PAC DuckDB Extension

We present the design for a DuckDB extension that compiles a SQL query into a PAC-private plan: generating per-contributor Poisson subsamples, computing per-sample aggregates, applying the PAC privacy aggregator inside the DB, and finally returning privatized outputs. The compiler emits the rewritten SQL (single-query form) into an execution file; that file is executed by DuckDB to produce the final result. This improves portability and builds on top of the SQL-to-SQL compiler approach that has been implemented in our previous work.

We present a minimal functioning version (v1) supporting group by + aggregates (SUM/COUNT/AVG), filters and inner joins only.

---

1. Goals, scope and constraints

We aim to produce an open-source compiler that (correctly) rewrites a supported SQL query into a PAC-compatible plan, optimizes its performance and emits executable SQL. The runtime will materialize a temporary `random_samples_<table>` per query, compute per-sample aggregates, run `pac_aggregate` inside DuckDB, and clean up afterwards.

- Supported SQL constructs: `WHERE` filters, `INNER JOIN`s, `GROUP BY`, aggregates `SUM`, `COUNT`, `AVG`. No nested aggregates, no window functions, no DISTINCT, no MIN/MAX (forbidden), no LEFT/RIGHT/OUTER joins, no complex subqueries.
- Protected unit: single table per deployment (e.g., `customer`) read from a metadata registry file. In the future, we should extend the model to multiple tables.
- Sampling: Poisson/Bernoulli per contributor with inclusion probability `p = 0.5`. Fixed `m = 128` subsets. Both are config knobs to document but fixed for v1.
- Privacy parameter input: Mutual information (MI) measured per-cell (default MI = `1/128` per cell).
- Cardinality refusal rule: global default `k = 3` (configurable).
- S*trict null-based refusal*: if any per-sample result for a cell is `NULL`, the cell is returned as `NULL`.
- Materialization: `random_samples_<table>` is created on-the-fly per query, used during execution, then dropped. This avoids stale reuse and preserves secrecy.
- Execution model: compiler emits SQL (single-query preferred). The emitted SQL is saved to a file (or returned to caller) and then executed.
- Memory model (v1): in-memory; `array_agg` over 128 samples allowed; users should be aware of potentially large memory if output cardinality is high.

---

2. Background

PAC Privacy bounds adversary posterior success (or advantage) using mutual information between input dataset and published output. Instead of pointwise worst-case sensitivity (DP), PAC estimates stability by repeatedly evaluating the query across random subsets drawn from an input distribution `D` (here: Poisson sampling of contributors). Empirical variance of per-sample outputs is used to compute the noise (or the mechanism) required to limit mutual information to a chosen bound β (the MI parameter).

For database queries, PAC-DB expands the query output into cells, computes per-sample evaluations per cell across `m` random subsets, estimates variance, and computes a release guaranteeing `MI(Xi; yi + noise) ≤ β`.

---

3. Required input

Before compiling a query the extension must read from a file or session settings:

- `privacy_unit_table`: which table to treat as contributor (string).
- `mi`: mutual information budget per cell (default `1/128`).
- `k`: minimum group cardinality for release (default `3`).
- `m` and `p` are fixed for v1: `m=128`, `p=0.5`. (config knobs for future work.)

---

4. High-level compilation algorithm

Given an input SQL query `Q`:

1. Validation pass
    - Parse logical plan; ensure query contains at least one aggregate and a scan of `privacy_unit_table`.
    - Confirm only supported operators occur (filters, inner joins, group by, SUM/COUNT/AVG). If not supported, reject with error.
2. Plan rewrite
    - Compile a `CREATE TEMP TABLE random_samples_<unit>` statement (generation described later). The table will contain `(unit_pk, sample_id)` rows for all included contributor/sample pairs.

