GProM - A Swiss Army Knife for Your Provenance Needs

Bahareh S. Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, Qitian Zeng
Illinois Institue of Technology

Abstract

We present an overview of GProM, a generic provenance middleware for relational databases. The system supports diverse provenance and annotation management tasks through query instrumentation*, i.e., compiling a declarative frontend language with provenance-specific features into the query language of a backend database system. In addition to introducing GProM, we also discuss research contributions related to GProM including the first provenance model and capture mechanism for transaction provenance, a unified framework for answering why- and why-not provenance questions, and provenance-aware query optimization. Furthermore, by means of the example of post-mortem debugging of transactions, we demonstrate how novel applications of provenance are made possible by GProM.*

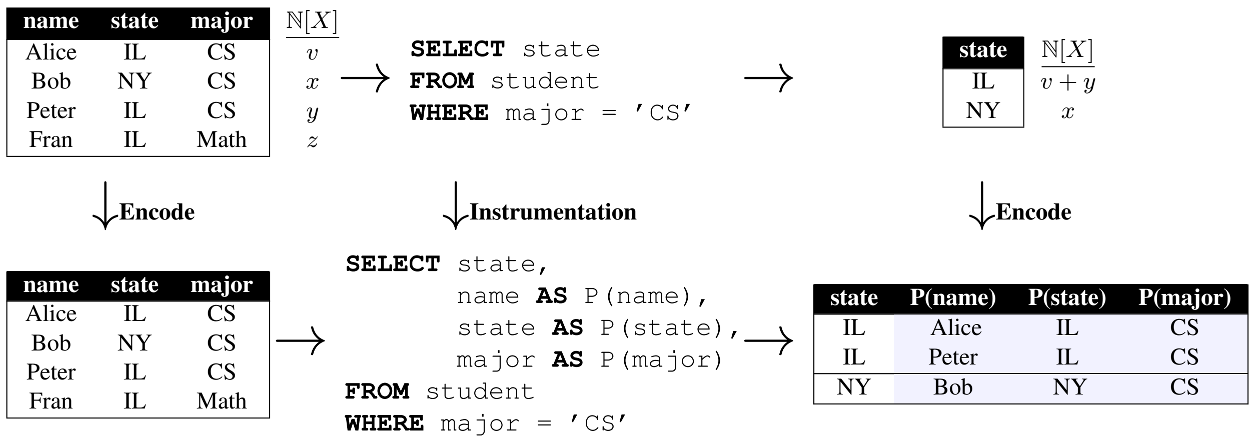
1 Introduction

Provenance, information about the origin of data and the queries and/or updates that produced it, is critical for debugging queries and transactions, auditing, establishing trust in data, and many other use cases. For example, consider a relation storing employee salaries. The relation is subjected to complex transactional updates such as calculating tax, applying tax deductions, multipling rates with working hours, and so on. How can we know whether the information in the current version of the relation is correct? If one employee's salary is incorrect, how do we know which update(s) or data caused that error? Data provenance, by providing a full record of the derivation history of data, makes it possible to identify the causes of such errors.

A persistent challenge in database provenance research has been to build efficient provenance-aware databases. That is, to design and implement systems that automatically capture provenance information for database operations and allow this information to be queried. In this work, we give an introduction to GProM (Generic Provenance Middleware), a system that enriches database backends with support for provenance. The system is available as open source software at `https://github.com/IITDBGroup/gprom`. At its core, GProM is a compiler that translates a frontend language (e.g., SQL with new language constructs for requesting and managing provenance) into queries expressed in the language of a database backend that generate a relational encoding of data annotated with provenance. Below we briefly discuss some of GProM's unique features.

- **Exploiting backend databases for provenance capture and storage**: GProM represents provenance in the data model of the backend database. To capture provenance for an operation, the system constructs a query expressed in the language of the database backend which returns this type of provenance encoding. This technique, which we refer to as *instrumentation*, enables us to exploit the advanced storage and query execution capabilities of modern database systems.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



name	state	major	$\mathbb{N}[X]$
Alice	IL	CS	v
Bob	NY	CS	x
Peter	IL	CS	y
Fran	IL	Math	z

```
SELECT state
FROM student
WHERE major = 'CS'
```

state	$\mathbb{N}[X]$
IL	$v + y$
NY	x

name	state	major
Alice	IL	CS
Bob	NY	CS
Peter	IL	CS
Fran	IL	Math

```
SELECT state,
       name AS P(name),
       state AS P(state),
       major AS P(major)
FROM student
WHERE major = 'CS'
```

state	P(name)	P(state)	P(major)
IL	Alice	IL	CS
IL	Peter	IL	CS
NY	Bob	NY	CS

Figure 1: Provenance instrumentation example: compute provenance polynomials for a query

- **On demand provenance capture for a large class of operations**: GProM supports provenance capture for queries, updates, and transactions. To the best of our knowledge it is the only system that can capture provenance for transactions. In contrast to many other systems which capture provenance eagerly for all operations no matter whether this provenance is needed or not, GProM only captures provenance if it is explicitly requested by a user or application.
- **Treating provenance requests as queries**: The user interacts with GProM through a declarative frontend language enriched with language constructs for requesting and managing provenance. Provenance requests are treated as queries which allows them to be combined with other language constructs. Thus, the full expressive power of the frontend language is available for querying provenance.
- **Low-overhead and non-invasive**: GProM was designed to minimize the performance impact for operations when no provenance is requested. Obviously, it would be impossible to reconstruct provenance for past operations unless some information is maintained. GProM relies on the temporal and auditing logging capabilities supported by many DBMS to capture sufficient information to be able to reconstruct provenance for past queries, transactions, and updates on demand. This approach has the advantage of being non-invasive, i.e., no changes to an application's SQL code are required to enable provenance capture.
- **Extensibility**: GProM was designed from the ground up with extensibility in mind - support for new provenance models, database backends, frontend languages, and optimizations can be added with ease. This has enabled us to support a large number of diverse provenance and annotation management tasks.

