# A Generic Provenance Middleware for Database Queries, Updates, and Transactions

Bahareh Sadat Arab
Illinois Institute of Technology
barab@iit.edu

Dieter Gawlick
Oracle Corporation
dieter.gawlick@oracle.com

Venkatesh Radhakrishnan
Oracle Corporation
venkatesh.radhakrishnan@oracle.com

Hao Guo
Illinois Institute of Technology
hguo@iit.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

## Abstract

We present an architecture and prototype implementation for a generic provenance database middleware (**GProM**) that is based on the concept of query rewrites, which are applied to an algebraic graph representation of database operations. The system supports a wide range of provenance types and representations for queries, updates, transactions, and operations spanning multiple transactions. GProM supports several strategies for provenance generation, e.g., on-demand, rule-based, and "always on". To the best of our knowledge, we are the first to present a solution for computing the provenance of concurrent database transactions. Our solution can retroactively trace transaction provenance as long as an audit log and time travel functionality are available (both are supported by most DBMS). Other noteworthy features of GProM include: extensibility through a declarative rewrite rule specification language, support for multiple database backends, and an optimizer for rewritten queries.

***Categories and Subject Descriptors*** H.2.4 [*Relational databases*]

***Keywords*** Provenance, Databases, Query Rewrite, Transactions

## 1. Introduction

Provenance tracking for database operations, i.e., automatically collecting and managing information about the origin of data, has received considerable interest from the database community in the last decade. Efficiently generating and querying provenance is essential for debugging data and queries, evaluating trust measures for data, defining new types of access control models, auditing, and as a supporting technology for data integration and probabilistic databases. The de-facto standard for database provenance is to model provenance as annotations on data and compute the provenance for the outputs of an operation by propagating annotations [1, 4, 11, 13, 15]. Many provenance systems [11] use a relational encoding of provenance annotations. These systems apply query rewrite techniques to transform a query $q$ into a query that propagates input annotations to produce the result of $q$ annotated with provenance. This approach has many advantages. It benefits from existing database technology, e.g., provenance computations are optimized by the database optimizer. Queries over provenance can be expressed as SQL queries over the relational encoding [11]. Alternatively, we can compile a special-purpose provenance query language into SQL queries over such an encoding [4, 14]. In spite of the advances made, the current state-of-the-art falls short in several aspects:

- Even though provenance of relational *updates* (we use *update* as an umbrella term for DML operations) is relatively well understood [2, 5, 15, 17], no comprehensive implementation exists. Furthermore, no solution for tracking transactions has been proposed so far.
- Systems are inflexible in their support for deciding when to compute provenance, when to store it, and how to store it. For example, Trio [1], DBNotes [4], and ORCHESTRA [15] compute and generate provenance for all operations and systems such as Perm [11] do not compute any provenance unless it is explicitly requested.
- Queries produced by query rewrites use atypical access patterns and operator sequences which often leads to poor execution plans, even for database systems with sophisticated optimizers.
- Most systems only support one type of provenance using one particular representation of provenance.

In this work, we present our vision and a prototype implementation for GProM, a generic provenance middleware, that addresses the aforementioned problems. We use annotation propagation and query rewrite techniques for computing, querying, storing, and translating the provenance of SQL queries, updates, transactions, and across transactions. To the best of our knowledge, we are the first to present an approach for tracking the provenance of concurrent transactions (our prototype supports snapshot isolation [3] and statement snapshot isolation[1]). We will support a wide variety of database backends from mature DBMS to SQL processors running over distributed analytics platforms.

We first present an overview of the system in Section 2 and then describe our main contribution, provenance computation for updates and transactions, in Section 3. The prototype implementation of GProM will be presented in Section 4. Appendix A shows

[1] To be explained later

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CONF 'yy,* Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

