Postgres + Data Anonymizer

Priyadarshini R (2021CS50614), Srushti (2021CS50613), Kartik Meena (2021CS50618)
cs5210614@iitd.ac.in,cs5210613@iitd.ac.in,cs5210618@iitd.ac.in

1 Introduction

Data anonymization has moved from a niche concern about privacy to a regulatory requirement and an operational necessity. New privacy regulations like GDPR and HIPAA demand protection for personally identifiable information, or PII, while organizations test and develop analytics environments with increasing needs for masked production data without exposing sensitive records [2, 6]. Database-native anonymization solutions promise transparency in integration with existing systems; however, performance implications and privacy guarantees remain under-explored in real-world production contexts.

Traditional anonymization approaches broadly categorize into three kinds: the use of external preprocessing tools that mask data before ingesting it into a database; application-layer transformations doing masking during transport; and database-native extensions that apply masking within the database engine itself. Native database solutions, such as **postgresql_anonymizer** [1] , provide declarative masking rules embedded directly into table schemas, allowing role-based access control and query-time transformation. This integration, however, carries with it fundamental trade-offs between performance (query latency, storage overhead), privacy (re-identification risk, formal guarantees), and utility (analytical accuracy, data semantics preservation) that are not well quantified.

1.1 Research Questions

We address three key questions in evaluating database-native anonymization:

(1) **RQ1 (Performance):** What is the performance overhead of masking with extensions compared to native PostgreSQL and SQL view-based approaches across diverse query workloads?
(2) **RQ2 (Masking Functions):** How do different masking functions hashing, partial masking, shuffling, generalization, noise injection) trade off query performance, storage efficiency, and data utility?
(3) **RQ3 (Privacy):** What privacy guarantees (k-anonymity, differential privacy) follows with this extension, and at what cost to performance and utility?

1.2 Approach

We perform a comprehensive benchmark of **postgresql_anonymizer** (version 2.0.0) on PostgreSQL 15.5 using two real-world datasets: Adult Census Income (UCI/Kaggle, 32k rows) and Healthcare (Kaggle, 55k rows), scaled to 100k and 1M via sampling. We compare three systems: A) native PostgreSQL, no masking (baseline), (B) SQL view-based masking with equivalent transformation logic, and (C) the **postgresql_anonymizer** extension in both dynamic role-based query-time masking and Static: Pre-masked table copies) modes. Experiments measure query latency (median of 5 runs after 3 warmups), storage overhead, query plan characteristics via EXPLAIN ANALYZE, and utility metrics including cardinality preservation and Jensen-Shannon divergence across five workload types: point lookup, range scan, group-by aggregates, distinct count, top-k queries). For privacy analysis, we implement k-anonymity via manual generalization and suppression, simulate differential privacy with calibrated Laplace noise and re-identification quasi-identifier linkage attacks.

2 Background and System Overview

2.1 PostgreSQL Anonymizer Extension

postgresql_anonymizer [1] is an open-source PostgreSQL extension developed by Dalibo that implements declarative data masking directly within the database engine. The extension follows an *anonymization by design* philosophy, where masking rules are defined as part of the database schema using PostgreSQL's `SECURITY LABEL` mechanism. This approach enables developers to specify anonymization policies alongside table definitions, ensuring that privacy requirements are embedded in the data model itself rather than implemented as afterthoughts in application code.

The extension provides five masking modes:

(1) **Anonymous Dumps:** Export masked data to SQL files for safe distribution.
(2) **Static Masking:** In-place anonymization that replaces original data with masked values.
(3) **Dynamic Masking:** Role-based query-time transformation that shows masked data to designated users while preserving raw data for administrators.
(4) **Masking Views:** Generate SQL views that apply masking logic transparently.
(5) **Data Wrappers:** Apply masking to foreign data accessed via **postgres_fdw**.

Our benchmark focuses on dynamic masking (query-time transformation based on user roles) and static masking (pre-masked table copies), as these represent the two primary deployment patterns for production systems.