The remainder of this paper is organized as follows. We explain the instrumentation approach underlying GProM in Section 2. In Section 3, we give a more technical introduction to GProM. Section 4 covers major contributions to provenance research related to GProM. We discuss post-mortem transaction debugging, one of the novel applications of provenance made possible by GProM, in Section 5. We conclude in Section 6. The research on GProM is enabled by the many fundmental contributions to provenance research made by the database community. For reasons of space it is beyond the scope of this paper to acknowledge these important contributions. We refer the interested reader to one of the many excellent surveys on provenance [4–6, 8, 12].

2 Instrumentation - Exploiting a DBMS for Provenance Storage and Capture

The de facto standard for database provenance [9] is to model provenance as annotations on data and to define a query semantics that determines how annotations propagate. Under such a semantics, each output tuple t of a query Q is annotated with its provenance, i.e., a combination of input tuple annotations that explains how these inputs were used by Q to derive t. For instance, using provenance polynomials, each tuple is annotated with a, typically unique, variable representing this tuple. Under this model, annotations are propagated such that every query result tuple is annotated with a polynomial over the variables representing the input tuples in the output's provenance. The addition and multiplication operations in such polynomials encode how these inputs have been combined to derive the output. Addition represents alternative use of inputs (e.g., union or projection) and multiplication represents conjunctive use (e.g., join). Relations annotated with provenance polynomials are a specific type of K-relations, relations where tuples are annotated with elements from a commutative semiring such as the semiring of provenance polynomials (denoted as $\mathbb{N}[X]$). Relational algebra over K-relations is defined based on the addition and multiplication operations of the semiring.

For example, consider a query listing all states that have CS students. The query and example $\mathbb{N}[X]$-relations encoding the input and output are shown at the top of Figure 1. The query result `(IL)` is annotated with $v + y$ which indicates that this tuple is part of the result as long as either v or x exist in the input (students Alice or Peter). As we will discuss in Section 4.1, the semiring provenance model which originally was defined for queries can be extended to also support transactional updates. GProM targets any type of provenance or other information that can be modelled as annotations. Current database systems do not natively support the propagation of annotations through operations. There are two approaches for making a database system provenance-aware. Either we extend the system's query execution engine to support these features natively or we encode provenance annotations using the data model supported by the database and *instrument* operations to propagate provenance annotations. GProM and many other database provenance systems such as Perm [7], LogicBlox, Orchestra, and ExSPAN apply the second approach. Using a relational encoding of provenance annotations, these systems compile queries with annotated semantics into relational queries that produce this encoding of provenance annotations. We refer to this reduction from annotated to standard relational semantics as *instrumentation*. The instrumentation approach can either be implemented as a compilation process in a middleware application or as a query rewrite layer within a DBMS that instruments queries before they are passed to the system's optimizer (e.g., the *Perm* system [7] is an extension of PostgreSQL with support for capturing and querying provenance). In GProM we have opted for a middleware implementation to be able to support multiple database backends.

An example of instrumentation is shown at the bottom of Figure 1. The input query with annotated semantics is instrumented to produce a relational encoding of provenance polynomial annotations. In this example, we use an encoding pioneered in Perm [7] which represents a tuple t annotated with polynomial k as follows. The polynomial is refactored into a sum of products where variables in each monomial (individual product in the sum) are ordered based on the occurrence of the relations in the query. A polynomial normalized in this fashion is then encoded as a set of tuples where each tuple in such a set represents one monomial. A variable is represented by the values of the tuple annotated with this variable in the input. Here P denotes a renaming function which is used to create names for attributes that store provenance. For instance, an attribute `student.name` would be renamed as `prov_student_name`. Note that this is only one of the representations supported in GProM, e.g., the user can alternatively request the system to use tuple identifiers to encode variables. Furthermore, annotations are generated on the fly in this example. This, however, is not a requirement. The instrumentation approach works perfectly well for inputs which have provenance associated with them.

3 System Overview

We now give a more detailed and technical overview of GProM. Figure 2 shows the high-level architecture of GProM. A user interacts with the system by sending a query written in a frontend language using one of the system's client interfaces (e.g., using a CLI). Frontend languages are declarative query languages that have been enriched with new language constructs for requesting and querying provenance. A frontend-specific parser translates an incoming query into a more suitable internal representation, e.g., relational algebra. The result is then

