Experimental Comparison of Database Provenance Systems

CS594 - Provenance and explanations

Final Report

Git Repository Link - https://github.com/dhotrerevathi/data_provenance.git

By: Revathi Dhotre and Sri Keshav Katragadda

Abstract

This report presents a detailed experimental comparison of two database provenance systems, ProvSQL and GProM, using the Flight Delay dataset we found on Kaggle. The dataset was divided into three sizes, small, medium, and large, to evaluate system performance across different data scales, focusing on key aspects such as query execution efficiency, functionality, and the ability to manage various provenance types.

In this experiment, we examined HOW, WHERE, and WHY provenance, testing the systems against 21 queries involving complex operations like joins, aggregations and selections. These queries were divided for a structured evaluation of each system's strengths and weaknesses.

By analyzing their performance and practical applicability, in this study we aim to highlight the capabilities of ProvSQL and GProM.

Introduction

In today's data-driven world, understanding the origin, transformation, and usage of data, collectively known as data provenance, is critical across industries, from scientific research to business analytics. Data provenance tracks the lineage and lifecycle of data, offering insights into where it comes from, how it has been transformed, and why it is relevant.

Provenance plays an important role in ensuring transparency, reproducibility, and accountability. It lets its users verify data quality and debug work pipelines. For example, in scientific research, provenance ensures experimental results are reproducible.

Data provenance is generally categorized into three types:

- **WHY**: Explains the reasons behind a specific data outcome.
- **WHERE**: Traces the origins and intermediate sources of data.
- **HOW**: Details the processes and transformations applied to the data.

This report explores the practical applications of data provenance by comparing two systems: ProvSQL and GProM. By looking at their strengths, limitations, and capabilities, we highlight how these tools support provenance tracking and management, underscoring their importance in generating actionable insights from complex datasets.

Research Objective -

This study aims to:

1. Compare the capabilities of ProvSQL and GProM in managing various types of provenance queries.
2. Evaluate the performance of both systems across datasets of different sizes.
3. Analyze the scalability of each system under increasing data complexity.
4. Provide insights to guide the selection of a provenance system for specific query needs.

Background and Related work -

The need to track, manage, and analyze data has driven the development of various tools and frameworks for data provenance. Among these, ProvSQL and GProM stand out for their unique strengths.

ProvSQL

ProvSQL, introduced at Provenance Week 2018, is an extension of the PostgreSQL database that integrates provenance tracking into an existing relational database management system (RDBMS). Its defining feature is the use of semiring-based provenance polynomials, which provide a compact and algebraic representation of data lineage.

- **Semiring-Based Provenance**:
  Semirings, mathematical structures supporting addition and multiplication, encode the lineage of query results in polynomial form. This enables ProvSQL to trace how query outputs are derived from input data efficiently. For instance, it can map how a tuple in the output relates to specific input tuples and transformations.
- **User-Defined Aliases**:
  ProvSQL allows users to assign custom names to provenance annotations, enhancing the clarity and readability of complex lineage expressions.
- **Functional Capabilities**:
  ProvSQL manages aggregations, joins, and selections, supporting WHERE provenance (data origins) and HOW provenance (transformation sequences).
- **Advantages**:
  ProvSQL is ideal for rigorous mathematical lineage tracking applications, such as scientific data analysis and compliance audits. Its integration with PostgreSQL makes it accessible to users who are familiar with the database.

GProM

GProM builds on its predecessor, Perm, with a novel framework for managing data provenance. Instead of semiring-based methods, GProM employs provenance game graphs and advanced query rewriting techniques to capture and analyze data lineage.

- **Provenance Game Graphs**:
  These visual and computational models represent relationships and transformations between data entities in a query. Nodes correspond to data operations, while edges trace the flow of annotations, making it easier to interpret dependencies.
- **Annotation Propagation**:
  GProM attaches annotations (e.g., metadata or tags) to data tuples and updates them through query operations, ensuring seamless tracking of provenance across stages.
- **Query Rewriting**:
  GProM modifies queries dynamically to compute provenance alongside query results, minimizing disruption to the query structure.
- **Functional Capabilities**:
  Supporting WHY, WHERE, and HOW provenance, GProM handles complex queries, including nested structures and multi-joins, with efficiency and flexibility.
- **Advantages**:
  GProM's visualization and dynamic adaptation make it ideal for debugging, optimizing workflows, and exploratory data analysis.

Summary

While ProvSQL and GProM significantly contribute to data provenance, they have different use cases. ProvSQL's semiring-based method uses precision and is best suited for scientific applications. GProM's game graph approach prioritizes interpretability and adaptability, which is better in debugging and exploratory scenarios. Together, we think these tools highlight the diverse strategies available for managing data provenance.

Feature set Comparision:

Feature	GProM	ProvSQL
Description	Middleware for provenance management combining SQL/Datalog queries with provenance queries.	PostgreSQL extension implementing provenance using semirings.

Supported DBMS	PostgreSQL, SQLite, Oracle, MonetDB.	PostgreSQL only.
Query Language	SQL and Datalog.	SQL only.
Provenance Types	Supports why and why-not provenance queries (provenance games with colored graphs).	Computes provenance using semirings (e.g., counting, Boolean, tropical semirings).
Graph Output	Generates provenance game graphs for why and why-not provenance.	Outputs provenance graphs and formulas with IDs replaced by aliases for better readability.
Semiring Support	Not explicitly supported.	Supports various semirings like counting, tropical, positive Boolean, and user-defined semirings.
Aggregate Functions	Supports aggregates like SUM, MAX, and AVG.	Limited support for aggregates (e.g., MAX, AVG, not supported as of 2021).
Provenance IDs	Not required; directly derives provenance information.	Uses add provenance to assign provenance IDs and create provenance mapping for user aliases.
Implementation	Middleware that translates queries to provenance-enhanced SQL, executed on the backend DBMS.	The extension is embedded in PostgreSQL, enhancing its native capabilities.
Visualization	Generates provenance graphs with color-coded nodes for root nodes (depending on query type).	Exports provenance graphs and formulas, which can be visualized using external tools.