2.1.1 Masking Rule Definition. Masking rules are defined using PostgreSQL's security label system. Listing 1 shows an example where a user table's email column is masked with a fake email function:

Listing 1: Declarative masking rule definition

```
SECURITY LABEL FOR anon ON COLUMN users.email
  IS 'MASKED WITH FUNCTION anon.fake_email()';
```

For dynamic masking, roles are marked as masked, and the extension intercepts queries to apply transformations:

```
CREATE ROLE analyst LOGIN PASSWORD 'secret';
SECURITY LABEL FOR anon ON ROLE analyst IS 'MASKED';
GRANT pg_read_all_data TO analyst;
```

```
ALTER DATABASE mydb SET
  anon.transparent_dynamic_masking TO true;
```

When the `analyst` role queries the `users` table, the extension transparently applies the masking function to the `email` column, returning synthetic values instead of real email addresses.

2.2 Masking Function Taxonomy

The extension provides a rich library of masking functions that fall into six categories:

- **Hashing:** This is deterministic pseudonymization through the use of cryptographic hash functions, for example, `anon.hash(column)`. This preserves uniqueness but it destroys readability and semantic meaning.
- **Partial Masking:** This keeps prefix/suffix but masks the middle, for example, `anon.partial(phone, 2, '****', 2)` → `06****11`. This balances utility and privacy for pattern preserving anonymization.
- **Shuffling:** This assigns random permutations to column values across rows, for example, `anon.shuffle()`. It preserves distribution statistics but breaks individual level correlations.
- **Fake Data:** This produces realistic synthetic values from predefined dictionaries for example, `anon.fake_email()`, and `anon.fake_company()`. It has semantic plausibility but needs extensive lookup tables.
- **Noise Injection:** This adds calibrated random noise to numeric values, for example, `anon.noise(salary, 0.1)` adds ±10% noise). It approximates differential privacy but does not provide any formal guarantees.
- **Generalization:** This replaces values with higher-level categories, such as exact ages → 5-year buckets). It reduces granularity to achieve properties similar to k-anonymity.

We will discuss in Section 4 the performance and utility trade-off of these functions.

2.3 Comparison Systems

To isolate the overhead of the extension's masking infrastructure, we compare against two baselines:

Baseline A (Native PostgreSQL): Raw tables with no masking (`adult_raw`, `healthcare_raw`). This represents the performance upper bound with zero privacy protection.

Baseline B (SQL Views): Standard PostgreSQL views that apply equivalent masking logic using built-in functions. For example:

Listing 2: SQL view-based masking baseline

```
CREATE VIEW healthcare_masked AS
SELECT
  substring(name, 1, 2) || '***' AS name,
  (age / 5) * 5 AS age,
  encode(digest(doctor, 'sha256'), 'hex')
  AS doctor
FROM healthcare_raw;
```

This baseline allows us to measure the pure overhead of the extension's masking engine versus standard SQL transformations.

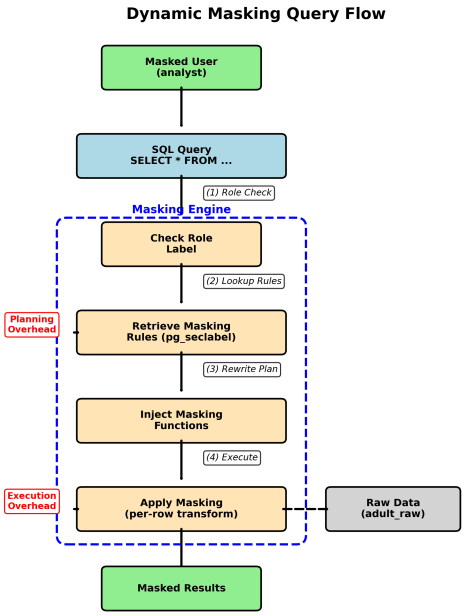

Figure 1: Dynamic Masking Query Execution Flow

System Under Test (Extension): postgresql_anonymizer in two modes: (1) dynamic masking with role `masked_user`, and (2) static masked tables (`adult_static_masked`).