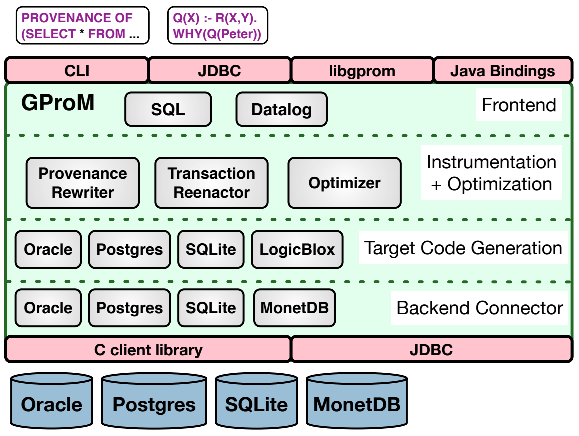


Figure 2: GProM Architecture

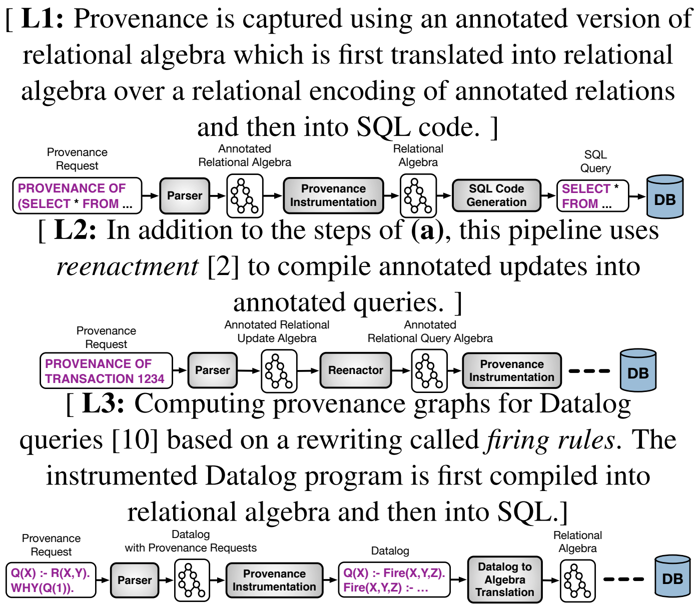

[L1: Provenance is captured using an annotated version of relational algebra which is first translated into relational algebra over a relational encoding of annotated relations and then into SQL code.]
Provenance Request
Annotated Relational Algebra
Relational Algebra
SQL Query
PROVENANCE OF (SELECT * FROM ...
Parser
Provenance Instrumentation
SQL Code Generation
SELECT * FROM ...
DB
[L2: In addition to the steps of (a), this pipeline uses reenactment [2] to compile annotated updates into annotated queries.]
Provenance Request
Annotated Relational Update Algebra
Annotated Relational Query Algebra
PROVENANCE OF TRANSACTION 1234
Parser
Reenactor
Provenance Instrumentation
DB
[L3: Computing provenance graphs for Datalog queries [10] based on a rewriting called firing rules. The instrumented Datalog program is first compiled into relational algebra and then into SQL.]
Provenance Request
Datalog with Provenance Requests
Datalog
Relational Algebra
Q(X) :- R(X,Y). WHY(Q(1)).
Parser
Provenance Instrumentation
Q(X) :- Fire(X,Y,Z). Fire(X,Y,Z) :- ...
Datalog to Algebra Translation
DB

Figure 3: Instrumentation pipelines for provenance: **(a) L1:** SQL queries, **(b) L2:** transactions, **(c) L3:** Datalog

processed by one or more instrumentation components which translate parts of a query containing provenance features by rewriting such parts to generate a relational encoding of provenance. Instrumentation is typically broken down into a multi-step compilation process which has the advantage that components implementing a compilation step can be utilized for multiple provenance tasks. GProM also features a generic optimizer that can be applied to any such compilation step (see Section 4.3). The output of instrumentation is then processed by a backend specific code generation module, e.g., translating relational algebra into Postgres's SQL dialect. The generated code is sent to the backend for execution using a backend connector. Connectors either use a native C-library or a JDBC driver to connect to the backend system. GProM was designed from the ground up to be as modular and extensible as possible. Most components of the system including the frontend parser, instrumentation and optimization components, code generators, and connectors are pluggable.

3.1 Frontends

So far we have implemented two frontends in GProM: 1) an SQL dialect with provenance features and 2) a Datalog frontend with support for requesting explanations (provenance) for existing and missing query answers. Importantly, the SQL dialect also supports other types of annotations such as temporal data and uncertainty. Our general philosophy in designing these language extensions was to make provenance requests proper query constructs that can be used in almost any place where regular queries are allowed. Importantly, this enables the full expressive power of the frontend language to be used for querying provenance information.

3.2 Instrumentation Pipelines

To implement a particular provenance task, e.g., compile a provenance request written in GProM's SQL dialect into Postgres's SQL dialect, the instrumentation components of GProM are arranged into a so-called *instrumentation pipeline*. Figure 3 shows some of the pipelines currently supported by GProM:

- **L1. Provenance for SQL Queries:** The pipeline from Figure **??** generates a relational encoding of provenance annotations such as the one shown in Figure 1.

- **L2. Provenance for Transactions:** Figure **??** shows a pipeline that retroactively captures provenance for transactions. In addition to the steps from Figure **??**, this pipeline uses a compilation step called *reenactment*. Reenactment translates transactional histories with annotated semantics into equivalent temporal queries with annotated semantics. We will discuss this pipeline in more detail in Section 4.1.
- **L3. Provenance for Datalog:** This pipeline (Figure **??**) produces provenance graphs that explain which successful and failed rule derivations of a Datalog program are relevant for (not) deriving a (missing) query result. A provenance request is compiled into a program that computes the edge relation of the provenance graph. This program is then translated into SQL. See Section 4.2 for a more detailed discussion.

Note that a frontend may support multiple pipelines. User requests written in the language of the frontend are automatically dispatched to the pipeline that is responsible for handling this type of request. For instance, the SQL frontend uses several pipelines including the pipelines L1 and L2 described above. If a user requests the provenance for a query then this request will be handled by pipeline L1 whereas if the user requests the provenance of a transaction then this request will be dispatched to pipeline L2.

3.3 Backends and Client Interfaces

GProM can be accessed through its native commandline shell (CLI), through a library (libgrom), using a Java API, or through the system's JDBC driver which wraps a vendor specific JDBC driver. GProM supports code generation for the SQL dialects of Oracle, Postgres, and SQLite as well as for LogiQL, LogicBlox's Datalog dialect. Backend connectors have been implemented for Oracle, Postgres, SQLite, and MonetDB.

4 Research Contributions

In addition to building a platform to support diverse provenance needs, our work on GProM has laid out the basis for future research related to data provenance. At the same time, building GProM required us to cover new ground in provenance research. In the following, we cover three major research contributions: tracking provenance of transactions, unifying why- and why-not provenance, and provenance-aware query optimization.