- Replace every scan of `privacy_unit_table` in the plan with a join to `random_samples_<unit>` on `unit_pk`. For now, we use the *same* `random_samples_<unit>` for all occurrences (aliases, self-joins).
- Add `ps.sample_id` into grouping keys so the engine computes per-group-per-sample aggregates (i.e., group by `(group_keys, sample_id)`).
- Collect per-sample aggregate results (one row per `group_keys` and `sample_id`) and aggregate `array_agg` the per-sample values ordered by `sample_id`.
- Wrap `pac_aggregate(array_of_samples, mi, k)` around the array to produce the final value for that cell. For categorical cells use `pac_categorical_aggregate`.
- Optimize the final query plan.
- Emit the final compiled SQL. The compiled SQL file will include `CREATE TEMP TABLE random_samples_<unit>`; the per-sample query; the outer `pac_aggregate` call; and finally `DROP TABLE random_samples_<unit>`.

3. Emit SQL file
4. Execution

---

5. `random_samples_<table>`

For each contributor primary key `id` and for each `sample_id ∈ [1..m]`, include the row in `random_samples` with probability `p = 0.5` independently. That produces m independent Poisson-style subsets.

```
CREATE TEMP TABLE random_samples_customer (
  customer_id <pk-type>,
  sample_id INTEGER
);
```

We omit an inclusion boolean; presence of a row means inclusion. The compiler emits SQL code that creates and populates this temp table. Example conceptual SQL:

```
CREATE TEMP TABLE random_samples_customer AS
SELECT c.id AS customer_id, s.sample_id
FROM customers c
CROSS JOIN generate_series(1,128) AS s(sample_id)
WHERE random() < 0.5;
```

The temp table is per-query: immediately after the main PAC aggregation completes, the compiled SQL file contains `DROP TABLE IF EXISTS random_samples_customer;` to avoid reuse. This enforces secrecy and prevents stale-sample reuse.

Reusing existing sample tables can break privacy: an attacker can correlate outputs across queries if the same sample set is reused.

---

6. Single-query translation pattern (canonical rewrite)

Original query (example, a typical TPC-H-like query):

```
SELECT c.nation, SUM(o.total_price) AS revenue
FROM customers c
JOIN orders o ON o.cust_id = c.id
WHERE c.region = 'EU'
GROUP BY c.nation;
```

Compiled PAC SQL (single-query style):

```
-- 1: create random samples
CREATE TEMP TABLE random_samples_customer AS
SELECT c.id AS customer_id, s.sample_id
FROM customers c
CROSS JOIN generate_series(1,128) AS s(sample_id)
WHERE random() < 0.5;

-- 2: per-sample aggregation (GROUP BY group_keys + sample_id)
WITH per_sample AS (
  SELECT
    c.nation AS nation,
    rs.sample_id AS sample_id,
    SUM(o.total_price) AS revenue_sample
  FROM random_samples_customer rs
  JOIN customers c ON c.id = rs.customer_id
  JOIN orders o ON o.cust_id = c.id
  WHERE c.region = 'EU'
  GROUP BY c.nation, rs.sample_id
)
-- 3: collect arrays across sample_id, apply PAC aggregator
SELECT
  nation,
```

```
  pac_aggregate(array_agg(revenue_sample ORDER BY sample_id), 1.0/128, 3)
AS revenue_pac
FROM per_sample
GROUP BY nation;

-- 4: cleanup
DROP TABLE IF EXISTS random_samples_customer;
```

Notes:

- `NULL`s are handled inside `pac_aggregate`.
- `pac_aggregate(samples_array, mi, k)` implements PAC variance estimation, noise calculation, and refusal logic (see next section).