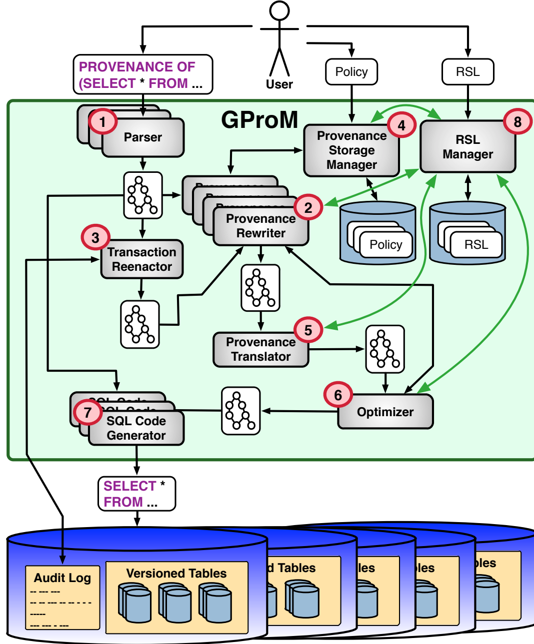


Figure 1: GProM Overview

how GProM processes an example provenance request for a transaction to translate it into SQL code. We present additional details about the extensibility mechanism, database independence, provenance translations, provenance import/export, optimizer, and storage policies, in Appendices B, C, D, E, F, and G. Related work will be discussed inline where appropriate.

## 2. System Overview

Figure 1 shows an overview of GProM. The user interacts with the system using an extension of the underlying database system's SQL dialect. Specifically, we support new language constructs for computing and managing provenance (similar to Perm [11]). Incoming statements are translated into a relational algebra graph representation which we call *algebra graph model* (*AGM*) ①. Similar to intermediate code representations used by compilers, this model is used as a representation of computation which is independent of the target language. If the statement does not use any provenance features, then the AGM model is translated back into the native SQL dialect using a vendor specific SQL code generator ⑦.

**Provenance Computation**: Similar to Perm [11] (and other systems [15]) we represent provenance information using a relational encoding of provenance annotations. This representation is flexible enough to encode typical database provenance models including *PI-CS* [11] (and, thus, provenance polynomials [13]), Where- and Why-provenance [6], and many others. The *provenance rewriter* module ② uses provenance-type specific rules to rewrite an input query $q$ into a query $q^+$ that propagates annotations to produce this encoding of data annotated with provenance.

**Supporting Past Queries, Updates, and Transactions**: One unique feature of GProM is that the system can retroactively compute the provenance of queries, updates, and transactions. This feature requires that a log of database operations is available (we call this an *audit log*) and that the underlying database system supports *time travel*, i.e., querying past versions of a relation. These features are available in most database systems or can be added using extensibility mechanisms. An audit log paired with time travel functionality is sufficient for computing the provenance of past queries using simple modifications of standard provenance rewrites [7, 18]. Our main contribution is to demonstrate that this is also sufficient for tracking the provenance of updates and transactions. If the user requests provenance for a transaction $T$, the *transaction reenactor* ③ extracts the list of SQL statements executed by $T$ from the audit log and constructs a *reenactment query* $q(T)$ that simulates the effects of these statements. We use the provenance rewriter to rewrite $q(T)$ into a query $q(T)^+$ that computes the provenance of the reenacted transaction. Note that the construction of $q(T)$ is independent of the provenance[2] rewrite and $q(T)$ is a standard relational query. Using this approach, we can compute any type of provenance for updates, transactions, and across transactions as long as rewrite rules for computing the provenance of queries have been implemented for this provenance type.

**Provenance Generation and Storage Policies**: As explained above we can reconstruct provenance for any past query, update, or transaction using the audit log and time travel. Thus, explicit provenance storage is unnecessary. The default in GProM is to only compute provenance if it is explicitly requested. Nonetheless, we also support automatic provenance generation and storage for use cases where retroactive reconstruction of provenance is not an option. The user can register provenance storage policies with the *storage manager* ⑥. These policies determine when and how to generate and store provenance.

**Optimizing Rewritten Queries**: GProM will include an *optimizer* ⑦ which applies heuristic and cost-based rules to transform rewritten queries into SQL code that can be successfully optimized by the underlying DBMS. This is necessary, because provenance rewrites generate queries with unusual access patterns and operator sequences. Even sophisticated database optimizers are not capable of producing reasonable plans for such queries.

**Rewrite Extensibility**: GProM relies on query rewrites for provenance computation, optimizations, transaction support, provenance storage, and translation between provenance representations. Thus, the main bottleneck for extending the system with new functionality is implementing new rewrites. We will develop a declarative *rewrite specification language* (*RSL*) for expressing rewrite rules in a concise and easy to understand manner. User and system-developer provided rewrites will be stored in a *repository* ⑧. Rules are applied to input queries using an interpreter for this language.