4.1 Provenance for Transactions and Reenactment

One major limitation of database provenance approaches is their lack of support for tracking provenance of transactional updates. This has prevented the use of provenance for applications such as debugging of transactions and auditing. For instance, consider the following scenario. A company uses a database to store mission-critical data and an attacker has compromised one of the database user accounts. Provenance for transactions would allow us to determine what data was accessed by the attacker through this account. Furthermore, it would allow us to determine what data was affected directly or indirectly by updates run by the compromised account (e.g., a reporting query returns an incorrect result because of the attacker has modified data). To address this shortcoming, we have developed *MV-semirings* (multi-version semirings), the first provenance model for transactions, and reenactment, a technique for retroactively capturing the provenance of past transactions using queries.

The MV-semiring model extends the semiring annotation framework [9] to account for tuple derivations under transactional updates [1, 2]. For any semiring $\mathcal{K}$, we can construct an MV-semiring $\mathcal{K}^{\nu}$. For instance, $\mathbb{N}[X]^{\nu}$ is the MV-version of the provenance polynomial semiring $\mathbb{N}[X]$. An annotation from an MV-semiring $\mathcal{K}^{\nu}$ is a symbolic expression over elements from $\mathcal{K}$ recording the derivation history of a tuple. These expressions use *version annotations* to enclose part of the provenance of a tuple to encode that the tuple version corresponding to this part of the provenance was processed by a certain update at a certain time. A version annotation $X^{id}_{T,\nu}(k)$ denotes that an operation of type X (update U, insert I, delete D, or commit C) that was executed at time

$\nu - 1$ (we assume a totally ordered time domain that is used to identify versions) by transaction T affected a previous version of a tuple with identifier id and previous provenance k. The nesting of version annotations in the MV-semiring annotation of a tuple records the sequence of updates that lead to the creation of the current version of the tuple. We have defined update operations and a transactional semantics for MV-semiring databases that is backward compatible to *snapshot isolation* (SI) and *read committed snapshot isolation* (RC-SI) for bag semantics databases. In the resulting semantics, each tuple in a version of a database produced by a history of transactions is annotated with its complete derivation history according to a SI or RC-SI history. Our model also supports provenance for queries, i.e., the provenance a query result cannot just be traced back to the inputs of the query, but also reaches back into the transactional history that produced these inputs. Furthermore, our model preserves a major advantage of the semiring framework: it generalizes set and bag semantics as well as other types of annotations expressible in the semiring framework such as incomplete databases. That is, we can determine the bag semantics database that is the result of a given snapshot isolation history from the annotated database for this history. We make use of this property to build a transactional debugger (see Section 5).

Example 4.1: Consider a tuple version t from an $\mathbb{N}[X]^\nu$-relation R (the MV-version of provenance polynomials) that was created by a SI history. Assume that in the current version of relation R, tuple t is annotated with $C^2_{T_1,6}(I^2_{T_1,4}(x_2))$. This annotation records that the tuple was produced by an insert (I) executed by transaction T_1 at time 3 and was assigned a tuple identifier 2. Transaction T_1 committed at time 5 after which this version of tuple t became visible to other transactions. This is encoded by the outer version annotation: $C^2_{T_1,6}$. Note that we assign a time stamp $\nu + 1$ to tuples created by an update or commit executed at time ν. We assign a fresh variable (x_2 in the example) to tuples created by an insert using a **VALUES** clause. Inserted tuples are assigned new tuple ids (id 2, shown as a superscript in the version annotation).

We have demonstrated [1, 2] that MV-semiring databases inherit many of the beneficial properties of K-relations (the semiring annotation framework) and are a strict generalization of K-relations in the following sense: given a semiring $\mathcal{K}$, the corresponding MV-semiring $\mathcal{K}^\nu$ is also a semiring. That means that we can apply the standard query semantics for K-relations to query a $\mathcal{K}^\nu$-relation. Furthermore, the K-relation corresponding to an $\mathcal{K}^\nu$-relation R can be extracted from R by applying a semiring homomorphism $\textsc{Unv}$ which evaluates the symbolic expression that is a $\mathcal{K}^\nu$ annotation by interpreting version annotation as functions from $\mathcal{K}$ to $\mathcal{K}$. Importantly, any semiring homomorphism $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ can be lifted to a homomorphism ${\mathcal{K}_1}^\nu \rightarrow {\mathcal{K}_2}^\nu$ which in addition to queries also commutes with transactional histories. Such a lifted homomorphism replaces $\mathcal{K}_1$ elements in an annotation with $\mathcal{K}_2$ elements according to h. For example, consider how to derive a bag semantics annotation from the $\mathbb{N}[X]^\nu$ annotation from the example above ($C^2_{T_1,6}(I^2_{T_1,4}(x_2))$). In the K-relational model, bag semantics is modelled by annotating tuples with their multiplicity (the semiring $\mathbb{N}$ of natural numbers). We first apply $\textsc{Unv}$ to get the provenance polynomial for tuple t. Both commit and insert annotations are interpreted as the identity function for $\mathbb{N}[X]$. Thus, $\textsc{Unv}(C^2_{T_1,6}(I^2_{T_1,4}(x_2))) = x_2$. Now further assume that tuple t appears with multiplicity 2 in the input, i.e., we apply a homomorphism $\mathbb{N}[X] \rightarrow \mathbb{N}$ based on the valuation $x_2 = 2$ and get 2. That is, in the current version of relation R, the tuple t from the example appears with multiplicity 2. The interested reader is referred to [1, 2] for the definition of update operations and transactional semantics for MV-databases as well as a more formal discussion of the properties of MV-semiring structures.

