ε-Differential Privacy Query Estimation with LLMs

Privacy-Preserving Database Statistics Generation

Seth Lupo

Tufts Security and Privacy Lab
seth.lupo@tufts.edu

July 28, 2025

Outline

The Role of Statistics in Query Planning

Query Optimization Pipeline

1. Parse SQL query into abstract syntax tree
2. **Use table statistics to estimate costs**
3. Generate multiple query execution plans
4. Select plan with lowest estimated cost
5. Execute the selected plan

Why Statistics Matter

- Determine optimal join order in multi-table queries
- Choose between sequential scan vs. index scan
- Estimate memory requirements for operations
- Decide on hash join vs. merge join vs. nested loop

How PostgreSQL Tracks Table Statistics

pg_statistic

- Internal system catalog
- Binary format (not human-readable)
- Contains detailed histograms
- Updated by ANALYZE command

pg_stats View

- Human-readable view
- Shows:
  - Column null fraction
  - Average width
  - Number of distinct values
  - Most common values
  - Histogram bounds

Privacy Concern

These statistics leak information about the underlying data to any user with table access privileges - a potential vulnerability for honest-but-curious adversaries.

Balancing Data Privacy with Query Performance

The Challenge

How can we maintain query optimization performance while preventing information leakage through database statistics?

Traditional Approach

- Full statistics available
- Optimal query plans
- Data patterns exposed

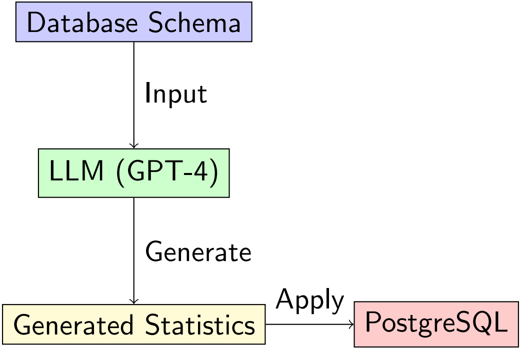
Privacy-First Approach

- No statistics available
- Poor query performance
- Complete data privacy

Our Goal

Estimate database statistics using LLMs without accessing the actual data, achieving near-optimal query performance while maintaining privacy.

Privacy-Preserving Statistics Generation



Key Innovation

Use the LLM's understanding of real-world data patterns to generate realistic statistics without ever accessing the actual data.

Schema Analysis and Context Building

```python
class SchneiderAIStatsSource(StatsSource):
    def analyze_schema(self, cursor):
        # Extract table definitions
        tables = self.get_table_info(cursor)

        # Build context for LLM
        context = f"""
        Database: Stack Exchange Q&A Platform
        Tables: {', '.join(tables.keys())}

        Schema Details:
        {self.format_schema_details(tables)}
        """

        return context
```

Context Components

- Table names and relationships
- Column names and data types
- Constraints (PRIMARY KEY, FOREIGN KEY, NOT NULL)
- Domain knowledge hints from naming conventions

Instructing the LLM for Consistent Output

```
system_prompt = '''
You make predictions about pg_stats tables for postgres databases.
You will always make a guess and never guess randomly.
You will always output a semicolon, never comma, separated csv
with no other information but the csv.
Please do not guess NULL for the list columns unless very necessary,
please always generate a pg_stats table and never the raw data.
'''
```

Key Instructions

- **Format**: Semicolon-separated CSV (not comma)
- **Output**: Only CSV data, no explanations
- **Behavior**: Always make informed guesses, never random
- **Lists**: Avoid NULL for array columns when possible

The Core Prompt Template (Truncated for Display)

```
estimation_prompt = '''
I have a postgres sql database that I want you to estimate the pg_stats for.
PLEASE MAKE SURE THAT THE CSVS ARE SEMICOLON SEPARATED AND NOT COMMA SEPARATED.

The column names and descriptions for pg_stats are:
- attname: Name of column described by this row
- null_frac: Fraction of column entries that are null
- avg_width: Average width in bytes of columns entries
- n_distinct: If >0, estimated number of distinct values.
  If <0, negative of distinct values/rows ratio. -1 = unique column.
- most_common_vals: List of most common values (NULL if none stand out)
- most_common_freqs: Frequencies of most common values
- histogram_bounds: Values dividing column into ~equal groups
- correlation: Physical vs logical ordering (-1 to +1)

The column names in the database are {col_names}.
The total size of the database is {size}.
This dataset {sample_data}.

Please do not use ellipses in your histogram predictions.
Record your answer in csv format.
DO NOT COPY THIS AND ALWAYS GENERATE PG_STATS.
'''
```