## 3. Support for Updates And Transactions

GProm is the first system capable of computing provenance for queries, update operations, transactions, and across transaction boundaries. Provenance computation for updates and transactions is implemented in the *transaction reenactment* module of the system. We can retroactively compute the provenance of transactions (and across transactions) as long as two conditions are met: 1) the underlying database supports *time travel*, i.e., we can retrieve a past version of a relation and 2) the database keeps a log of executed SQL statements (the aforementioned *audit log*). We require the audit log to contain at least the following information for each executed statement: an identifier for the transaction (*xid*) this statement was part of, a timestamp storing when the statement was executed, and the SQL code for the statement.

As observed by Zhang et al. [18], an audit log and time travel combined with standard provenance rewrites can be used to compute the provenance of past queries without the need to store any additional information. Chirigati et al. [7] also use these features to compute provenance of database operations with the goal to com-

[2] This is because the reenactment query $q(T)$ and transaction $T$ are annotation-equivalent, i.e., they have the same result and provenance.

bine workflow and database provenance. However, their approach has the disadvantage that it models the provenance of a tuple as old versions of this tuple and does not model additional provenance dependencies to other input tuple versions. For example, consider an update which inserts tuples from a relation $R$ into a relation $S$ (`INSERT INTO S (SELECT * FROM R)`). No previous version exists for tuples inserted into relation $S$, because temporal databases do not track dependencies across relations. We reenact update operations to unearth such additional dependencies and compute the provenance of transactions. Our transaction reenactment approach produces a query $q(T)$ which reenacts the updates executed by a transaction $T$. To compute the provenance of transaction $T$, we rewrite $q(T)$ using rules designed to compute the provenance of queries. One important advantage of this approach is that we only need rewrite rules for computing the provenance of queries. How to implement such rules is relatively well understood [11]. We compute the provenance of a transaction $T$ as follows.

**Gather Transaction Information**: We access the audit log to retrieve the SQL statements $u_0, \ldots, u_n$ of transaction $T$ and for each statement $u_i$ the time $\tau_i$ when the statement was executed.

**Translate updates**: We transform each SQL statement $u_i$ into an AGM reenactment query $q(u_i)$. Assume that statement $u_i$ did update relation $R_i$. Query $q(u_i)$, if evaluated over the state of relation $R_i$ as of the time when $u_i$ was originally executed, returns the updated content of relation $R_i$. We use the database backend's time travel features to access this version of $R_i$.

**Construct Reenactment Query**: The individual update reenactment queries are then merged into a global reenactment query $q(T)$ simulating the whole transaction. We need to reconstruct the input for each update to correctly reenact the transaction. Different concurrency control mechanisms enforce different visibility rules for concurrent modifications and, thus, each concurrency control mechanism requires a different merge process. For example, updates of a transaction $T$ running under snapshot isolation [3] only see modifications by transactions that did commit before $T$ started and modifications by previous updates of $T$.

**Rewrite For Provenance Computation**: The query $q(T)$ is rewritten for provenance computation according to the type of provenance requested by the user. The result $q(T)^+$ is then passed to the storage, translation, optimizer, and SQL code generator modules to translate it into efficient SQL code.

### 3.1 Update Reenactment

The provenance of updates has been studied in related work [2, 5, 15, 17], but none of these approaches addresses the complications that arise when updates are run as parts of concurrent transactions.[3] Buneman et al. [5] have studied a copy-based provenance type for the nested update language and nested relational calculus. Vansummeren et al. [17] define provenance for SQL DML statements. This approach modifies the updates to store provenance. Our approach differs in that we reconstruct provenance on demand instead of computing and storing provenance for all operations.

The three types of SQL update statements (`UPDATE`, `INSERT`, and `DELETE`) all modify a single relation $R$. Conceptually, an update statement reads the version of relation $R$ ($R^{t_0}$) before the update[4] and returns an updated version of relation $R$ ($R^{t_1}$). We can reenact the modifications using a query that runs over the version of the database before the update ($D^{t_0}$). The result relation of such a query is $R^{t_1}$, the updated version of relation $R$. For example, consider the following `UPDATE` statement that doubles the value of attribute $B$ for all tuples fulfilling the condition `A = 'b'`. An example instance of relation $R$ is shown in Figure 2.