We have proven that if we extend our query model with a new operator that creates version annotations, then any update, transaction, or (partial) history in our model can be equivalently expressed as a query, e.g., from an update u we can derive a query $\mathbb{R}(u)$ which returns the same database state as the original update u (if executed over the same input). We call such queries *reenactment queries*. The equivalence of an operation and its reenactment query under annotated semantics has an important implication: instead of computing provenance eagerly during transaction execution we can compute it retroactively by running reenactment queries. Since our model generalizes bag-semantics snapshot isolation, we can use reenactment to recreate a database state valid at a particular time by simply running a query - including database states that were only visible within one transaction.

We have implemented support for transaction provenance in GProM based on reenactment. The instrumentation pipeline implementing transaction provenance is shown in Figure **??**. In addition to supporting provenance capture for past transactions using the auditing logging and time travel capabilities of modern DBMS, this pipeline also allows a hypothetical sequence of updates to be evaluated using reenactment.

Example 4.2: Assume a user is interested in evaluating the effect of a hypothetical update over the current version of a bag semantics relation `Emp(name,salary)` which increases the salary of all employees by $500 if their current salary is less than $1000. For simplicity, assume that the user is not interested in provenance (we use semiring $\mathbb{N}$ instead of $\mathbb{N}[X]^{\nu}$). This request is expressed using GProM's **`REENACT`** statement:

```
REENACT(UPDATE Emp SET salary = salary + 500 WHERE salary < 1000;);
```

To reenact this update over the current version of relation `Emp`, GProM would construct a reenactment query which returns the new state of `Emp` produced by the hypothetical update. This state is computed as a union between the set of tuples that would not be updated (do not fulfill the update's condition) and the updated versions of tuples that fulfill the update's condition (we have to increase their salary by `500`):

```
SELECT * FROM Emp WHERE NOT(salary < 1000)
UNION ALL
SELECT name, salary + 500 AS salary, b FROM Emp WHERE salary < 1000;
```

4.2 Unifying Why and Why-not Provenance

The problems of explaining why a tuple is in the result of a query or why it is missing from the result, i.e., why and why-not provenance, have been studied extensively. However, these two problems (computing provenance and explaining missing answers) have been treated mostly in isolation. An important observation is that for queries with negation, the two problems coincide: to explain why a tuple t is not in the result of a query Q, we can equivalent ask why t is in the result of $\neg Q$. Thus, a provenance model for queries with negation should naturally be able to support why-not questions. While there are extensions of the semiring model for set difference, which encodes a form of negation, the problem is that in general $\neg Q$ may not be safe. Thus, to unify the two worlds of why and why-not provenance we need a provenance model that permits unsafe queries. We have introduced in [10] a graph-based provenance model for first-order (FO) queries expressed as non-recursive Datalog queries with negation. We apply the closed world assumption to deal with unsafe queries.

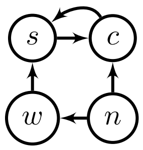
Our approach for computing provenance according to this model is based on the observation that typically only a part of provenance, which we call an *explanation*, is actually relevant for answering a user's provenance question about the existence or absence of a result. An explanation for a why (why-not) question should justify the existence (absence) of a result as the success (failure) to derive the result through the rules of the query. Furthermore, it should explain how the existence (absence) of tuples in the database caused the derivation to succeed (fail). The main driver of our approach is a rewriting of Datalog rules that captures successful and failed rule derivations. This rewriting replaces the rules of a program with so-called *firing rules*. To efficiently compute an explanation, we generate a Datalog program consisting of a set of firing rules that computes the relevant part of the provenance bottom-up. Evaluating this program over a given instance returns the egde relation of the explanation (provenance graph).

We have implemented this approach in GProM as Pipeline L3 (shown in Figure **??**). The user provides a why or why-not question and the corresponding Datalog query as an input. For this pipeline, we use GProM's Datalog frontend, which provides language constructs for expressing provenance requests. For instance, `WHY(Q(a))` would instruct GProM to explain why `Q`(a) is a query result. The system instruments the input program (query) to capture provenance relevant to the user question based on the firing rule rewriting mentioned above. This program is translated into relational algebra and the resulting algebra expression is then translated into SQL and sent to the backend databases to compute the edge relation of the explanation for the provenance question. Based on this edge relation, we render a provenance graph e.g., the graphs shown in Figure 5 and 6.

```
r1 : Q(X,Y) :- Train(X,Z),
               Train(Z,Y),
               not Train(X,Y).
```

Relation **Train**

fromCity	**toCity**
new york	washington dc
new york	chicago
chicago	seattle
seattle	chicago
washington dc	seattle