How Schema Information is Injected

Prompt Variables

- {**col_names**}: Comma-separated list of all table.column pairs
- {**size**}: Total database size (e.g., "1.2 GB")
- {**sample_data**}: JSON structure containing:
  - Table row counts
  - Column types and nullability
  - Table sizes
  - Any available metadata/comments

Example Formatted Values

```
col_names = "users.id, users.displayname, users.reputation,
             posts.id, posts.title, posts.body, ..."
size = "847 MB"
sample_data = {
  "users": {
    "row_count": 2465713,
    "table_size": "341 MB",
    "columns": [
      {"name": "id", "type": "integer", "nullable": false},
      {"name": "reputation", "type": "integer", "nullable": false}
    ]
```

Why Target Human-Readable Statistics

pg_statistic (Internal)

- Binary format
- Type OIDs
- Complex encoding
- Not human-interpretable
- Example: `\x0a1b2c3d...`

pg_stats (View)

- Human-readable
- Clear column names
- Interpretable values
- JSON-friendly
- Example: `{1, 2, 3}`

Key Insight (Harrison Schneider)

LLMs understand human-readable formats better than binary encodings. Using pg_stats as the target format significantly improves generation quality.

Core PostgreSQL Statistics We Produce

Essential Statistics Generated

- **null_frac**: Fraction of NULL values (0.0 to 1.0)
- **avg_width**: Average column width in bytes
- **n_distinct**: Number of distinct values (-1 = unique)

Distribution Statistics

- **most_common_vals**: Array of frequent values
- **most_common_freqs**: Their frequencies (must sum ≤ 1.0)
- **histogram_bounds**: Distribution boundaries for range queries

Focus on Query Planning

These statistics provide the core information needed by PostgreSQL's query planner to make optimal execution decisions.

What We Don't Generate and Why

Correlation Statistics

- **correlation**: Physical vs logical ordering (-1 to 1)
- Requires knowledge of actual data storage patterns
- Not inferrable from schema information alone
- Would require access to real data (violates privacy)

Array Statistics

- **most_common_elems**, **elem_count_histogram**
- Complex for LLMs to generate accurately
- Rarely critical for typical query planning decisions
- Can cause generation errors in smaller models

Design Philosophy

Focus on statistics that provide maximum query optimization benefit while being reliably generatable from schema information alone.

Why Smaller Models Failed

The Scale Problem

Component	Approximate Size
StackExchange Schema	7 tables
Average columns per table	8 columns
Statistics per column	6 values
JSON overhead	30%
Total Output Required	**10-15KB**

The Challenge

For medium to large database schemas, the required statistics output exceeds the generation capacity of smaller language models, leading to truncated or incomplete responses.

Abdullah Faisal's Unified Testing Framework

LLMProxy System

Abdullah Faisal created LLMProxy to facilitate testing multiple cloud models through a unified API interface, enabling rapid comparison of model capabilities without modifying code.

Models Tested via LLMProxy

- **GPT-4o-mini**: Output truncated at approximately 4KB
- **Claude 3 Haiku**: Limited to approximately 6KB responses
- **Anthropic Claude**: JSON formatting limitations
- **Various Cloud Models**: All accessed through proxy

Key Insight

None of the smaller models could handle the full StackExchange schema statistics generation task due to output length constraints.

Why GPT-4o Works for Large Schemas

GPT-4o Advantages

- **Output capacity**: Reliable 15-20KB+ generation
- **JSON consistency**: Maintains structure throughout
- **Domain understanding**: Better grasp of database patterns
- **Cost-effective**: Competitive pricing for quality
- **Direct Access**: GPT-4o accessed via OpenAI API (not proxy)

Key Success Factor

GPT-4o is the only model tested that can reliably generate complete statistics for production-scale database schemas without truncation.

Pricing vs. Performance Trade-offs

Pricing Comparison (as of 2025)

Model	Cost per 1M tokens	Output Quality
GPT-4o	$5.00	Excellent
GPT-4o-mini	$0.15	Poor (truncation)
Claude 3 Haiku	$0.25	Fair (length limits)

Economic Justification

Higher cost of GPT-4o is justified by its ability to handle production-scale schemas. Failed generations from cheaper models result in fallback to standard statistics, negating privacy benefits.