$R^{t_0}$

| A | B |
|---|---|
| a | 12 |
| b | 3 |

$R^{t_1}$

| A | B |
|---|---|
| a | 12 |
| b | 6 |

Provenance

| A | B | P(A) | P(B) |
|---|---|---|---|
| a | 12 | a | 12 |
| b | 6 | b | 3 |

Figure 2: Relation $R$ before ($R^{t_0}$) and after the update ($R^{t_1}$), and the provenance of the update operation.

```
UPDATE R SET B = B * 2 WHERE A = 'b';
```

Execution of this update triggers insertion of a new entry into the audit log. The update can be reenacted based on the database instance at time $t_0$ (the version seen by this operation). $R^{t_1}$, the updated version of relation $R$, will contain the updated version for all tuples fulfilling the `WHERE` clause condition and the previous version ($R^{t_0}$) for all remaining tuples (the ones not fulfilling the condition). Consequently, the update operation $u$ can be reenacted as a query $q(u)$ which returns all tuples from $R^{t_0}$ that do not fulfill the condition[5] and returns the updated version of all tuples from $R^{t_0}$ fulfilling the condition. Here `R AS OF` $t$ denotes the use of time travel to retrieve the version of $R$ at time $t$. Note that the result of this query is exactly $R^{t_1}$ shown in Figure 2.

```
SELECT a, b * 2 AS b
FROM R AS OF t0 WHERE A = 'b';
UNION ALL
SELECT *
FROM R AS OF t0 WHERE (A = 'b') IS NOT TRUE;
```

To compute the provenance of update $u$, we use the provenance rewriter to rewrite this reenactment query into a query $q(u)^+$ that computes the provenance of $u$. As an example, consider the resulting query (shown below) for the *PI-CS* provenance type introduced in Perm [11]. The relational representation used by this query uses additional attributes $P(A)$ and $P(B)$ to pair an updated tuple version with the tuple versions in its provenance. Figure 2 shows the result of this query for the example instance.

```
SELECT a, b * 2 AS b, A AS P(A), B AS P(B)
FROM R AS OF t0
WHERE A = 'b';
UNION ALL
SELECT a, b, A AS P(A), B AS P(B)
FROM R AS OF t0
WHERE (A = 'b') IS NOT TRUE;
```

Our provenance rewriter can produce a variation of this rewritten query that in addition shows the unique tuple identifier and version number for each tuple in the result and each tuple in the provenance. This is useful to disambiguate tuples with the same attribute values and to determine which version of a tuple is in the provenance. In some database systems, such a (tuple ID,version) pair can be used to extract additional information about the update [8], e.g., the user who executed it. Alternatively, we also support returning only updated tuples, limiting the result size of the provenance query to the number of updated tuples.

### 3.2 Transaction Reenactment

If we ignore concurrent operations, then a transaction is just a sequence of update statements where each update accesses the database version produced by the previous updates. For simplicity, we limit the discussion to transactions where all updates modify the same relation. Our system also supports transactions that

[3] Note that the "transactions" studied by Archer et al. [2] are sequences of update operations and not concurrent database transactions.

[4] Updates may access additional relations from the same database version.

[5] We need to use `(a = 'b') IS NOT TRUE` instead of `NOT(a = 'b')` to account for cases where the update's condition evaluates to `NULL`

do not follow this pattern. To compute the provenance of such a transaction we chain the translations of the transaction's updates so that reenactment query $q(u_i)$ reads from the result of reenactment query $q(u_{i-1})$. The result of this chaining process is a query $q(T)$ that runs over the version of relation $R$ as of transaction start and returns the updated version of $R$ produced by the transaction.

If transactions are run concurrently, then we need to reconstruct the database version seen by each update which will be a mix of tuple versions from some past state of the database and local tuple versions produced by previous statements of the same transaction. The exact details depend on the concurrency control mechanism applied by the database system. Currently, we support *snapshot isolation* [3] and *statement snapshot isolation*. Under snapshot isolation, a transaction $T$ does not see any modifications of currently running transactions. Consequently, such a transaction can be reenacted by chaining update reenactment queries as mentioned above ignoring changes made by concurrent transactions. Under statement snapshot isolation each statement within a transaction $T$ sees a consistent snapshot based on its start time (instead of all statements of $T$ using the same snapshot based on the transaction start time) and write operations wait for concurrent writers to commit instead of aborting. Reenactment of statement snapshot isolation transactions is more involved, but follows the same principle: reconstructing the input seen by each update. Computing the provenance of an reenactment query $q(T)$ produces the provenance of the whole transaction $T$ with respect to the updated relation (recall that we limit the discussion to transactions which only update a single relation).