---

7. `pac_aggregate`

We implement the exact algorithm described in the thesis (Algorithm 2 / Theorem 1 logic). `pac_aggregate` runs inside DuckDB as a built-in UDF/aggregate and receives the per-sample vector. It computes noise required for MI bound and returns either the privatized scalar or `NULL` on refusal.

```
-- Numeric aggregate
pac_aggregate(samples ARRAY<DOUBLE>, mi DOUBLE DEFAULT 1.0/128, k INTEGER
DEFAULT 3)
  RETURNS DOUBLE
-- Categorical aggregate (for top-k / names): samples ARRAY<TEXT>
pac_categorical_aggregate(samples ARRAY<TEXT>, mi DOUBLE DEFAULT 1.0/128, k
INTEGER DEFAULT 3)
  RETURNS TEXT
```

- `samples`: an array containing per-sample aggregate numeric outputs for a fixed (group, attribute). Elements correspond to samples `i=1..m` in order (or missing if a group had no contributors included in that sample, represented as `NULL` in the array).
- `mi`: mutual information budget β for this cell (default `1/128`).
- `k`: minimal group cardinality (configurable; default `3`). The function must inspect contributor count if available, or infer from `samples` presence.

The function receives only the array of per-sample outputs; it must be able to compute per-sample counts or detect `NULL` samples. For cardinality `k` check we rely on per-sample counts, so the per-sample aggregation should include counts as part of the array payload or be available via a parallel `array_agg(count)` input. For v1, the compiler will ensure that for every numeric cell the `per_sample` CTE also includes `COUNT(*) AS cnt_sample`, and `pac_aggregate` will accept two arrays: `values_array, counts_array`. So final signature for numeric is:

```
pac_aggregate(values ARRAY<DOUBLE>, counts ARRAY<INT>, mi DOUBLE, k INT)
RETURNS DOUBLE
```

The compiler emits `array_agg(revenue_sample ORDER BY sample_id)` and `array_agg(cnt_sample ORDER BY sample_id)`.

Output: a single numeric value equal to the privatized release for that cell, or `NULL` if refusal criteria triggers.

For categorical `pac_categorical_aggregate`, the function receives an array of per-sample winners and returns either a uniformly selected winner across the set of observed winners or `NULL` if refusal.

Algorithm (numeric, high-level):

1. Cardinality check: compute total unique contributors across samples if possible. Simpler: compute `median` or `mean` of per-sample `counts_array`. If the mean/median < `k`, return `NULL`. (For v1 we implement: if the maximum `counts_array` < `k` then `NULL`).
2. Null handling: if any element of `values` is `NULL` (i.e., group missing in a sample), return `NULL` per strict refusal rule.
3. Compute empirical variance σ²m: compute sample variance of `values` across m samples: `σ²_m = VAR_SAMPLES(values)`. Use the unbiased estimator (divide by m−1) for stability.
4. Noise variance mapping: using the PAC-DB mapping (Algorithm 2): set noise variance `Δ = σ²_m / (2 * β)` (β is MI). This follows the thesis where `Δ := σ²_m / (2β)`. (Cite thesis Section 3.5.3 and Algorithm 2.)
5. Sample secret subset: the final PAC-DB release picks a single sample `i` uniformly at random from `1..m` and returns `value[i] + N(0, Δ)`. However, since we do not want the runtime to leak which `i` was chosen, the mechanism already randomizes `i` internally and returns the noisy scalar. (Implementation picks `i = random_integer(1, m)`, draws gaussian noise with variance Δ, returns `values[i] + N(0, Δ)`.)
6. Return `values[i] + gaussian(0, sqrt(Δ))`.

Algorithm (categorical):

1. If any per-sample value is `NULL`, return `NULL`.
2. Build the deduped set of observed winners `Y = distinct(samples)`.
3. If the deduped set is empty → `NULL`. If cardinality check fails (counts < k) → `NULL`.
4. Return a uniformly random element from `Y`. (This matches the thesis's suggested simple technique.)

Implementation considerations:

- Determinism for debugging: use session-level random seed option to reproduce runs; otherwise use system RNG.
- Gaussian noise: implement Box-Muller or use DB internal RNG to generate normally distributed noise. Use double precision.
- Precision and numeric stability: if `σ²m` is extremely small (near zero), Δ will be near zero; then return `yi + negligible_noise`.
- Edge cases: if `values` are identical across all samples, `σ²m = 0`, Δ=0 → return `yi` (no noise needed).