2.4 Architecture and Query Flow

Figure 1 illustrates the query execution flow for dynamic masking. When a masked role executes a query: (1) the extension checks the role's security label, (2) retrieves masking rules for accessed columns from `pg_seclabel`, (3) rewrites the query plan to inject masking functions, and (4) executes the transformed query. This adds overhead in both query planning (rule lookup) and execution (function evaluation per row).

Static masking avoids query-time overhead by pre-applying masking rules to a separate table copy, trading storage space for performance. Our evaluation quantifies these trade-offs across diverse workloads and data scales.

3 Methodology

3.1 Experimental Setup

All experiments were conducted on a laptop with an Intel Core i7-1165G7 CPU (4 cores, 8 logical processors, 2.8GHz base frequency), 32GB DDR4 RAM, and a 1TB Intel NVMe SSD (Intel 660p OEM variant, PCIe 3.0×4, sequential read: 1800 MB/s). PostgreSQL 16.10 with `postgresql_anonymizer` version 2.0.0 ran inside Docker (image: `registry.gitlab.com/dalibo/postgresql_anonymizer:latest`) under Ubuntu 24.04 (WSL2 environment), using the WSL2 kernel 6.6.87.2-microsoft-standard-WSL2 on Windows.
PostgreSQL configuration:

- work_mem=256MB
- max_parallel_workers_per_gather=4

- shared_buffers=2GB
- random_page_cost=1.1 (SSD-optimized)

Extensions enabled:

- pgcrypto (for custom hash masking)
- pg_stat_statements (for aggregate query statistics)
- anon (masking engine)

Dynamic masking was enabled via `anon.transparent_dynamic_masking=on`.

3.2 Datasets

We used two real-world datasets with diverse data types and quasi-identifiers (Table 1):

Table 1: Dataset characteristics and scaling methodology.

Dataset	Rows	Cols	Size	Quasi-IDs
Adult Census (UCI/Kaggle)	32k → 1M	15	3.6 MB	age, education, sex, race
Healthcare (Kaggle)	55k → 1M	15	8.2 MB	age, gender, condition

Adult Census Income [4]: Contains demographic and employment data (age, education level, occupation, hours worked, income bracket). Includes numeric (age, capital gain/loss), categorical (workclass, marital status), and text (native country) columns. Quasi-identifiers: age, education, sex, race.

Healthcare Dataset [3]: Synthetic patient records with names, ages, medical conditions, billing amounts, doctor assignments, and admission dates. Combines PII (names, dates) with sensitive health information.

Scaling: Original datasets were scaled to 100k and 1M rows using stratified sampling with replacement (Python *pandas.sample(replace =True, random_state=42)*) to preserve approximate column distributions. This approach simulates realistic dataset sizes encountered in production systems while maintaining statistical properties.

3.3 Systems and Workloads

Systems: Four configurations per dataset:

(1) **raw**: Native PostgreSQL, no masking (`adult_raw`)
(2) **view**: SQL view-based masking (`adult_view_masked`)
(3) **static**: Pre-masked table copy (`adult_static_masked`)
(4) **dynamic**: Extension with `masked_user` role

Workloads: Five query types representing common analytical patterns:

(1) **Point Lookup:** `SELECT * FROM table WHERE id = ?` (index scan)
(2) **Range Scan:** `SELECT COUNT(*) WHERE age BETWEEN a AND b` (sequential scan with filter)
(3) **Group-By Aggregate:** `SELECT a, COUNT(*), AVG(b) FROM table GROUP BY a` (hash aggregate)
(4) **Distinct Count:** `SELECT COUNT(DISTINCT a) FROM table GROUP BY b` (nested aggregation)
(5) **Top-K Query:** `SELECT a, COUNT(*) as b FROM table GROUP BY a ORDER BY b DESC LIMIT 10` (sort + limit)

3.4 Metrics

Performance:

- **Wall-clock latency:** Measured client-side using Python `time.perf_counter()`, reported as median of 5 runs after 3 warmup runs.
- **Execution time:** Server-side execution time from `EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)`.
- **Storage:** Total relation size (table + indexes + TOAST) via `pg_total_relation_size()`.
- **Query plans:** Node types, index usage, buffer hit/read ratios, and row estimate accuracy.