Customization	Supports query customization for provenance analysis through SQL or Datalog.	Allows user-defined semirings and provenance mapping customization for enhanced readability.
Ease of Installation	Open-source requires a separate frontend and backend setup.	Open-source; installed as an extension to PostgreSQL.
Documentation	Well-documented with reproducible examples.	Well-documented and continuously updated with new features.
Performance	Performance depends on middleware configuration and backend DBMS.	Performance is tied directly to PostgreSQL and semiring computations.
Best Use Cases	Use for complex queries, transactional provenance, and combined why/why-not provenance requirements.	Use for mathematical representations of provenance (e.g., formulas) and applications needing semirings.
Example Applications	Provenance analysis for debugging, auditing, and compliance in multi-DBMS environments.	Mathematical modeling of provenance for queries in scientific and analytical domains.

Experimental Setup -

Dataset Characteristics

For this study, we used a real-world flight dataset from Kaggle. The dataset is divided into three sizes. Small (500,000 rows), Medium (1,000,000 rows), and Large (3,000,000 rows) for our evaluation of ProvSQL and GProM across varying data volumes and complexities.

The database schema below outlines the structure of the flight delay dataset, specifying the tables and their columns. This schema was used to construct queries and evaluate provenance systems:

TABLE NAME	COLUMNS
airlines	airline_id, airline_code
airports	airport_id, airport_code
flight_delays	delay_id, operation_id, flight_date, airline_code, flight_number, departure_delay, arrival_delay, delay_carrier, delay_weather, delay_security, delay_late_aircraft
flight_operations	operation_id, schedule_id, flight_date, airline_id, airline_code, flight_number, actual_departure, actual_arrival, actual_duration, taxi_out, taxi_in, wheels_off, wheels_on, air_time, cancelled, diverted
flight_schedules	schedule_id, airline_id , airline_code , flight_number, origin_id, origin_code ,destination_id , destination_code, scheduled_departure, scheduled_arrival, scheduled_duration, distance

The database schema below explains our structure of the flight dataset, specifying the tables and their columns. We used this schema to construct queries and evaluate provenance systems:

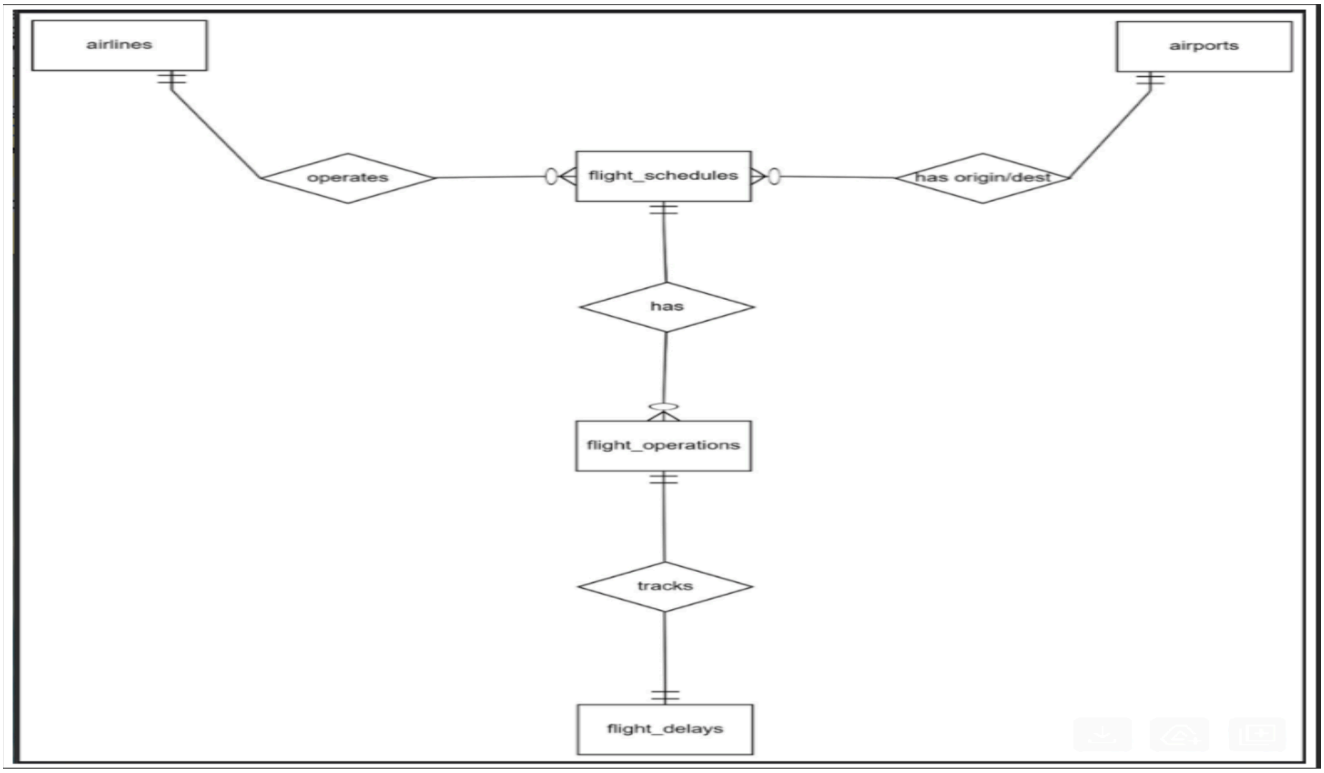


Fig: Entity Relationship Diagram

Key Features:

1. **Real-World Attributes**: The dataset includes detailed flight information such as flight numbers, airports, schedules, delays, and airline codes. These support queries involve join aggregations and selections.
2. **Scalability Testing**:
    - Small: Tests baseline performance with low data volume.
    - Medium: Tests system efficiency under moderate workloads.
    - Large: Tests high-volume real-world scenarios.
3. **Data Distributions**: Realistic patterns, such as peak-time delays and mixed airport traffic, add practical relevance.
4. **Provenance Challenges**:
    - **Joins**: Tracing relationships between flights, airlines, and airports.
    - **Aggregations**: Summarizing delays or flight counts.
    - **Selections**: Filtering conditions like delays exceeding 30 minutes.

This allowed us to comprehensively analyze system scalability, robustness, and applicability to real-world use cases, such as airline operations and regulatory compliance.

System Configuration

We conducted the experiments in a controlled environment with the following setup:

1. **Operating System**: Ubuntu 22.04 (64-bit).
2. **Virtualization Platform**: Oracle VirtualBox
3. **Database Management System**: PostgreSQL.
4. **Hardware**:
    - **Memory**: 80 GB.
    - **Processors**: 4 cores.