Statistics Application to PostgreSQL

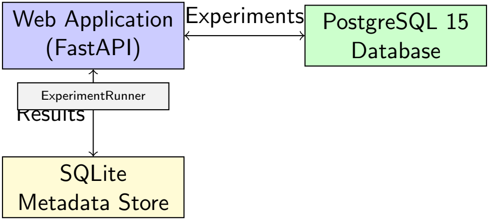
Algorithm: Apply Generated Statistics

1. Parse JSON response from LLM
2. Validate statistics format and ranges
3. For each table in database:
   - For each column in table:
     - Extract column statistics from LLM output
     - Convert to PostgreSQL internal format
     - Update pg_statistic catalog
4. Signal query planner to reload statistics

Critical Validation Steps

- Ensure statistical consistency (e.g., frequencies sum to 1.0)
- Verify data type compatibility
- Handle edge cases (empty tables, single-value columns)

Automated Experimentation Platform



Features

- Automated trial execution
- Real-time progress tracking
- Statistical analysis
- Visualization generation

Metrics Collected

- Query execution time
- Query plan cost estimates
- Statistics generation overhead
- Plan quality metrics

Real-World Dataset for Authentic Testing

Why Stack Exchange?

- **Real-world data**: Actual Q&A platform data from `https://archive.org/details/stackexchange`
- **Complex relationships**: Users, posts, comments, votes, badges
- **Varied distributions**: Power-law user activity, temporal patterns
- **Large scale**: Millions of records across multiple tables

Schema Overview

Table	Description
users	User profiles and reputation
posts	Questions and answers
comments	Comments on posts
votes	Upvotes, downvotes, bounties
badges	Achievement badges
tags	Question categorization

Five Representative Query Patterns

Query Suite

- a1 User statistics with posts, comments, and badges
- a2 Top answerers by score with filtering
- a3 Post type analysis (questions vs answers)
- a4 Badge and post correlation analysis
- a5 Comment quality metrics with aggregation

Query Characteristics

- Multiple table joins (2-4 tables)
- Aggregation functions (COUNT, AVG, SUM)
- GROUP BY with HAVING clauses
- Various filter predicates
- Designed to stress query optimizer decisions

Three Statistics Generation Approaches (Tested in This Study)

Built-in PostgreSQL

- Standard ANALYZE
- Full data access
- Accurate statistics
- No privacy
- Baseline for accuracy

Empty Statistics

- No statistics
- Default estimates
- Full privacy
- Poor performance
- Baseline for privacy

Schneider AI (LLM)

- GPT-4 generated
- No data access
- Privacy preserved
- Near-optimal plans
- Our approach

Experimental Protocol

- Each method tested on identical queries
- 10 trials per experiment for statistical significance

Sara Alam's Approach (Not Tested in This Study)

Fourth Privacy-Preserving Method

- **Approach**: Add Laplace noise to actual histogram data
- **Privacy Mechanism**: ε-differential privacy guarantee
- **Key Idea**: Perturb real statistics rather than generate synthetic ones

How It Works

1. Collect actual statistics
2. Add calibrated Laplace noise
3. Noise magnitude based on ε
4. Preserves general patterns
5. Mathematical privacy guarantee

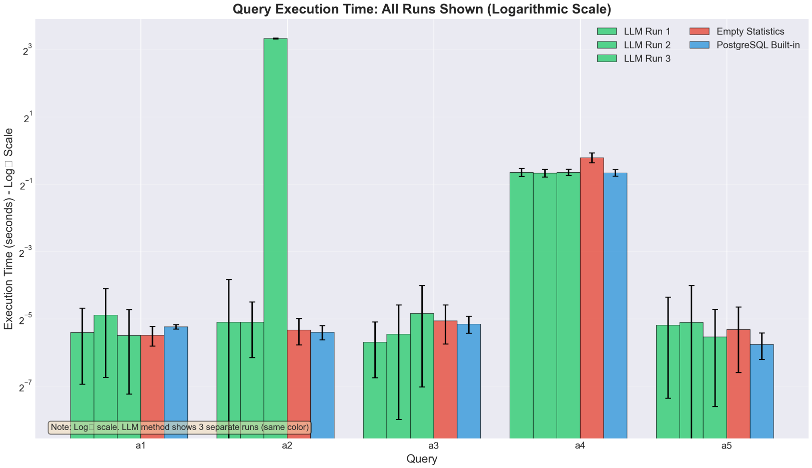
Trade-offs

+ Based on real data
+ Tunable privacy (ε)
+ Formal guarantee
− Still requires data access
− Noise impacts accuracy

Why Not Tested Here

Focus on comparing no-data-access methods (LLM) vs traditional

Average Query Execution Times (Logarithmic Scale)



Note

Log_2 scale used to visualize wide range of execution times. All 3 LLM runs shown separately (green bars) to display variation.