---

8. Arrays in the SQL rewrite

To satisfy `pac_aggregate(values_array, counts_array, mi, k)`, the compiler must ensure `per_sample` CTE computes both the per-sample value and per-sample contributor count:

```
WITH per_sample AS (
  SELECT
    <group_keys>,
    rs.sample_id AS sample_id,
    SUM(o.total_price) AS revenue_sample,
    COUNT(DISTINCT c.id) AS cnt_sample  -- count of contributors in that
per-sample group
  FROM random_samples_customer rs
  JOIN customers c ON c.id = rs.customer_id
  JOIN orders o ON o.cust_id = c.id
  WHERE ...
  GROUP BY <group_keys>, rs.sample_id
)
```

Then outer query does:

```
SELECT
  <group_keys>,
  pac_aggregate(
    array_agg(revenue_sample ORDER BY sample_id),
    array_agg(cnt_sample ORDER BY sample_id),
    :mi, :k
  ) AS revenue_pac
FROM per_sample
GROUP BY <group_keys>;
```

This ensures `pac_aggregate` has both per-sample numeric outputs and per-sample contributor counts to implement cardinality checks.

---

9. Categorical outputs

For queries that return categorical outputs (e.g., "top supplier"), the rewrite pattern:

1. Compute per-sample top seller (e.g., `arg_max` or `max_by` in each sample) as `top_seller_sample`.
2. `per_sample` CTE returns `(group_keys, sample_id, top_seller_sample)`.

Outer query:

```
SELECT
  group_keys,
  pac_categorical_aggregate(
    array_agg(top_seller_sample ORDER BY sample_id),
    array_agg(cnt_sample ORDER BY sample_id),
    :mi, :k
  ) AS top_seller_pac
FROM per_sample
GROUP BY group_keys;
```

`pac_categorical_aggregate` dedupes the set of observed winners (Y) and returns a uniformly random member of Y or `NULL` if refusal.

Rationale: categorical outputs do not have a natural numeric variance; the thesis suggests returning a random element among potential winners to ensure MI(Q(X), X) = 0 conditional on the deduped set being independent of subset choice.

---

10. Operator reordering (skip for now)

Problem statement. The DB optimizer may attempt to collapse or reorder `GROUP BY` nodes and may push or pull filters and joins. Such transformations could change the shape of the intermediate results that our `pac_aggregate` expects, threatening both correctness and the required per-sample structure for variance estimation.

Concrete issues:

- If the optimizer rewrites away `per_sample` grouping by merging it with the outer grouping, the `array_agg` may be computed incorrectly (single group aggregate across all samples rather than sample-separated).

- If the optimizer pushes filters down past the `random_samples` join or merges joins differently, the effective sampling distribution could change.

Mitigation (v1 conservative approach):

- The compiler will generate the `per_sample` CTE as a named temp table where necessary (materialized) to force the DB to compute per-sample groups first. This is the multi-statement alternative when the optimizer is suspected to interfere.

- When emitting the single-query CTE form, the compiler will mark the `per_sample` CTE with constructs/idioms that discourage optimizer collapse (e.g., use explicit `GROUP BY ...` and avoid expressions that the optimizer can trivially merge). We will include a short explanatory note in the compiled SQL and code comment for future-proofing.

- For production-hardening later, implement a physical-plan-level guard (in the extension) that, during optimization, recognizes the per-sample group and ensures grouping boundaries are preserved (this is future work).

Summary for v1: assume DuckDB does not collapse the structure; but implement a safe alternative (materialize per-sample results into temp table) if necessary. The compiler may produce either a single WITH-based statement or a small sequence materializing per-sample results, depending on optimizer behavior during tests.