This configuration ensured compatibility with both systems and provided a realistic test environment for evaluating performance.

PostgreSQL Optimization

To handle the workload of large datasets and complex queries, we used PostgreSQL with the following adjustments:

1. **Increased** effective_cache_size **(4GB → 40GB)**: Enhanced query planning by better utilizing memory for caching, reducing disk I/O.
2. **Expanded** shared_buffers **(128MB → 4GB)Raised** temp_buffers **(8MB → 4GB)**: Allowed temporary tables to remain in memory, improving processing speed **Increased** work_mem **(4MB → 16GB)**:

    These adjustments were important for us to prevent bottlenecks and ensure the systems performed under optimal conditions. By adjusting the environment to our use case's experimental workload, we compared ProvSQL and GProM fairly.

Methodology -

Query Test Suite

To compare the provenance capabilities of ProvSQL and GProM, we have 21 queries that try to mirror real-world operations. The queries were made to test performance, scalability, and support for different types of provenance: WHY, WHERE, and HOW.

Key Query Types:

1. **Basic Selections**: Direct data retrieval will start our baseline performance.
    - Example: Q1 gets all rows from the flight_schedules table.
2. **Joins**: Combining data across tables to test tracking in interconnected datasets.
    - Examples: Q2 joins flight_schedules and airlines; Q3 calculates delay statistics through multi-table joins.
3. **Aggregations**: Summarizing data using functions like SUM, COUNT, and AVG to evaluate provenance in aggregate results.
    - Examples: Q3 computes total delays; Q19 calculates average taxi-out times.
4. **Grouping**: Grouping data by attributes to assess provenance handling in grouped contexts.
    - Examples: Q12 summarizes delays by airline and date; Q18 counts diverted flights by origin airport.
5. **Projections**: Selecting specific columns to test provenance tracking in partial datasets.
    - Example: Q14 extracts origin and destination codes for selected flights.
6. **Complex Combinations**: Mixing joins, filters, aggregations, and projections to test the systems' ability to handle multifaceted queries.
    - Examples: Q8 retrieves flights with delays over 300 minutes; Q21 computes average arrival delays per destination.

Representative Queries:

- **Q3**: Calculates multiple delay types (departure, arrival, carrier, weather) for airline AA using multi-table joins and grouping.
  Purpose: Tests provenance in complex aggregations.

- **Q8**: Retrieves flights with delays exceeding 300 minutes, grouped by origin and airline.
  Purpose: Evaluate provenance in grouped filtering logic.
- **Q18**: Counts and ranks diverted flights by origin airport.
  Purpose: Tests provenance for grouped and ordered summaries.
- **Q21**: Computes average arrival delays per destination and ranks results.
  Purpose: Assesses provenance tracking with aggregations and ranking.

Query Execution Goals

1. **Performance Benchmarking**: Measure execution times across Small, Medium, and Large datasets to analyze scalability and efficiency.
2. **Provenance Tracking**: Evaluate how effectively each system manages, represents, and retrieves provenance information.
3. **System Comparison**: Identify strengths and limitations in handling operations like joins, aggregations, and groupings.
4. **Real-World Applicability**: Ensure queries reflect industry scenarios such as aviation analytics, logistics, and compliance auditing.

List of All 21 Queries

QUERY CODE	QUERY	PURPOSE
Q1	SELECT * FROM flight_schedules;	• Retrieves all records from the flight_schedules table. • **Purpose:** Basic selection operation to test baseline query performance.
Q2	SELECT fs.flight_number, a.airline_code FROM flight_schedules fs JOIN airlines a ON fs.airline_code = a.airline_code LIMIT 5;	• Fetches flight numbers and airline codes by joining flight_schedules and airlines. • **Purpose:** Tests basic join operations with a limit clause.
Q3	SELECT a.airline_code, SUM(fd.departure_delay) AS total_departure_delay, SUM(fd.arrival_delay) AS total_arrival_delay, SUM(fd.delay_carrier) AS total_carrier_delay, SUM(fd.delay_weather) AS total_weather_delay	• Aggregates delays by airline for AA. • **Purpose:** Tests complex multi-join queries with grouping and aggregations.

	FROM flight_delays fd JOIN flight_operations fo ON fd.operation_id = fo.operation_id JOIN airlines a ON fo.airline_id = a.airline_id WHERE a.airline_code = 'AA' GROUP BY a.airline_code;	
Q4	SELECT a.airline_code, SUM(fs.scheduled_duration) AS total_scheduled_duration FROM flight_schedules fs JOIN airlines a ON fs.airline_id = a.airline_id WHERE a.airline_code = 'UA' GROUP BY a.airline_code;	• Computes the total scheduled duration for UA flights. • **Purpose:** Tests grouping and aggregation.

Q5	SELECT a.airline_code, COUNT(fo.operation_id) AS total_diverted_flights FROM flight_operations fo JOIN airlines a ON fo.airline_id = a.airline_id WHERE fo.diverted = 1 GROUP BY a.airline_code;	• Counts diverted flights per airline. • **Purpose:** Tests provenance for boolean filters with grouping.
Q6	SELECT fo.flight_number, fo.actual_duration, fo.taxi_out, fo.air_time, fo.taxi_in, a.airline_code FROM flight_operations fo JOIN airlines a ON fo.airline_id = a.airline_id WHERE fo.operation_id = 101;	• Fetches detailed flight operation data for a specific operation_id. • **Purpose:** Tests provenance for specific selections.
Q7	SELECT fs.origin_code, SUM(fd.departure_delay) AS total_departure_delay, SUM(fd.arrival_delay) AS total_arrival_delay FROM flight_schedules fs JOIN flight_operations fo ON fs.schedule_id = fo.schedule_id JOIN flight_delays fd ON fo.operation_id = fd.operation_id GROUP BY fs.origin_code;	• Aggregates delays by origin airport. • **Purpose:** Tests grouping with summations.
Q8	SELECT fs.origin_code, a.airline_code, fd.flight_number, MAX(fd.departure_delay) AS max_departure_delay, MAX(fd.arrival_delay) AS max_arrival_delay FROM flight_delays fd JOIN flight_operations fo ON fd.operation_id = fo.operation_id JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id JOIN airlines a ON fo.airline_id = a.airline_id GROUP BY fs.origin_code, a.airline_code, fd.flight_number HAVING MAX(fd.departure_delay) > 300 OR MAX(fd.arrival_delay) > 300;	• Fetches flights with extreme delays by airline and origin. • **Purpose:** Tests grouping with conditions and MAX aggregations.

