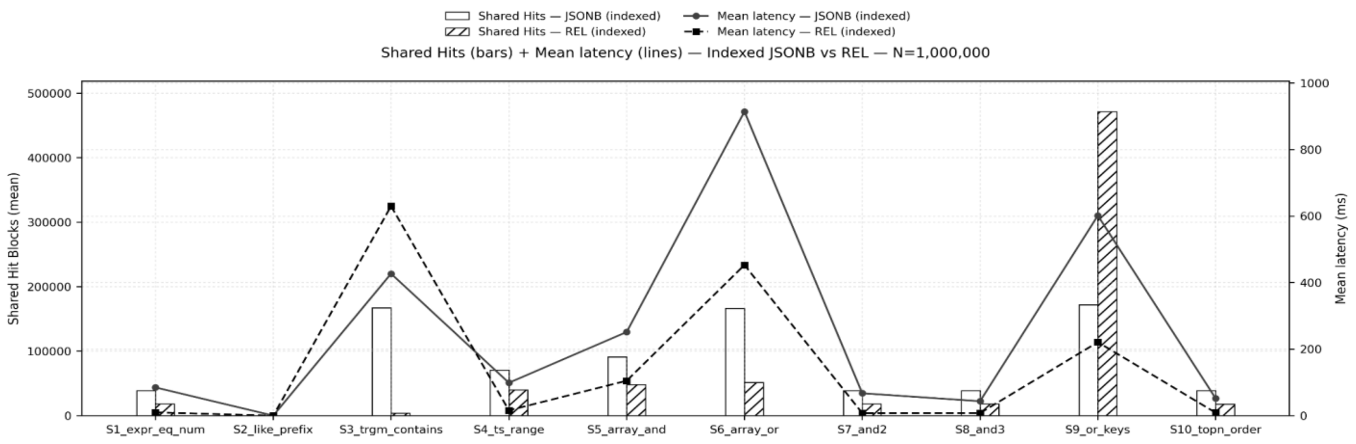
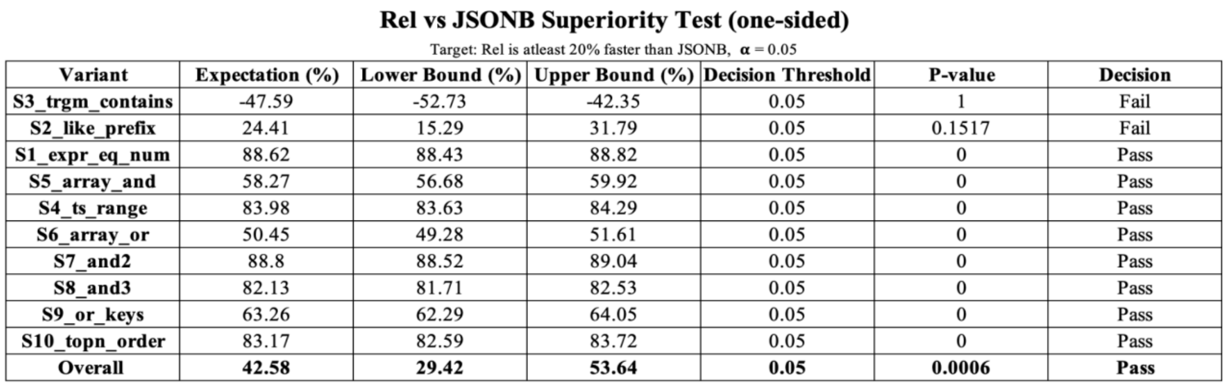


Figure 3: Graph of number of shared hit blocks for *N = 1,000,000* for each scenario.

Table 2: One-sided superiority test results at *N = 1,000,000* (Δ = 20 %, α = 0.05) showing that the indexed typed-column schema is significantly faster than JSONB across 8 scenarios.