---

11. Forbidden constructs

- MIN/MAX: forbidden in v1, compilation fails with an explicit message. Rationale: min/max are unstable and require special handling.
- DISTINCT, window functions, LEFT/OUTER joins, correlated subqueries: compilation fails; the compiler will return an error explaining why and suggesting alternatives (e.g., rewrite to supported operator set).
- If the query does not reference the protected table or contains only safe non-dependent elements (e.g., Q2 in TPC-H), the compiler should decline to PAC the query and return original SQL (or label it as MI=0 and run unchanged).
- If `per_sample` grouping results in missing sample entries leading to frequent nulls, `pac_aggregate` returns `NULL` and the caller receives `NULL` for that cell.

---

12. Security model

The adversary:

- Knows the PAC-DB algorithm, the implementation, the MI budget, and the distribution used to create `random_samples` (i.e., Poisson p = 0.5 and m).
- Does not know which sampled subset (index `i`) was used to produce the released cell value, nor the per-sample seed.
- May issue adaptive queries.

Given the above, and assuming independent re-sampling per cell and correct implementation of `pac_aggregate` (Algorithm 2), the mutual information between the chosen subset and published output is bounded by `mi` per cell, which composes linearly across cells/queries as in the thesis (Section 3.4). When `pac_aggregate` returns `NULL`, the system refuses to release to preserve privacy.

Operational protections:

- Per-query fresh `random_samples` ensures no cross-query correlation from sample reuse.
- The extension drops temp tables at the end of execution.

- Seed storage: for production, do not persist seeds to disk; only keep them ephemeral in memory during execution. Session-level reproducible seed available in debug mode only.

---

13. Limitations, assumptions, and future work

Limitations (v1):

- Only supports a narrow SQL subset (GROUP BY + SUM/COUNT/AVG + inner joins + filters).
- Memory: `array_agg` produces arrays of length up to 128 per output cell — large result cardinalities may require more memory.
- Conservative refusal rules: strict `NULL`-based refusal will cause some cells to be `NULL` more often than strictly necessary for tight PAC bounds; this is safe but conservative.
- The compiler assumes DuckDB does not collapse the `per_sample` grouping in a way that breaks semantics; a materialized fallback is included to handle optimizer behavior but may increase IO.

Assumptions:

- The privacy unit table is declared correctly in metadata and is the right atomic contributor.
- Poisson sampling with p=0.5 and m=128 is appropriate for MI computations used.
- The adversary model used in thesis (does not know subset chosen) holds.

Future work:

- Operator algebraic rules for safe join and filter pushdown; rigorous conditions for pushdown.
- Optimized single-pass transformations and physical operator `sample` that can be fused with scans.
- Persistent rotated sample caches with privacy-aware refresh to amortize cost.
- Broader support for aggregates, categorical release mechanisms beyond uniform sampling, and improved refusal heuristics.
- Composition accounting: track MI consumption across sessions and users.

---

14. Example: TPC-H Q4 style (concise)

Original Q4 (simplified):

```
SELECT count(*) FROM orders o
JOIN lineitem l ON l.order_key = o.order_key
WHERE l.receipt_date > l.commit_date;
```

PAC-compiled (conceptual):

```
CREATE TEMP TABLE random_samples_customer AS
  SELECT c.id AS customer_id, s.sample_id
  FROM customers c
  CROSS JOIN generate_series(1,128) AS s(sample_id)
  WHERE random() < 0.5;

WITH per_sample AS (
  SELECT
    rs.sample_id,
    COUNT(*) AS cnt_sample
  FROM random_samples_customer rs
  JOIN customers c ON c.id = rs.customer_id
  JOIN orders o ON o.cust_id = c.id
  JOIN lineitem l ON l.order_key = o.order_key
  WHERE l.receipt_date > l.commit_date
  GROUP BY rs.sample_id
)
SELECT pac_aggregate(array_agg(cnt_sample ORDER BY sample_id),
                     array_agg(cnt_sample ORDER BY sample_id),
                     :mi, :k) AS pac_count
FROM per_sample;

DROP TABLE IF EXISTS random_samples_customer;
```

(This is simplified to show pattern - in practice you may need to propagate group keys.)