Our approach is not limited to single transactions. We can also construct reenactment queries for tracking the provenance of a relation back to a certain database version or back to its creation and beyond (we support user-provided provenance for imported data).

## 4. Prototype Implementation

We have implemented a first prototype of GProM using Oracle as a database backend. At this point in time we have a working parser, algebraic query model, update and transaction reenactor, and provenance rewriter. We currently support reenactment for transactions run under snapshot isolation and statement snapshot isolation[6], provenance rewrites for the PI-CS provenance type [11], provenance computations as parts of queries, and some basic simplification rules for our heuristic optimizer. The required time travel and audit log functionality is readily available through Oracle's *Total Recall* and *fine grained auditing* (*FGA*) features [8].

## 5. Conclusions

We present our vision for GProM, a database-independent middleware for computing the provenance of queries, updates, and transactions. Our approach takes query rewrite techniques to the next level by using them for provenance computation, transaction reenactment, provenance translation, provenance storage, and optimization. The system can be extended with new rewrite rules specified in a declarative rewrite specification language (*RSL*). To the best of our knowledge, we are the first to support provenance computation within and across concurrent database transactions. Notably, this feature only requires the underlying database to support time travel and to provide an audit log. Furthermore, this feature is independent of what type of provenance is computed and, aside from the runtime and storage overhead caused by maintaining the audit log and data required to support time travel, results in no overhead for normal database operations. Our prototype implementation using Oracle demonstrates the feasibility of our approach.

There are many interesting avenues for future work such as implementing additional provenance types, a comprehensive study of heuristic and cost-based optimizations for rewritten queries, the language design and implementation of RSL, implementing additional provenance formats, and supporting lock-based concurrency control mechanisms in transaction reenactment.

[6] Called isolation level `SERIALIZABLE` and `READ COMMITTED` by Oracle.

## Acknowledgments

This research is partially supported by a gift from the Oracle Corporation. We would like to thank the following contributors: Shukun Xie, Bowen Dan, Pankaj Purandare, Zefeng Lin, and Ying Ni.

## References

[1] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB*, pages 1151–1154, 2006.

[2] D. W. Archer, L. M. Delcambre, and D. Maier. User Trust and Judgments in a Curated Database with Explicit Provenance. In *In Search of Elegance in the Theory and Practice of Computation*, pages 89–111. Springer, 2013.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24 (2):1–10, 1995.

[4] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4):373–396, 2005.

[5] P. Buneman, J. Cheney, and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. *TODS*, 33 (4):1–47, 2008.

[6] 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.

[7] F. Chirigati and J. Freire. Towards integrating workflow and database provenance. In *IPAW*, pages 11–23. 2012.

[8] D. Gawlick and V. Radhakrishnan. Fine grain provenance using temporal databases. In *TaPP*, 2011.

[9] B. Glavic and G. Alonso. Provenance for Nested Subqueries. In *EDBT*, pages 982–993, 2009.

[10] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul. Ariadne: Managing fine-grained provenance on data streams. In *DEBS*, pages 39–50, 2013.

[11] B. Glavic, R. J. Miller, and G. Alonso. Using SQL for Efficient Generation and Querying of Provenance Information. In *In Search of Elegance in the Theory and Practice of Computation*, pages 291–320. Springer, 2013.

[12] T. Grust, M. Mayr, and J. Rittinger. Let SQL drive the XQuery workhorse (XQuery join graph isolation). In *EDBT*, pages 147–158, 2010.

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

[14] G. Karvounarakis, Z. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.

[15] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *TODS*, 38(3): 19, 2013.

[16] P. Missier, K. Belhajjame, and J. Cheney. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT*, pages 773–776, 2013.

[17] S. Vansummeren and J. Cheney. Recording Provenance for SQL Queries and Updates. *IEEE Data Engineering Bulletin*, 30(4):29–37, 2007.

[18] J. Zhang and H. Jagadish. Lost source provenance. In *EDBT*, pages 311–322, 2010.

**USacc**

| ID | Owner | Balance | Type |
|---|---|---|---|
| 1 | Fanny Marble | 2,000,000 | Premium |
| 2 | Peter Bright | 1,000 | Standard |

**CanAcc**

| ID | Owner | Balance | Type |
|---|---|---|---|
| 3 | Alice Bright | 1,500,000 | US dollar |
| 4 | Mark Smith | 20,000 | Standard |
| 5 | Mark Smith | 50 | US dollar |

Figure 3: Example Instance For Running Example

# A. Example Provenance Computation

We demonstrate how our approach handles a provenance request for a transaction using an example which requires the application of several types of rewrites. For simplicity, we will use SQL code through this section instead of the AGM representation used internally by GProM.