Rel vs JSONB Superiority Test (one-sided)

Target: Rel is atleast 20% faster than JSONB, **α** = 0.05

Variant	Expectation (%)	Lower Bound (%)	Upper Bound (%)	Decision Threshold	P-value	Decision
S3_trgm_contains	-47.59	-52.73	-42.35	0.05	1	Fail
S2_like_prefix	24.41	15.29	31.79	0.05	0.1517	Fail
S1_expr_eq_num	88.62	88.43	88.82	0.05	0	Pass
S5_array_and	58.27	56.68	59.92	0.05	0	Pass
S4_ts_range	83.98	83.63	84.29	0.05	0	Pass
S6_array_or	50.45	49.28	51.61	0.05	0	Pass
S7_and2	88.8	88.52	89.04	0.05	0	Pass
S8_and3	82.13	81.71	82.53	0.05	0	Pass
S9_or_keys	63.26	62.29	64.05	0.05	0	Pass
S10_topn_order	83.17	82.59	83.72	0.05	0	Pass
Overall	**42.58**	**29.42**	**53.64**	**0.05**	**0.0006**	**Pass**

IV. Conclusion

There is consistency between the visual analysis and the formal statistical test: overall typed-column indexes deliver lower latency than JSONB.

As the number of records grows, the absolute latency gap (typed column vs JSONB) widens in most cases, but in relative terms (JSONB/typed column), there is no consistent visible trend within the tested range and across the scenarios, thus no evidence of worsening relative performance was found that can be considered a scalability issue.

JSONB often shows a higher number of shared hit blocks even when similar execution paths are chosen, which is primarily explained by larger payloads. The finding suggests that JSONB relies on buffers to a greater extent.

The conclusion of the study is that tables with indexes on typed columns are expected to systematically provide a better read performance (faster by at least 20%) for N=1,000,000 in 8/10 scenarios, though we expect this conclusion to apply to tables with the number of records greater than 1,000,000, as execution paths are expected to stay identical to the experiment.

JSONB, in its turn, has an advantage of flexibility, which can accommodate various payload structures and sizes without schema migrations, but it comes at the expense of read performance.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", https://bitcoin.org/bitcoin.pdf

[2] Miorandi, Sicari, De Pellegrini & Chlamtac, "Internet of things: Vision, applications and research challenges", *Computer Networks*, 2012 — broad taxonomy and research agenda.

[3] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", https://arxiv.org/abs/2005.11401, 2020

[4] PostgreSQL documentation, https://www.postgresql.org/docs/current/datatype-json.html

[5] Philip McMahon, Maria-Livia Chiorean, Susie Coleman, Akash Askoolum, "Bye bye Mongo, Hello Postgres", The Guardian Engineering blog, https://theguardian.engineering/blog/info-2018-nov-30-bye-bye-mongo-hello-postgres

[6] G. Turutin "JSONB-indexing-vs-rel.-indexing" in Github, https://github.com/GennadiiTurutin/JSONB-indexing-vs-rel.-indexing

[7] PostgreSQL 17, https://www.postgresql.org/about/news/postgresql-17-released-2936/

[8] Bradley Efron, R. J. Tibshirani, "An Introduction to Bootstrap," https://www.taylorfrancis.com/books/mono/10.1201/9780429246593/introduction-bootstrap-bradley-efron-tibshirani, 1993

V. Authors

Gennadii Turutin is Vice President of Distributed Ledger Technology at iCapital Network. He holds an M.S. in Statistics from the City University of New York (Baruch College) and has professional experience spanning sustainability, AI engineering, distributed ledger systems, and financial data infrastructure. His previous roles include Senior Data Engineer at TIFIN and Senior Software Engineer at Prescriptive Data (Nantum AI). His research interests include databases, AI, and the intersection of blockchain and finance.

Mikita Puzevich is a Financial Analytics Associate at Reach Financial, a lending FinTech company based in New York City. He holds an M.S. in Statistics from the City University of New York (Baruch College). His professional experience encompasses financial modeling, data analytics, and capital markets. At Reach Financial, he develops Python- and SQL-based tools for loan portfolio analysis, builds data pipelines, and designs Tableau dashboards. Earlier, he worked as a capital markets analyst, assisting in the issuance of several securitization deals and supported performance monitoring of securitizations and credit warehouses.