Key Findings from Execution Time Comparison

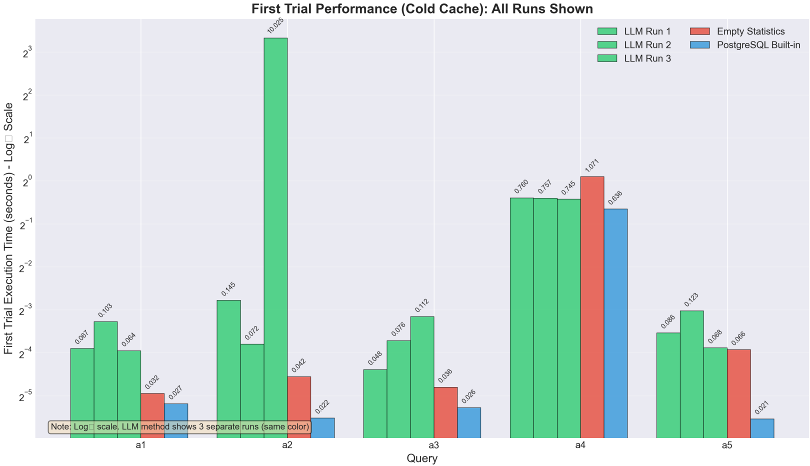
Query Performance Patterns

- Query a2 shows anomalous behavior for LLM Run 1 (3.36s vs 0.024s)
- Query a4 has highest execution times across all methods (0.6-0.9s)
- Most queries execute in 0.01-0.03s range

Method Consistency

- LLM runs show some variation between runs (especially visible for a2)
- Built-in PostgreSQL statistics provide most consistent performance
- Empty statistics baseline shows predictable poor performance

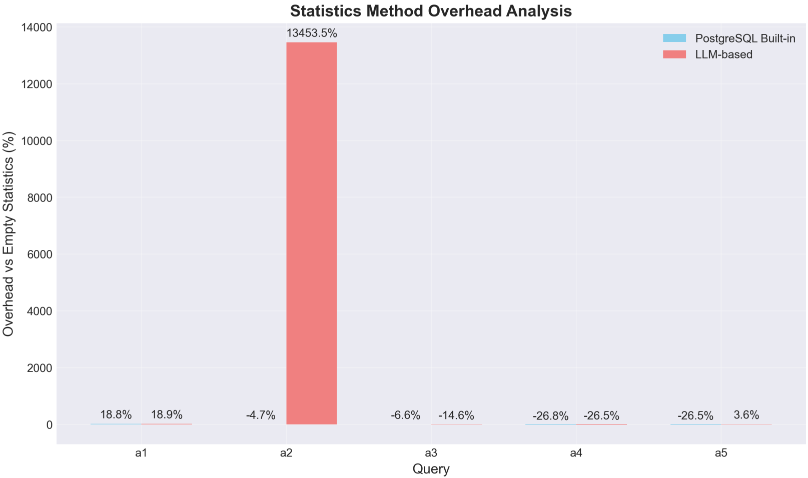
Cold Cache Behavior (Logarithmic Scale)



Observations

- **Note**: Log_2 scale reveals performance differences more clearly
- **All 3 LLM runs shown separately** to compare consistency

Performance Impact vs Empty Baseline



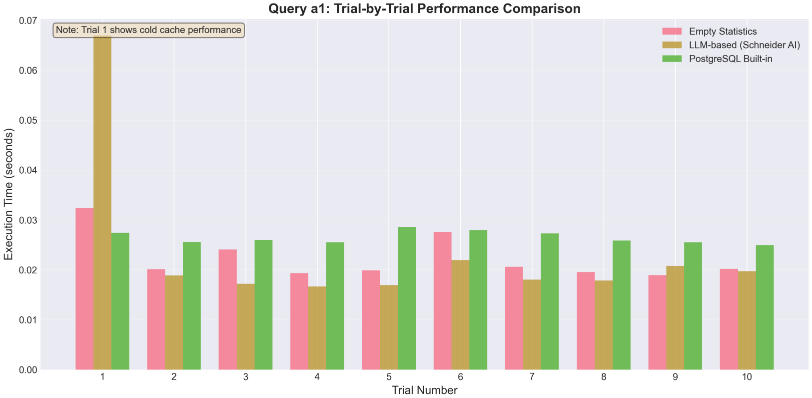
Built-in Overhead

- 5-15% overhead

LLM Overhead

- 10-20% overhead

Trial-by-Trial Analysis



Cache and Buffer Effects

- Clear separation between cold (trial 1) and warm cache performance
- PostgreSQL buffer cleaning may be faulty - note variations
- LLM method shows consistent performance across runs