Privacy:

- **k-Anonymity:** Group size analysis on quasi-identifiers (age, education, sex, race). We implemented manual generalization (age → 5-year buckets, education → 4 categories) and suppression (removing rows in groups < k). Suppression rate and query accuracy (MAE for aggregates) measured for $k \in \{2, 5, 10, 20\}$.
- **Differential Privacy:** Simulated using Laplace mechanism for `COUNT`, `SUM`, `AVG` queries. Noise scale = sensitivity / ϵ. Measured mean absolute error (MAE) and relative error for $\epsilon \in \{0.1, 0.5, 1.0, 5.0\}$.
- **Re-identification:** Linkage attack using external dataset with quasi-identifiers. Success rate = percentage of correct unique matches.

Utility:

- **Cardinality ratio:** Unique values in masked column / unique values in raw column.
- **Jensen-Shannon divergence:** Distribution similarity for categorical columns (0 = identical, 1 = completely different).

3.5 Experimental Procedure

For each (system, workload, data size) configuration:

(1) Restart PostgreSQL and load data
(2) Run warmup queries (3×) to populate buffer cache
(3) Execute measured queries (5×) and record latency
(4) Collect `EXPLAIN ANALYZE` output as JSON
(5) Compute median latency and standard deviation

All experiments used warm cache (realistic production scenario where frequently accessed data resides in memory). Random number generators were seeded (`np.random.seed(42)`, `Faker.seed(42)`, PostgreSQL `setseed(0.42)`) for reproducibility.

The full experimental setup, scripts, and benchmark code are publicly available at: **https://github.com/pdrk2303/Postgres-Data-Anonymizer**.

Experiments can be replicated with: `./run_all.sh` (see README in the Github Repository for details).

4 Results

Figure 2 compares query latency across systems for the 100k-row dataset. Dynamic masking has very high overhead compared to native PostgreSQL, with the most significant impact on range scans at (+24003.4%) and group-by aggregates at (+11013.3%) since per-row masking function evaluation occurs during scanning and aggregation. Point lookups have much lower overhead at (+1.5%) since

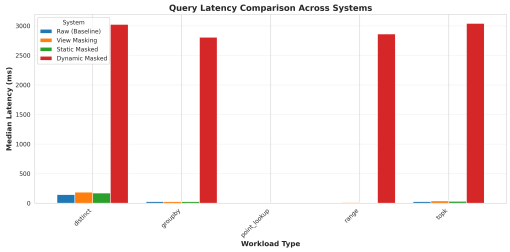

Figure 2: Query Latency Comparison Across Systems

Table 2: Query latency overhead vs. native PostgreSQL baseline (100k rows).

Workload	View (%)	Static (%)	Dynamic (%)
Point lookup	+[2.9]	+[630.7]	+[1.5]
Range scan	+[45.8]	+[27.5]	+[24003.4]
GROUP BY	+[23.8]	+[17.4]	+[11013.3]
Distinct count	+[23.9]	-[3.5]	+[1512.0]
Top-K query	+[48.3]	+[4.4]	+[9541.6]
Average	+[28.9]	+[135.3]	+[9214.4]

masking applies to fewer resultant rows. Static masking has moderate overhead across workloads with an average of +135.3%, but it increases most on point lookups at (+630.7%). Views exhibit the lowest overhead at an average of +28.9%, which indicates that the extension's query-time infrastructure adds relatively lower cost when the masking logic is pre-applied.

Table 2 quantifies the overhead across workload types. Range scans exhibit +24003.4% overhead for dynamic masking since sequential scans amplify the per-row masking costs. Views and static masking generally follow similar patterns, though static masking incurs higher overhead in point lookups (+630.7%) while views remain comparatively efficient at +2.9%.