Q9	SELECT fs.destination_code, a.airline_code,	• Counts cancellations per destination and

	COUNT(fo.cancelled) AS total_cancellations FROM flight_operations fo JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id JOIN airlines a ON fo.airline_id = a.airline_id WHERE fo.cancelled = 1 GROUP BY fs.destination_code, a.airline_code;	airline. ● **Purpose:** Tests provenance for conditional aggregations.
Q10	SELECT a.airline_code, SUM(fo.air_time) AS total_air_time, SUM(fs.distance) AS total_distance FROM flight_operations fo JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id JOIN airlines a ON fo.airline_id = a.airline_id GROUP BY a.airline_code;	● Aggregates total airtime and distance by airline. ● **Purpose:** Tests provenance for multiple aggregations.
Q11	SELECT fd.flight_number, fs.origin_code, fs.destination_code, fd.departure_delay, fd.arrival_delay FROM flight_delays fd JOIN flight_operations fo ON fd.operation_id = fo.operation_id JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id WHERE fs.origin_code = 'JFK' AND fs.destination_code = 'LAX';	● Fetches flight delays for flights from JFK to LAX. ● **Purpose:** Tests provenance for multiple filters.
Q12	SELECT a.airline_code, fd.flight_date, SUM(fd.delay_carrier) AS carrier_delay, SUM(fd.delay_weather) AS weather_delay, SUM(fd.delay_nas) AS nas_delay, SUM(fd.delay_security) AS security_delay, SUM(fd.delay_late_aircraft) AS late_aircraft_delay FROM flight_delays fd JOIN flight_operations fo ON fd.operation_id = fo.operation_id JOIN airlines a ON fo.airline_id = a.airline_id GROUP BY a.airline_code, fd.flight_date;	● Summarizes delay types by airline and flight date. ● **Purpose:** Tests provenance for grouped summaries.

Q13	SELECT airline_code, SUM(departure_delay + arrival_delay) AS total_delay FROM flight_delays WHERE airline_code = 'AA' GROUP BY airline_code;	● Computes total delay for airline AA. ● **Purpose:** Tests simple grouping and summations.
Q14	SELECT fs.origin_code, fs.destination_code FROM flight_schedules fs JOIN airlines a ON fs.airline_id = a.airline_id WHERE a.airline_code = 'UA';	● Fetches routes for airline UA. ● **Purpose:** Tests basic joins and filtering.
Q15	SELECT fo.flight_number, fo.flight_date, fo.cancelled, fd.delay_weather, fd.delay_carrier FROM flight_operations fo JOIN flight_delays fd ON fo.operation_id = fd.operation_id WHERE fo.cancelled = 1;	● Fetches details of cancelled flights. ● **Purpose:** Tests provenance for boolean filters.

Q16	SELECT fd.flight_number, fd.departure_delay, fd.delay_carrier, fd.delay_weather, fd.delay_nas FROM flight_delays fd WHERE fd.departure_delay > 300;	• Fetches flights with significant departure delays. • **Purpose:** Tests provenance for large value filtering.
Q17	SELECT fd.flight_number, fs.origin_code, fs.destination_code, fd.departure_delay, fd.arrival_delay FROM flight_delays fd JOIN flight_operations fo ON fd.operation_id = fo.operation_id JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id WHERE fd.departure_delay > 300 OR fd.arrival_delay > 300;	• Fetches flights with significant delays by origin and destination. • **Purpose:** Tests filtering with multiple conditions.

Q18	SELECT fs.origin_code, COUNT(fo.operation_id) AS total_diverted_flights FROM flight_operations fo JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id WHERE fo.diverted = 1 GROUP BY fs.origin_code ORDER BY total_diverted_flights DESC;	• Counts and ranks diverted flights by origin. • **Purpose:** Tests provenance for grouped and ordered aggregations.
Q19	SELECT fs.origin_code, AVG(fo.taxi_out) AS avg_taxi_out FROM flight_operations fo JOIN flight_schedules fs ON fo.schedule_id = fs.schedule_id GROUP BY fs.origin_code ORDER BY avg_taxi_out DESC;	• Calculates average taxi-out time by origin. • **Purpose:** Tests averaging and ordering.
Q20	SELECT fs.origin_code, SUM(fd.delay_weather) AS total_weather_delay FROM flight_schedules fs JOIN flight_operations fo ON fs.schedule_id =	• Aggregates weather delays by origin airport. **Purpose:** Tests provenance for grouped sums with ordering.

	fo.schedule_id JOIN flight_delays fd ON fo.operation_id = fd.operation_id GROUP BY fs.origin_code ORDER BY total_weather_delay DESC;	
Q21	SELECT fs.destination_code, AVG(fd.arrival_delay) AS avg_arrival_delay FROM flight_schedules fs JOIN flight_operations fo ON fs.schedule_id = fs.schedule_id JOIN flight_delays fd ON fo.operation_id = fd.operation_id GROUP BY fs.destination_code ORDER BY avg_arrival_delay DESC;	• Calculates average arrival delay per destination. **Purpose:** Tests grouping with averages and ordered results.

Provenance Types -

1. WHERE Provenance

- **Purpose**: Answers questions like, *"Which rows in the input tables contributed to this result?"* WHERE provenance is crucial for auditing, debugging, and verifying data lineage in regulatory compliance and traceability contexts.
- **Evaluation**: Queries such as Q2 and Q11 assessed WHERE provenance. For example, Q11 traced the source tuples contributing to flights from JFK to LAX by finding origin and destination codes in relevant tables.
- **Significance**: Important for ensuring data traceability and accountability in auditing and data governance industries.

2. WHY Provenance

- **Purpose**: Addresses questions like, *"Why does this result exist?"* WHY provenance is vital for understanding the rationale behind outputs, making it key in scientific research and decision-making.
- **Evaluation**: Queries like Q3 and Q8, which involve aggregations and grouping, were ideal for assessing WHY provenance. For example, Q3 explained how total delay values for airline AA were derived from individual flight delays.

- **Significance**: Critical for scenarios where understanding the reasoning behind results is necessary, such as impact analysis and research validation.