Result of query **Q**

X	**Y**
washington dc	chicago
new york	seattle
seattle	seattle
chicago	chicago

Figure 4: Example train connection database and query

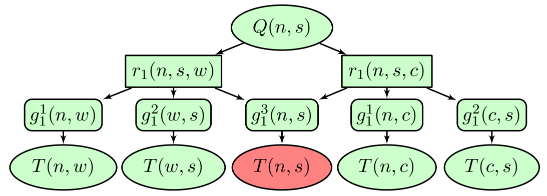


Figure 5: Provenance graph for `WHY(Q(n,s))`

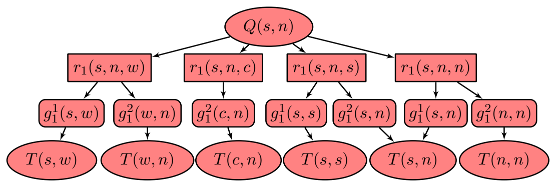


Figure 6: Provenance graph for `WHYNOT(Q(s,n))`

Example 4.3: Consider the train connections relation shown in Figure 4 and Datalog query r_1 that computes which cities can be reached with exactly one transfer, but not directly. A user might wonder why it is possible to reach Seattle from New York with one intermediate stop but not directly (`WHY(Q(n,s))`) or why it is not possible to reach Seattle from New York in the same fashion (`WHYNOT(Q(s,n))`). The provenance graph in Figure 5 explains, for the question `WHY(Q(n,s))`, how Seattle can be reached from New York via one intermediate hop, but not directly. Here, we use the following abbreviations: T = Train, n = New York, s = Seattle, w = Washington DC, and c = Chicago. In the example instance, there are two ways to reach Seattle from New York in this fashion: stopping either in Washington DC or in Chicago. These options correspond to two successful derivations of rule r_1 with X=n, Y=s, and Z=w (or Z=c, respectively). The provenance graphs produced by GProM contain three types of nodes: tuple nodes (ovals), rule nodes (rectangles), and goal nodes (rounded rectangles). The color of a node denotes its success (green) or failure (red), e.g., $\texttt{Q}(n,s)$ is labelled successful, because this tuple exists in the query result. In Figure 5 there are two rule nodes denoting the two successful derivations of $\texttt{Q}(n,s)$ by rule r_1. The provenance graph for question `WHYNOT(Q(s,n))` (Figure 6) explains why it is not true that New York can be reached from Seattle with exactly one transfer, but not directly. The tuple $\texttt{Q}(s,n)$ is missing from the query result, because all potential ways to derive this tuple through r_1 have failed. In this example, there are four failed derivations - each choosing one of the four cities present in the database as an intermediate stop. For instance, we cannot reach New York from Seattle via Washington DC (the first failed rule derivation from the left in Figure 6), because there exists no connection from Seattle to Washington DC (a tuple node $\texttt{T}(s,w)$ in red), and WashingtonDC to New York (a tuple node $\texttt{T}(w,n)$ in red). Note that the goal $\neg\texttt{T}(s,n)$ is successful and, thus, is not part of the explanation (successful goals do not contribute to failed derivations).

4.3 Provenance-aware Query Optimization

The instrumentation approach implemented in GProM has the distinct advantage that it does not require any changes to the backend database system. However, because of the intrinsic complexity and unusual structure of instrumented queries, even sophisticated database optimizers are often producing suboptimal plans for such queries. DBMS optimizers have to trade optimization time for query performance. Thus, optimizations that do not benefit common workloads are typically not considered. To address this problem, we have developed a heuristic and cost-based optimization framework for instrumentation pipelines [16]. A unique feature of our optimizer is that it is plan-space and query language agnostic and, thus, can be applied to optimize any instrumentation pipeline in GProM. Our experimental results demonstrate that this approach is quite effective, improving performance by several orders of magnitude for diverse provenance tasks.

Recall that an instrumentation pipeline is a multistep compilation process. To optimize this process we can either target an intermediate language that is the result of a compilation step or the compilation step itself.

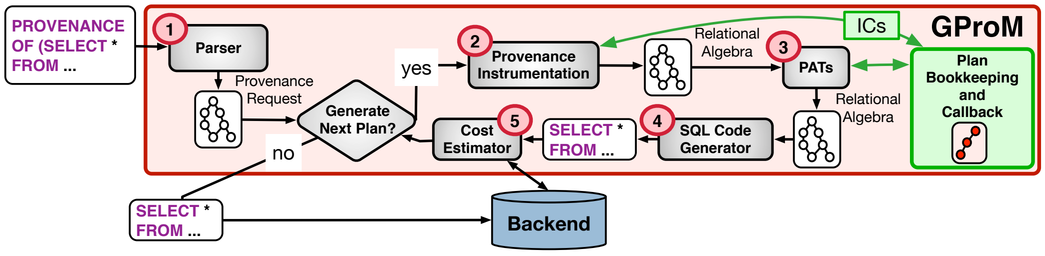


Figure 7: Optimizer workflow

As an example for the first type of optimization, consider a compilation step that outputs relational algebra, e.g., Pipelines L1 and L3 from Figure 3. We can optimize the generated algebra expression using algebraic equivalences before passing it on to the next stage of the pipeline. In [16], we focus on relational algebra since it is an intermediate language used by almost all pipelines supported by GProM. We investigate algebraic equivalences that are beneficial for instrumentation, but which are usually not applied by database optimizers. We call this type of optimizations *provenance-specific algebraic transformations* (PATs). For instance, pull up projections that create provenance annotations and remove unnecessary duplicate elimination and window operators. To be able to support rules whose conditions depend on non-local information and to simplify definition of rules, we infer properties such as candidate keys for the algebra operators of a query. For example, a duplicate elimination operator δ is redundant if its input relation is duplicate free, i.e., if it has at least one super key. Whether this is the case depends not only the operator itself, but also on its context: the subtree below the operator in this case. One of the properties we infer is a set $keys$ of super keys for an operator's output. Given this property, a PAT rule that removes duplicate eliminations is trivially expressed as: rewrite $\delta(R)$ as R if $keys(R) \neq \emptyset$.

For the second type of optimization mentioned above consider the compilation step from Pipeline L1 that translates relational algebra with annotated semantics into relational algebra. If we know two equivalent ways of translating an operator with annotated semantics into relational algebra, then we can choose the translation that maximizes performance. We refer to this type of optimizations as *instrumentation choices* (ICs). For instance, we introduce two ways for instrumenting an aggregation for provenance capture: 1) using a join [7] to pair the aggregation output with provenance from the aggregation's input; 2) using window functions (SQL's **OVER** clause) to directly compute the aggregation functions over inputs annotated with provenance.