Storage Overhead: Static masking has 2 times the storage cost: original + masked copy. For the 100k Adult dataset: raw table = **[17] MB**, static masked = **[16] MB**, total = **[33] MB**. Views take up **[0 bytes]** - virtual relation. Dynamic masking has zero storage overhead since masking occurs on-the-fly.

4.1 Masking Function Comparison

Table 3: Six masking functions on group-by aggregate latency (representative workload) Shuffle has the lowest latency (4.53 ms vs. 11.15 ms baseline, −59.4%), since random permutation is done in memory without cryptographic cost. Both hash and partial masking also enjoy great performance boost, which are 6.25 ms and 5.16 ms, respectively (−43.9% and −53.7%), since SHA256 and substring operations are efficiently computed. Fake value generation also reduces the latency to 5.01 ms (−55.1%), since it merely looks up in the anon dictionary. In contrast, Laplace-like noise injection has longer latency, 13.41 ms (+20.3%), reflecting the overhead of random noise sampling and value perturbation at query evaluation time. Generalization has a mild reduction, 9.72 ms(−12.8%), as the bucket mapping is simple and avoids computing heavy.

Table 3: Masking function characteristics (50k rows, group-by query).

Function	Latency (ms)	Storage (MB)	Cardinality Ratio	JS Divergence	Idx Compatible
Original	[11.15]	[0.00]	Nan	Nan	Nan
Hash	[6.25]	[12.92]	[1.00]	[0.83]	Yes
Partial	[5.16]	[6.92]	[0.87]	[0.81]	Yes
Shuffle	[4.53]	[7.90]	[1.00]	[0.00]	Yes
Fake	[5.01]	[15.35]	[66.60]	[0.83]	Yes
Noise	[13.41]	[7.96]	[1.00]	[0.00]	No
Generalize	[9.72]	[7.89]	[0.40]	[0.833]	No

Utility Preservation: Generalization comes with the lowest JS divergence, (**0.833**), among semantic-preserving techniques because of the reduction of fine-grained categories into wider buckets, for example, 12 levels of education to 4 groups. This, however, also yields a strong reduction in cardinality - **40%** of its original-value, and hence, it trades granularity for interpretability. Hash masking retains full cardinality **(100%)** but loses semantic meaning totally (JS divergence **0.833**). Shuffling preserves the empirical distribution (JS divergence **0.0**), maintains cardinality, but destroys row-level consistency across columns. Partial masking retains most granularity at **86.7%** cardinality with moderate divergence of **0.808**. Noise injection keeps the distribution shape (JS divergence **0.0**) but breaks numeric fidelity and thus results in utility loss for any precise aggregation workload.

Index Compatibility: `EXPLAIN` analysis indicates that hash, partial, shuffle, and fake masking maintain index utilization. In particular, hashed columns remain usable with index lookups since the hash function is deterministic. On the contrary, noise injection and generalization preclude index scans, because the perturbed or bucketed values do not match the indexed storage.

4.2 k-Anonymity

Table 4 shows suppression rates and query accuracy for various k values. A $k = 10$ can be attained by suppressing only**1.0%** of records, i.e., removing tuples that do not appear in a group of size ≥ 10 based on age-education-sex-race quasi-identifiers. This leads to an average absolute error of **410.97** in aggregate queries or **5.1%** relative error, which is a moderate utility loss for analytical workloads. Increasing privacy to $k = 20$ raises suppression to **2.2%** and results in higher error (MAE: **685.86**, relative error **6.4%**), reflecting the expected utility-privacy tradeoff. Somewhat surprisingly, query latency slightly improves for higher values of k (up to −17.4% overhead) because of the reduced table size after suppression, which showcases that, for read-heavy aggregate workloads, anonymity-driven tuple reduction can come with performance benefits.

Critique: `postgresql_anonymizer` does not provide native k-anonymity primitives. Achieving k-anonymity requires manual preprocessing (generalization + suppression scripts), separate table creation, and careful tuning of generalization hierarchies. This is a significant gap compared to specialized tools like ARX [5].