3. HOW Provenance

- **Purpose**: Answers questions like, *"What transformations produced this result?"* HOW provenance provides a step-by-step explanation of the data pipeline.
- **Evaluation**: Complex queries such as Q12 and Q18 tested HOW provenance. For example, Q12 involved transformations like joining, grouping, and summing to compute delay summaries by flight date.
- **Significance**: Essential for debugging pipelines, optimizing query performance, and ensuring reproducibility in analytics and engineering workflows.

Results and Analysis

This section analyzes the performance, feature support, resource utilization, and query response times of ProvSQL and GProM across three dataset sizes: Small, Medium, and Large. The results highlight each system's strengths and limitations.

Performance Metrics Across Dataset Sizes

Small Dataset (500,000 rows):

- **ProvSQL**:
    - Performed well on simpler queries (e.g., Q2: 2.83 ms, Q6: 2.94 ms).
    - Struggled with complex queries like Q3 and Q7, with execution times of 34,913 ms and 172,790 ms, respectively.
    - Failed to process Q8 and queries Q18–Q21, reflecting limitations in handling certain scenarios.
- **GProM**:
    - Consistently faster for most queries, with execution times as low as 1.51 ms (Q1) and 303 ms (Q13).
    - Processed queries like Q8 (23,625 ms) and Q18–Q21, where ProvSQL failed, demonstrating broader query support.

Medium Dataset (1,000,000 rows):

- **ProvSQL**:
    - Execution times increased proportionally, with inefficiencies in complex queries (e.g., Q3: 38,097 ms, Q7: 235,834 ms).
    - Performed better than GProM in a few cases, such as Q5 (360 ms vs. 339 ms).
    - Unable to process Q8 and Q18–Q21.

- **GProM**:
    - Maintained scalability with smaller increases in execution times for queries like Q10 (6,634 ms) and Q12 (232,416 ms).
    - Handled all queries, including Q18–Q21, showcasing superior adaptability for complex queries.

Large Dataset (3,000,000 rows):

- **ProvSQL**:
    - Struggled significantly, with execution times as high as 1,645,004 ms (Q7) and 2,308,662 ms (Q10).
    - Unable to process Q8 and Q18–Q21, emphasizing its scalability limitations.
- **GProM**:
    - Scaled effectively, with reasonable execution times for queries like Q3 (5,130 ms) and Q21 (40,454 ms).
    - Successfully processed all queries, including the most complex ones.

Comparative Analysis of Feature Support

1. **Query Types**:
    - **ProvSQL**: This is best suited for basic selection, projection, and single-table aggregations (e.g., Q1, Q4) and is limited in handling complex joins and advanced groupings.
    - **GProM**: Broader support for multi-join queries (e.g., Q3, Q10) and ranking-based aggregations (e.g., Q18, Q19).
2. **Provenance Types**:
   Both systems support WHERE, WHY, and HOW provenance. GProM excelled in visualizing complex provenance graphs, especially for multi-join operations.
3. **System Capabilities**:
    - **ProvSQL**: Robust semiring-based mathematical representation for specific workflows but less flexible for advanced queries.
    - **GProM**: Dynamic query rewriting and provenance game graphs offer better adaptability for complex use cases.

Resource Utilization Patterns

1. **Memory Usage**:
    - **ProvSQL**: High memory consumption for large datasets due to semiring-based annotations.
    - **GProM**: More efficient memory use, leveraging dynamic annotation propagation.

2. **CPU Usage**:
    - **GProM**: Demonstrated better CPU utilization during complex queries, benefiting from its query rewriting techniques.

Query Response Time Comparisons

- **ProvSQL**:
  Faster for basic queries (e.g., Q1, Q2) but much slower for queries involving multiple joins, aggregations, or large datasets.
  Failed to handle several queries in the medium and large datasets.
- **GProM**:
  Consistently faster for most query types, especially on medium and large datasets, showcasing superior scalability and optimization.

Query Analysis -

This section evaluates the performance of ProvSQL and GProM across different query types, focusing on key trends and insights.

General Observations:

1. **Simple Queries (Q1, Q2, Q6):**
    - Both systems perform well on basic selection and filtering queries.
    - ProvSQL is much faster for specific cases (e.g., Q2, Q6).
    - GProM shows slightly higher execution times for simple queries on smaller datasets but becomes better as dataset size increases.
2. **Complex Aggregations and Joins (Q3, Q7, Q10):**
    - GProM consistently outperforms ProvSQL for queries involving multiple joins and aggregations, showing a 33x–65x speed advantage in Q3 and Q10 for smaller datasets and scaling up to 119x faster for larger datasets.
    - ProvSQL's execution times increase sharply with dataset size, highlighting scalability challenges.
3. **Advanced Queries (Q8, Q18–Q21):**
    - ProvSQL does not support advanced queries involving complex filtering, grouping, and ranking.
    - On the other hand, GProM handles these queries efficiently, demonstrating robust feature support and scalability even on large datasets.
4. **Grouped and Ranked Aggregations (Q12, Q19, Q21):**
    - GProM is better at handling grouped summaries and ranking operations, scaling effectively with dataset size.

    - For example, in Q12, GProM is 10–18x faster than ProvSQL, with the gap becoming bigger as datasets grow.
5. **Real-World Queries (Q5, Q9, Q15):**
    - GProM shows better scalability for queries involving counting and filtering (e.g., Q5: counting diverted flights), with a consistent 30–50% performance advantage.
    - ProvSQL remains competitive for smaller datasets but struggles with scalability for larger workloads.

Key Insights:

- ProvSQL's semiring-based mathematical framework ensures accurate provenance representation but is limited in scalability and feature support.
- GProM provides comprehensive support for advanced queries involving grouping, ranking, and complex aggregations, with execution times increasing slowly as dataset size grows.

Specific Strengths and Limitations

ProvSQL -

Strengths:

1. Efficient for basic queries and small datasets.
2. Provides rigorous, semiring-based mathematical provenance representation.

Limitations:

1. Poor scalability for large datasets and complex queries.
2. Limited support for advanced query types, including ranking and group operations.