Since some PATs are not always beneficial and for some ICs there is no clearly superior choice, there is a need for *cost-based optimization* (CBO). We have developed a CBO for instrumentation pipelines that can be applied to any pipeline no matter what compilation steps and intermediate languages are used. This is made possible by decoupling the plan space exploration from actual plan generation. Figure 7 shows how our cost-based optimizer is integrated with GProM. Our CBO treats an instrumentation pipeline as a blackbox function which it calls repeatedly to produce backend dialect queries (plans). Plans are sent to the backend for planning and cost estimation. We refer to one execution of the pipeline as an iteration. It is the responsibility of the pipeline's components to signal to the optimizer the existence of optimization choices (called *choice points*) through the optimizers callback API. The optimizer responds to a call from one of these components by instructing it which of the available options to choose. We keep track of which choices had to be made, which options exist for each choice point, and which options were chosen. This information is sufficient to enumerate the plan space by making different choices during each iteration. Our approach provides great flexibility in terms of supported optimization decisions, e.g., we can choose whether to apply a PAT or select which ICs to use. Adding an optimization choice only requires adding a few LOC to inform the optimizer about the availability of options.

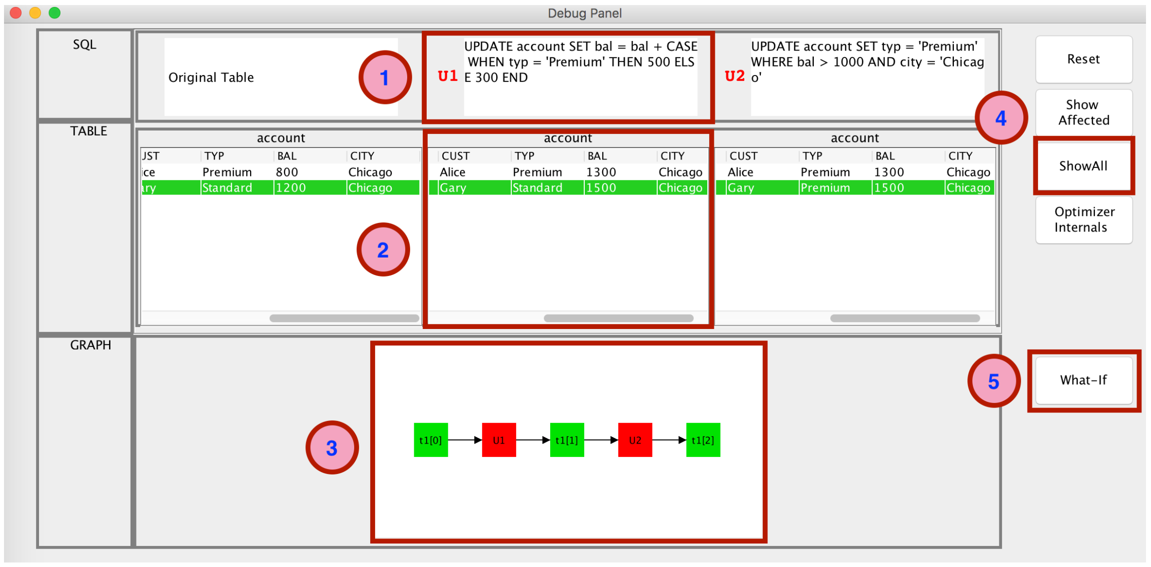
4.4 Further Contributions and Research that Utilizes GProM

In addition to the three major contributions outlined above, GProM has been the basis of many additional research thrusts. In [15], we have demonstrated how to improve interoperability between GProM and other provenance-aware systems. Specifically, we have extended Pipeline L1 from Section 3.2 to translate provenance generated by GProM into the W3C PROV standard format (`https://www.w3.org/TR/prov-overview/`) and how to propagate provenance imported as PROV through queries. Reenactment enables changes to data to be virtualized - instead of running an update we can instead just record the update statement and evaluate its effect in a non-destructive manner using reenactment. Based on this idea we have presented our vision of *provenance-aware versioned dataworkspaces* (*PVDs*) [13] which are virtual copies of a database with non-linear version histories (like the ones supported by version control systems) which can be used for exploratory purposes. We have identified historical what-if queries [3] as another use case for reenactment. Using reenactment, we can efficiently determine the effect of hypothetical changes to past update operations on the current database state. For instance, we can answer queries such as *"How would the revenue of our company be affected if we would have charged 10% interest for account overdraws instead of 7%"*. In [11] we have introduced an approximate summarization technique for why and why-not provenance extending our previous work on Datalog provenance in GProM. Using a sampling-based method, we overcome the main roadblock for explaining missing answers - the prohibitively large size of why-not provenance for databases of realistic size.

5 Post-mortem Debugging of Transactions

Aside from providing a solid platform for research on provenance and related fields, GProM and the research fueling the system have also enabled novel applications of provenance that would not have been possible before. In this section, we introduce postmortem debugging of transactions as an important example for this type of application. Debugging transactions and understanding their execution is of immense importance for developing OLAP applications, to trace causes of errors in production systems, and to audit the operations of a database. Debugging transactions, just like debugging of parallel programs, is hard because errors may only materialize under certain interleavings of operations. This problem is aggravated by the wide-spread use of lower isolation levels. Nonetheless, even for serializable histories an error may only arise for some execution orders. To debug an error, we have to reproduce the interleaving of operations which lead to the error. This problem can be addressed by supporting post-mortem debugging for transactions, i.e., enabling a user to retroactively inspect transaction executions to understand how the statements of a transaction affected the database state. While there are debuggers for procedural extensions of SQL, e.g., Microsoft's T-SQL Debugger (`http://msdn.microsoft.com/en-us/library/cc645997.aspx`), these debuggers treat SQL statements as black boxes, i.e., they do not expose the dataflow within an SQL statement. Even more important, they do not support post-mortem debugging of transaction executions within their original environment.