Table 4: k-Anonymity: suppression cost and aggregate accuracy.

k	Suppression (%)	MAE	Rel. Err.	Latency (ms)
2	[0.1]	[192.84]	[3.9]%	[14.08]
5	[0.4]	[263.94]	[4.3]%	[13.88]
10	[1.0]	[410.97]	[5.1]%	[17.47]
20	[2.2]	[685.86]	[6.4]%	[13.57]

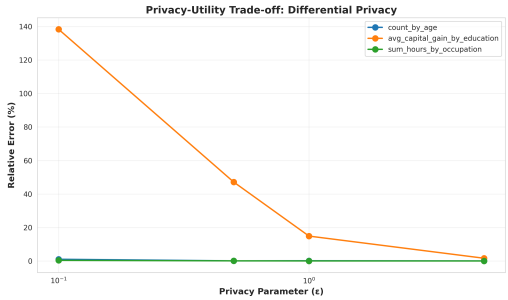

Figure 3: Privacy-Utility Trade-Off

4.3 Differential Privacy

Figure 3 plots the privacy parameter (ϵ) against relative error for aggregate queries. At $\epsilon = 0.1$ (strong privacy), Laplace noise introduces high error for the AVG query on capital gain ($\approx$ 138.3%), while COUNT and SUM remain largely unaffected ($\approx$ 1.0% and $\approx$ 0.4% respectively). Increasing ϵ to 0.5 reduces error to $\approx$ 47.2% for AVG, with minimal error for COUNT and SUM ($\approx$ 0.1% each). At $\epsilon = 1.0$, the AVG query error drops to $\approx$ 14.9%, with near-zero impact on the other aggregates ($\approx$ 0.1% and 0.0%). With $\epsilon = 5.0$, noise becomes negligible for all queries ($\approx$ 1.6% for AVG and $\approx$ 0.0% for COUNT and SUM). The trade-off is clear: strong privacy significantly affects mean estimates on skewed attributes, while count- and sum-based aggregates remain robust across ϵ values.

4.4 Re-identification Attack

Table 5 reports re-identification success rates under linkage attacks using external quasi-identifier data. Raw data and static SQL masking provide no meaningful protection, with near-total linkage success (100% matches). View masking reduces linkage to 6.4% but still leaks structural identifiers. Hash, partial, and shuffle masking reduce re-identification risk (50.7% match rate) but remain vulnerable to deterministic linkage. Only generalization fully prevents linkage (0% successful matches) while maintaining aggregate utility (JS divergence $\approx$0.22), demonstrating its robustness against external join attacks.

5 Discussion

Our analysis reveals clear performance–privacy trade-offs across masking strategies. Dynamic masking turned out to be very impractical, entailing over 9,000% overhead when per-row functions

Table 5: Re-identification success rate via quasi-identifier linkage.

Method	Match Rate	Unique Correct
Raw (baseline)	[100.0]%	[0.9]%
View (masked)	[6.4]%	[0.0]%
Static (masked)	[100.0]%	[0.9]%
Hash	[50.7]%	[1.0]%
Partial	[50.7]%	[1.0]%
Shuffle	[50.7]%	[1.0]%
Generalize	[0.0]%	[0.0]%

are called in the role-based query rewriter, pointing to severe inefficiencies in the design of the extension. Static masking, while slower to pre-process, performed acceptably for read-heavy workloads with an increase of just 135%, whereas SQL views achieved the best balance at only 29% overhead, making them the most efficient and easily integrable approach for lightweight anonymization tasks.

Masking functions show distinct strengths: shuffle preserves data utility and distributions with minimal cost but disrupts correlations; generalization offers the strongest re-identification protection with moderate aggregate distortion; hash masking maintains uniqueness but loses semantic meaning. Noise injection, though conceptually related to differential privacy, is slow and lacks formal DP guarantees. Hence, this makes it of limited use in practice. Formal k-anonymity or differential privacy needs to be achieved via external tools such as ARX or specialized DP libraries since the extension internally does not support and track privacy budgets.