GProM -

Strengths:

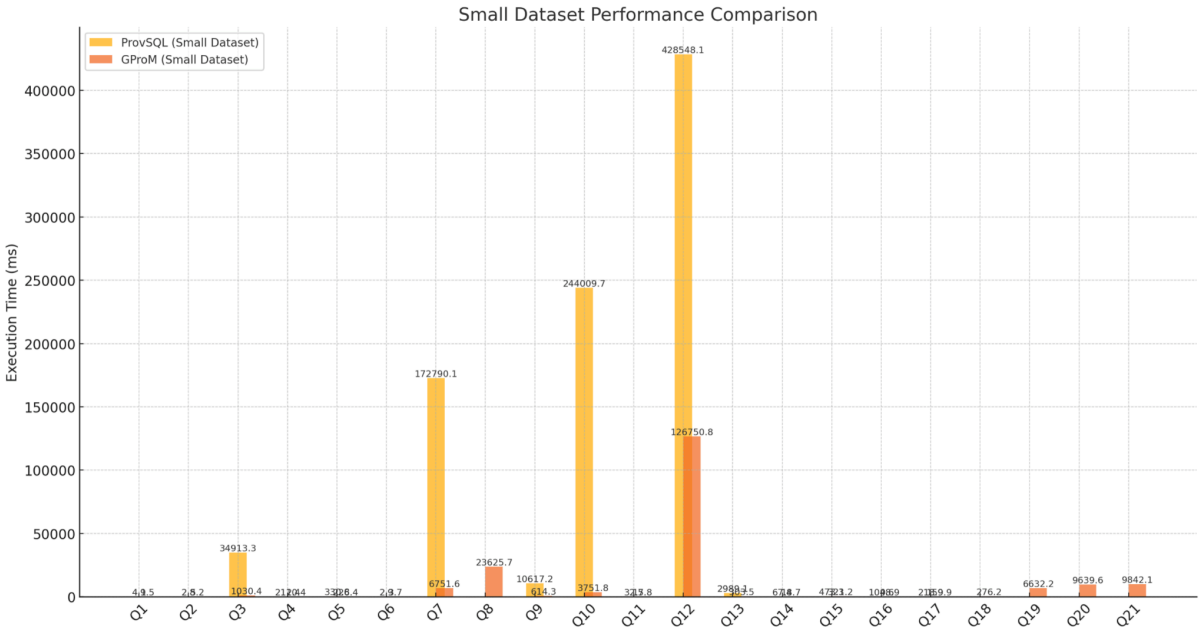
1. Better scalability and efficiency for complex and large-scale queries.
2. Comprehensive query support, including multi-join operations, advanced aggregations, etc.
3. Dynamic query rewriting and provenance game graphs enhance adaptability and visualization.

Limitations:

1. Slightly slower execution times for simple queries on smaller datasets (e.g., Q2, Q6)

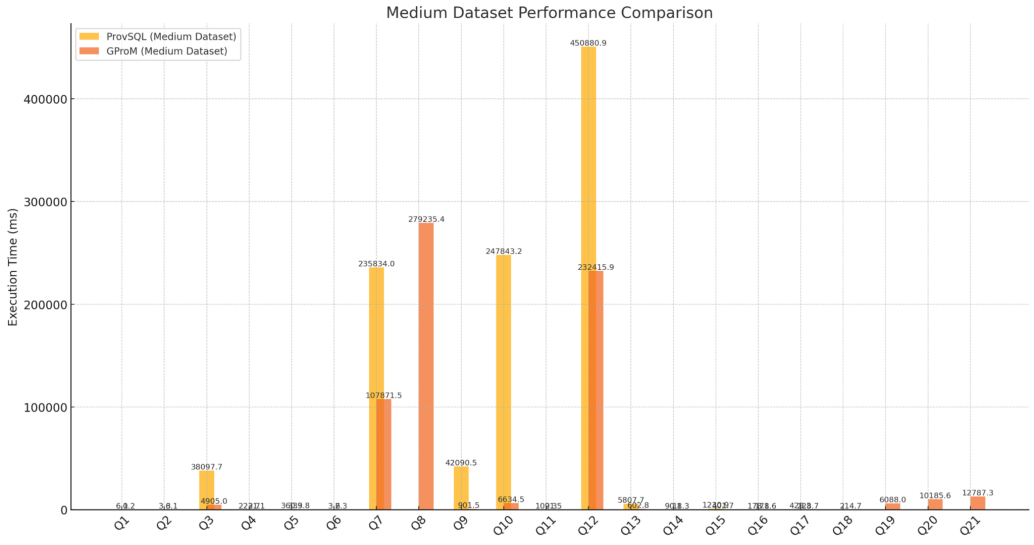
Summary of Comparative Analysis

- **Performance**: GProM significantly outperforms ProvSQL for complex and large-scale queries, with speed advantages of up to 119x in some cases.
- **Scalability**: GProM scales effectively with dataset size, maintaining reasonable execution times even for advanced queries. ProvSQL struggles with scalability and fails to process certain queries (e.g., Q18–Q21).
- **Feature Support**: GProM supports a broader range of SQL features, including ranking and grouped aggregations, making it more versatile for real-world applications.
- **Provenance Capabilities**: GProM's comprehensive provenance tracking and visualization make it ideal for advanced data analysis. ProvSQL, while suitable for simpler tasks, is less adaptable to complex scenarios.