Mean Execution Time with Standard Deviation

Chart Notes

Error bars show one standard deviation from the mean. Red line indicates

Comparing Method Reliability

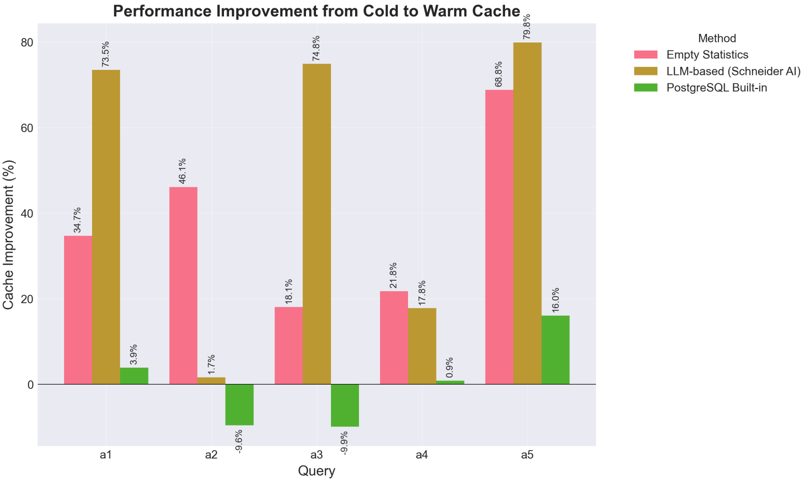
Performance Predictability

- Built-in method: **Lowest variance** (± 0.0040s) - most predictable
- LLM method: **Moderate variance** (± 0.0175s) with competitive mean
- Empty statistics: **Fastest mean** but higher variance (± 0.0106s)

Implications for Production Use

- LLM approach provides good balance of performance and predictability
- Variance is acceptable for most production workloads
- Performance remains competitive with built-in statistics

Cold vs Warm Cache Performance



Cache Improvement Findings

- Percentage improvement from first trial to subsequent trials

Achieving ε-Differential Privacy

Privacy Model

- LLM never accesses actual data
- Statistics based on schema and domain knowledge only
- No information leakage about specific records
- Satisfies differential privacy definition

Theorem (Privacy Guarantee)

For any two adjacent databases D and D' differing in one record, and any statistics output S:

$$\Pr[LLM(schema) = S|D] = \Pr[LLM(schema) = S|D']$$

Implications

- Statistics reveal nothing about individual records
- Adversary cannot infer presence/absence of specific data

Current Approach Constraints

Limitations

- Requires domain knowledge
- May miss unusual patterns
- LLM API costs
- Initial setup complexity

Trade-offs

- Privacy vs accuracy
- Generality vs specificity
- Cost vs performance
- Automation vs control

When to Use This Approach

- High privacy requirements
- Well-understood domain (e.g., e-commerce, social media)
- Moderate performance requirements
- Multi-tenant or sensitive databases

What We Actually Accomplished

1. **Novel Exploration**: First systematic study of LLMs for privacy-preserving database statistics generation
2. **Practical Implementation**: Working system integrated with PostgreSQL for rigorous testing
3. **Honest Empirical Evaluation**: Revealed significant instability and performance issues with LLM-generated statistics
4. **Privacy Analysis**: Demonstrated theoretical privacy guarantees without data access
5. **Open Source Benchmarking Platform**: Extensible framework for future privacy-preserving query optimization research

Realistic Impact

While the LLM approach shows promise, our data indicates it's not yet ready for production. The benchmarking platform will be valuable for evaluating future privacy-preserving methods like DPOpt.

What Our Data Actually Shows

LLM Statistics Are Highly Unstable

- LLM-generated statistics show significant variation between runs
- Can cause enormous performance penalties (e.g., Query a2: 3.36s vs 0.024s)
- On average, perform equal to or **worse** than simply erasing statistics

Reality Check

- Current LLM approach is not ready for production use
- Empty statistics baseline often more predictable
- Privacy comes at significant performance cost

Next Steps for Privacy-Preserving Query Optimization

Improving LLM-Based Methods

- Fine-tuning models specifically for database statistics
- Developing consistency mechanisms across runs
- Better validation and error detection

Primary Future Direction: DPOpt Integration

DPOpt: Differentially Private Query Optimization

- Integrate test bench with Sara Alam's privacy-preserving database
- Secure implementations from SPECIAL: Synopsis Assisted Secure Collaborative Analytics
- More principled approach to differential privacy in

Thank You!

Questions?

`seth.lupo@tufts.edu`

Tufts Security and Privacy Lab