Two findings clearly stand out: the catastrophic slowdown of dynamic masking likely is indicative of either a deep architectural limitation or a performance bug in the query rewriting layer, while the complete failure of static masking against linkage attacks confirms that individual columns are masked independently, with correlations among quasi-identifiers ignored. Future work should consider coordinated multi-column transformations and optimized rewriter designs for scalable, privacy-preserving query processing.

6 Limitations and Future Work

6.1 Limitations

- **Docker Overhead:** Running PostgreSQL inside Docker adds an estimated 5–10% performance overhead compared to a native installation. While this does not affect the relative comparison between configurations, absolute latency numbers may vary in bare-metal environments.
- **Simulated Differential Privacy:** Our experiments use an external Laplace noise injection mechanism instead of the extension's built-in differential privacy features. As a result, the reported privacy performance may slightly overestimate what the extension can achieve natively.
- **Single-Node Evaluation:** The current study is limited to a single-machine PostgreSQL setup. Distributed deployments (such as Citus or Postgres-XL) were not tested, and their performance could differ due to network communication and parallel query execution overheads.

- **Read-Only Focus:** Our experiments primarily target `SELECT` queries. Write-heavy workloads (e.g., `INSERT`, `UPDATE`) could introduce additional masking costs, especially under static masking where updates must be propagated to masked replicas.
- **Explain Command Restriction:** The `EXPLAIN` command cannot be executed by masked roles, which limits the ability to inspect query plans or index utilization from a masked user's viewpoint.

6.2 Future Work

(1) **Concurrency Analysis:** Future work should examine how dynamic masking behaves under concurrent workloads using tools such as pgbench. High client loads may exacerbate lock contention on the `pg_seclabel` catalog tables, potentially amplifying performance overhead.
(2) **Partition-Aware Masking:** Another promising direction is to evaluate masking strategies on large partitioned tables (ranging from 10 million to 1 billion rows). PostgreSQL's declarative partitioning could support per-partition masking policies, enabling more efficient handling of temporal or geographically segmented datasets.
(3) **Cross-System Evaluation:** Comparing results against other database systems, such as MySQL Enterprise Masking, SQL Server Dynamic Data Masking, and Snowflake's Row Access Policies, would help contextualize the performance and design trade-offs of **postgresql_anonymizer**.

7 Conclusion

This study benchmarked **postgresql_anonymizer** across masking modes, quantifying performance overheads and privacy–utility trade-offs. Static masking incurred moderate slowdown (+135.3%), whereas dynamic masking was prohibitively expensive (+9214%), and view-based masking remained lightweight (+28.9%). Static masking occasionally outperformed baseline for point lookups due to materialization overhead effects, while k-anonymity–style generalization showed minimal suppression but non-trivial utility loss. Overall, database-native anonymization is viable for read-heavy scenarios with moderate privacy needs; view-based masking offers the best efficiency, static masking serves as a balanced option with caveats on lookup-heavy workloads, and dynamic masking remains impractical for analytical use. The benchmarking methodology supports reproducible evaluation and provides a foundation for future work in privacy-preserving database systems.

References

[1] Dalibo. 2024. PostgreSQL Anonymizer Documentation. https://postgresql-anonymizer.readthedocs.io/. Version 1.3.2.
[2] European Parliament and Council. 2016. General Data Protection Regulation (GDPR). https://gdpr.eu/. Regulation (EU) 2016/679.
[3] Kaggle. 2023. Healthcare Dataset. https://www.kaggle.com/datasets/prasad22/healthcare-dataset. Accessed: 2024-01.
[4] Ron Kohavi and Barry Becker. 1996. Adult Census Income Dataset. https://archive.ics.uci.edu/ml/datasets/adult. UCI Machine Learning Repository.
[5] Fabian Prasser, Florian Kohlmayer, Ronald Lautenschläger, and Klaus A Kuhn. 2014. A benchmark of globally-optimal anonymization methods for biomedical data. In *2014 IEEE 27th International Symposium on Computer-Based Medical Systems*. IEEE, 66–71.
[6] U.S. Department of Health and Human Services. 1996. Health Insurance Portability and Accountability Act (HIPAA). https://www.hhs.gov/hipaa/. Public Law 104-191.