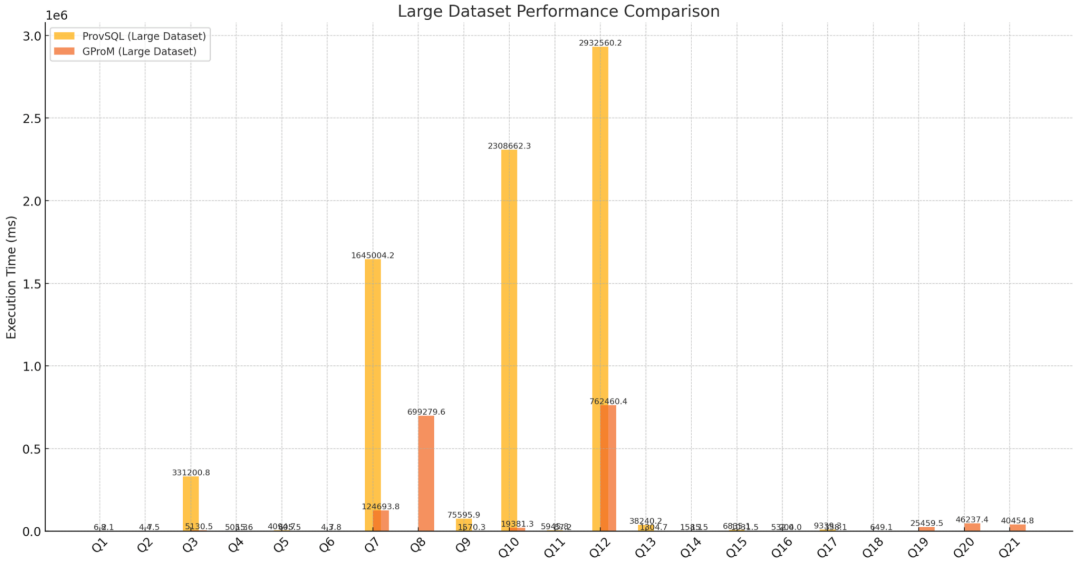
Small Dataset Performance Comparison

- **Summary:**
    1. The graph above shows the execution times (in milliseconds) for ProvSQL and GProM across 21 queries for a small dataset of 500,000 rows.
    2. GProM demonstrates consistently lower execution times for most queries, highlighting its efficiency even for smaller workloads.



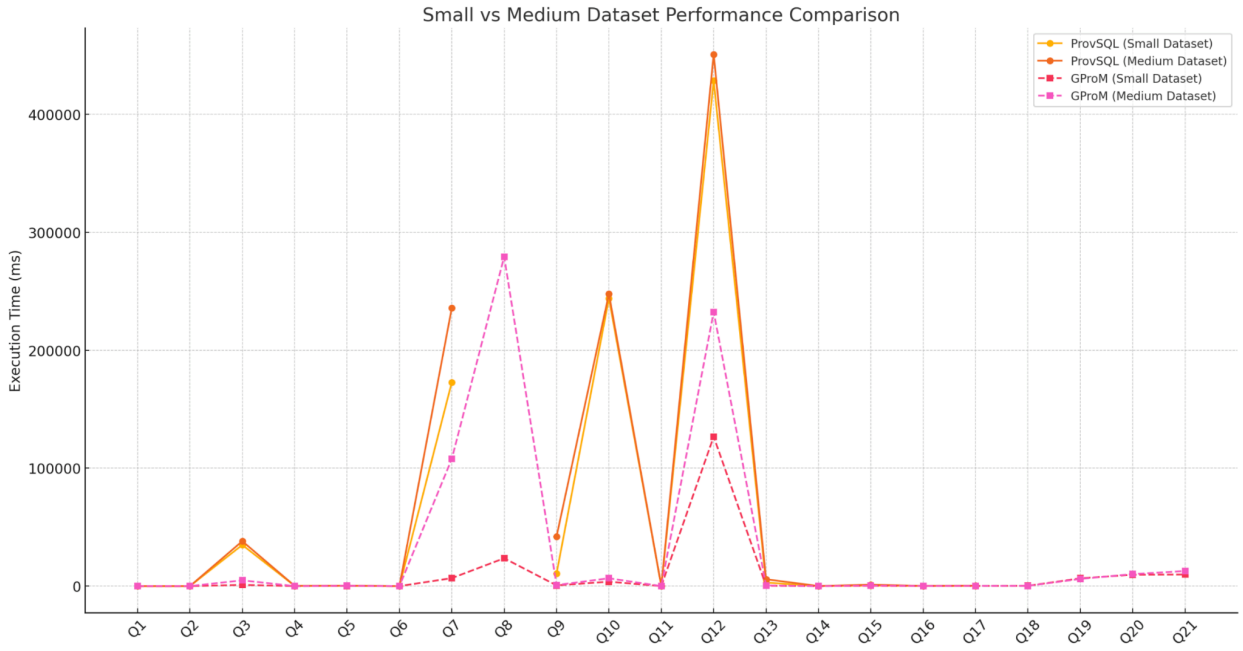
Summary:

- The above graph presents the performance of both systems for a medium-sized dataset of 1,000,000 rows.
- The gap between GProM and ProvSQL widens as query complexity increases, particularly for aggregations and joins.



Summary:

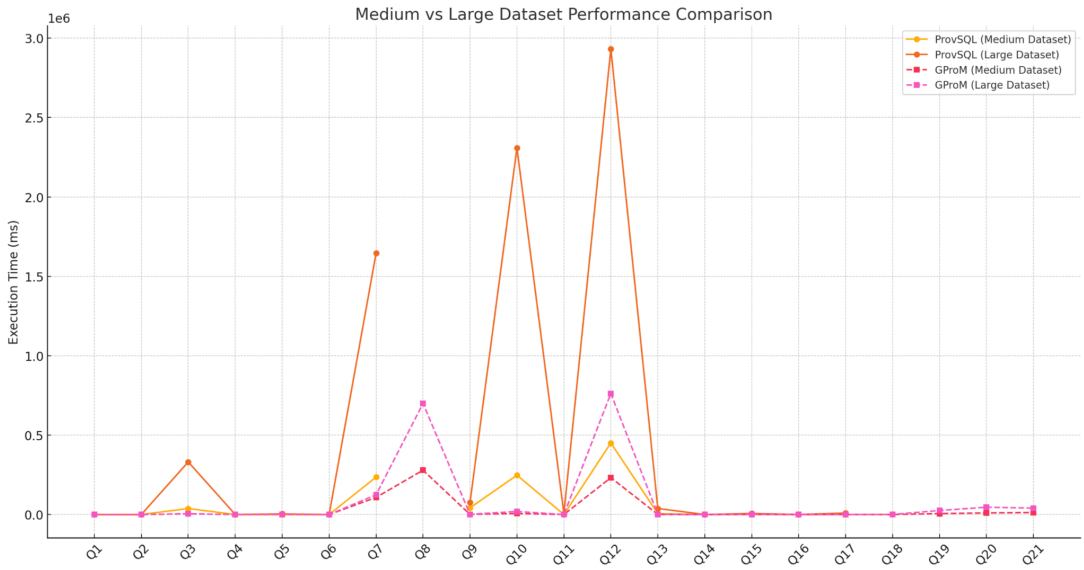
- The graph compares the execution times of ProvSQL and GProM for a large dataset of 3,000,000 rows.
- ProvSQL struggles with scalability, while GProM maintains reasonable execution times across all queries.



Small vs Medium Dataset Performance Comparison

Summary: This graph compares ProvSQL and GProM's execution times for small (500,000 rows) and medium (1,000,000 rows) datasets.