Supporting post-mortem debugging for transactions is quite challenging, because past database states are transient and the dataflow within and across SQL statements is opaque. While temporal databases provide access to past database versions, this is limited to committed versions. In [14], we present a non-destructive, post-mortem debugger for transactions that relies on GProM's reenactment techniques to reproduce the intermediate states of relations seen by the operations of a transaction. The approach uses provenance to expose data-dependencies between tuple versions and to explain which statements of a transaction affected a tuple version. Advanced debuggers for programming languages allow code to be changed during a debugging session to test a potential fix for a bug. We exploit the fact that GProM supports reenactment of hypothetical transactions to support such what-if scenarios, i.e., changes to a transaction's SQL statements. Being based on GProM's reenactment functionality, our approach uses the temporal database and audit logging capabilities available in many DBMS and does not require any modifications to the underlying database system nor transactional workload.



```
UPDATE account SET bal = bal + CASE
 WHEN typ = 'Premium' THEN 500 ELS
E 300 END
```

```
UPDATE account SET typ = 'Premium'
WHERE bal > 1000 AND city = 'Chicag
o'
```

account

JST	TYP	BAL	CITY
ice	Premium	800	Chicago
ry	Standard	1200	Chicago

account

CUST	TYP	BAL	CITY
Alice	Premium	1300	Chicago
Gary	Standard	1500	Chicago

account

CUST	TYP	BAL	CITY
Alice	Premium	1300	Chicago
Gary	Premium	1500	Chicago

Figure 8: Screenshot of the Debugger GUI

```
UPDATE account
SET bal = bal + CASE WHEN typ = 'Premium'
                THEN 500 ELSE 300 END

UPDATE account SET typ = 'Premium'
WHERE bal > 1000 AND city = 'Chicago';
```

(a) Transaction T

account

cust	typ	bal	city
Alice	Premium	800	Chicago
Gary	Standard	1200	Chicago

(b) Before T

account

cust	typ	bal	city
Alice	Premium	1300	Chicago
Gary	Premium	1500	Chicago

(c) After T

Figure 9: Example transaction

Example 5.1: Transaction T shown in Figure 9 adds a bonus to bank accounts ($500 for premium customers and $300 for standard customers) and gives premium status to all accounts whose balance is larger than $1000. Figure 9 also shows the state of the account relation before and after the update. For instance, after the execution of Transaction T, Gray's account enjoys premium status. However, Gary has only received a $300 bonus, the bonus for standard accounts. Our debugger can be used to inspect the internal states of the transaction that lead to this behaviour. Figure 8 shows a screenshot of the debugger's GUI for Transaction T. We show one column for each of T's operations plus a column for the initial states of the relations accessed by T. Each such column shows the SQL code of the statement (①) and the relation (②) modified by the statement (the version created by the statement). For each tuple version, we show which transaction created that version. In Figure 8, the user has selected Gary's account. Thus, the debugger shows a provenance graph (③) for this tuple and highlights the updates that affected it. From the intermediate states of the relations and the provenance graph it is immediately clear that the bonus payment was applied when Gary's status was still standard. Our debugger supports two types of what-if scenarios by clicking the "What-if" button (⑤): 1) the user can edit the data in a table and 2) the user can modify, delete, or add an update statement.

6 Conclusions and Future Work

We give a comprehensive overview of GProM (**G**eneric **Pro**venance **M**iddleware). GProM is a fully implemented system that we hope will serve as a platform for future research on provenance and annotation management as well as a framework for building provenance-aware applications. The diverse types of research threads and applications for which we have employed GProM, demonstrate the potential of our system and the feasibility of its modular, extensible design. We also give an overview of research contributions related to GProM, most notably, a provenance model for transactions and reenactment, capturing why and why-not provenance for

Datalog queries, and provenance-aware query optimization. The query instrumentation technique that is at the heart of GProM is applicable for a wide range of use cases that are not necessarily all related to provenance. For instance, we have extended GProM to evaluate temporal queries and to compute uncertainty annotations. GProM provides solid support for classical applications of provenance and has enabled novel applications. We discuss post-mortem debugging of transactions as one exciting use case of the system. Finally, we highlight interesting future work and discuss ongoing research efforts that benefit from GProM.

Acknowledgements. This work was supported by NSF Award #1640864. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank our collaborators without whom this work would not have been possible and the Oracle corporation for generously supporting this work through multiple gifts.

References

[1] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, pages 841–850, 2016.

[2] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Using reenactment to retroactively capture provenance for transactions. *TKDE*, 2017 (to appear).

[3] B. Arab and B. Glavic. Answering historical what-if queries with provenance, reenactment, and symbolic execution. In *TaPP*, 2017.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[5] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[6] B. Glavic and K. R. Dittrich. Data Provenance: A Categorization of Existing Approaches. In *Proceedings of the 12th GI Conference on Datenbanksysteme in Buisness, Technologie und Web*, pages 227–241, 2007.

[7] B. Glavic, R. J. Miller, and G. Alonso. Using SQL for efficient generation and querying of provenance information. *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, pages 291–320, 2013.

[8] M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

[9] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[10] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *ICDE*, pages 485–496, 2017.

[11] S. Lee, X. Niu, B. Ludäscher, and B. Glavic. Integrating approximate summarization with provenance capture. In *TaPP*, 2017.

[12] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, 2010.

[13] X. Niu, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, O. Kennedy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, 2016.

[14] X. Niu, B. Glavic, S. Lee, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, F. Su, and X. Zou. Debugging transactions and tracking their provenance with reenactment. *PVLDB*, 10(12):1857–1860, 2017.

[15] X. Niu, R. Kapoor, D. Gawlick, Z. H. Liu, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Interoperability for provenance-aware databases using PROV and JSON. In *TaPP*, 2015.

[16] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. Provenance-aware query optimization. In *ICDE*, pages 473–484, 2017.