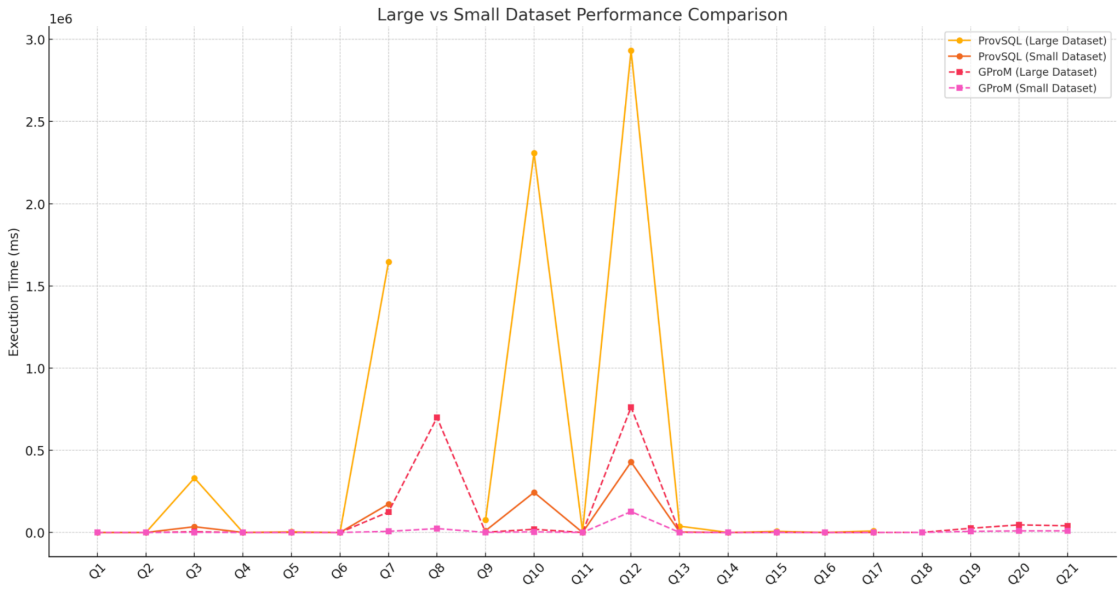
Key Insight: Execution times increase with dataset size, but GProM maintains better scalability and handles complex queries like Q3, Q7, and Q12 significantly faster than ProvSQL. ProvSQL struggles with query scalability, particularly in aggregation-heavy queries



Medium vs Large Dataset Performance Comparison

Summary: The graph illustrates the performance differences between medium (1,000,000 rows) and large (3,000,000 rows) datasets.

Key Insight: GProM demonstrates linear scalability with dataset size, whereas ProvSQL shows a lot of growth in execution times for complex queries. GProM processes queries Q18–Q21 efficiently, which ProvSQL cannot handle.



Large vs Small Dataset Performance Comparison

- **Summary**: This graph compares execution times between large (3,000,000 rows) and small (500,000 rows) datasets.
- **Key Insight**: ProvSQL's execution times grow greatly with dataset size, highlighting its scalability challenges. GProM consistently outperforms ProvSQL across all query types, maintaining reasonable execution times even for large datasets.

Conclusion

This study provided an in-depth analysis of two provenance systems, ProvSQL and GProM, evaluating their performance, scalability, and feature support across varying dataset sizes. The results clearly demonstrate each system's unique strengths and limitations, as well as the challenges encountered during their deployment and usage.

Key Findings:

1. **Performance and Scalability:**
    - GProM consistently outperformed ProvSQL in handling complex queries and larger datasets, exhibiting better scalability and lower execution times across all tested query types.
    - ProvSQL performed well for more straightforward queries on smaller datasets but struggled with complex operations involving joins, aggregations, and grouping, especially as dataset sizes increased.
2. **Feature Support:**
    - GProM demonstrated good support for all 3 tested provenance types and excelled in handling advanced query features such as nested aggregates and ordering with aggregates. Its ability to successfully execute all queries, including those unsupported by ProvSQL, highlights its versatility.
    - ProvSQL's WHY NOT provenance feature provided value for debugging and analyzing missing data, which GProM does not natively support.
    - **System Challenges:**
    - Both presented significant setup and configuration challenges for us as this is our first time using these systems, including dependency issues, architecture mismatches, and database integration issues. These obstacles were very time-consuming.
    - Large datasets in our experiment posed additional challenges for both systems, with memory bottlenecks and excessive execution times being our common issues. ProvSQL was particularly sensitive to these constraints due to its memory-intensive semiring-based representations.
    -
3. **Practical Implications:**
    - GProM's broader query support, better scalability, and better performance make it the preferred choice for general-purpose provenance use cases, particularly in large-scale or complex data environments.
    - ProvSQL's WHY NOT feature, while valuable in specific debugging scenarios, limits its general applicability due to its poor scalability and lack of advanced query support.

Lessons Learned:

1. The setup and configuration of provenance tools require significant database management and debugging knowledge.
2. Scalability remains critical for provenance systems, particularly when dealing with large datasets and complex queries.
3. Very good documentation is important to reduce the learning curve and improve accessibility for new users like us.

4. The choice of a provenance tool must align with specific use case requirements, considering factors such as dataset size, query complexity, and the desired type of provenance analysis.

Final Recommendation: Based on our evaluation, GProM is the stronger choice for general-purpose provenance tracking and analysis, offering better scalability, broader feature support, and faster query execution. ProvSQL, while limited in its overall capabilities, can serve as a useful tool for workflows that require mathematical rigor or WHY NOT provenance analysis in smaller datasets.

References-

1) Arab, B., Gawlick, D., Radhakrishnan, V., Guo, H., & Glavic, B. (2014). A Generic Provenance Middleware for Database Queries, Updates, and Transactions. *Proceedings of the 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*.
2) Senellart, P. (2018). ProvSQL: Provenance and Probability Management in PostgreSQL. *Proceedings of the VLDB Endowment*, 11(12), 2034-2037.
3) Arab, B., Feng, S., Glavic, B., Lee, S., Niu, X., & Zeng, Q. (2018). GProM: A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin*, 41(1), 51–62.
4) Senellart, P. (2018). ProvSQL: Provenance and Probability Management in PostgreSQL. *Proceedings of the VLDB Endowment*, 11(12), 2034-2037.