Consider a bank that stores information about US accounts in a relation `USacc` and Canadian accounts in relation `CanAcc`. Example instances for these relations are shown in Figure 3. The bank decides to give free US accounts to Canadian customers that already have an US dollar account (`Type = 'US␣dollar'`) with the Canadian branch. The new accounts in relation `USaccs` are created from the data of the corresponding `CanAcc` tuples. These newly created accounts should get "premium" status if their balance is over 1,000,000 US Dollar. Finally, "standard" accounts with low balance (below 100 US Dollar) should be removed as it is the bank's policy to require a minimum 100 dollar balance for "standard" accounts in the US. A transaction implementing these changes is shown in Figure 6 (left hand side).

Assume that this transaction has been run under isolation level `SERIALIZABLE` (recall that this is actually snapshot isolation) and was assigned the transaction identifier (*xid*) `0A0202F5`. Figure 4 shows the content of the audit log produced by running this transaction. Recall that our prototype implementation uses Oracle's fine grained auditing feature for the audit log. Oracle stores the audit log in a relation called `fga_log$`. An internal version counter called the *system change number* (*SCN*) is used to represent versions. Note that all statements are assigned the same SCN in the example, because in a `SERIALIZABLE` transaction each statement sees the same snapshot (version) of the database modulo changes of previous statements of the same transaction.

## A.1 Request Provenance

If interested in the provenance of this transaction, the user may request it as follows using the transaction's *xid*.

```
PROVENANCE OF TRANSACTION '0A0202F5';
```

GProM's parser will recognize that this statement requests the provenance of a transaction and pass an initial AGM graph consisting of a dummy operator representing the provenance computation to the transaction reenactor module.

## A.2 Gather Transaction Information

The transaction reenactor gathers information about the transaction from the audit log. In particular, it determines the SCNs and sql code for all statements executed by the transaction using the query shown below. The result of this query also can be used to determine the isolation level under which the transaction was executed.

```
SELECT SCN, sql
FROM audit_log
WHERE xid = '0A0202F5'
ORDER BY execOrder;
```

**USacc after $u_1$**

| ID | Owner | Balance | Type |
|---|---|---|---|
| 1 | Fanny Marble | 2,000,000 | Premium |
| 2 | Peter Bright | 1,000 | Standard |
| 3 | Alice Bright | 1,500,000 | Standard |
| 5 | Mark Smith | 50 | Standard |

**USacc after $u_2$**

| ID | Owner | Balance | Type |
|---|---|---|---|
| 1 | Fanny Marble | 2,000,000 | Premium |
| 2 | Peter Bright | 1,000 | Standard |
| 3 | Alice Bright | 1,500,000 | Premium |
| 5 | Mark Smith | 50 | Standard |

**USacc after $u_3$**

| ID | Owner | Balance | Type |
|---|---|---|---|
| 1 | Fanny Marble | 2,000,000 | Premium |
| 2 | Peter Bright | 1,000 | Standard |
| 3 | Alice Bright | 1,500,000 | Premium |

Figure 5: Updated Example Instances

## A.3 Translate updates

In the next step, we generate an individual reenactment query $q(u_i)$ for each update $u_i$ of the transaction. The translation for updates has been explained in Section 3. An insert statement adds new tuples to the relation. This is simulated by computing the union of the version of the relation before the insert and the newly inserted tuples. A delete retains the unmodified versions of all tuples that do not fulfill the `WHERE` clause condition. The reenactment queries generated for the running example are shown on the right hand side of Figure 6.

## A.4 Construct Reenactment Query

The individual update translations are then merged into a global reenactment query by recursively replacing in each reenactment query $q(u_i)$ all accesses to relation `USacc` with the query $q(u_{i-1})$ producing the version of the relation read by $u_i$. The result of this process is shown below.

```
WITH
u1 AS
(SELECT ID, Owner, Balance, 'Standard' AS Type
FROM CanAcc AS OF SCN 3652
WHERE Type = 'US␣dollar'
UNION ALL
SELECT * FROM USacc AS OF SCN 3652),

u2 AS
(SELECT ID, Owner, Balance, 'Premium' AS Type
FROM u1 WHERE Balance > 1000000
UNION ALL
SELECT * FROM u1
WHERE (Balance > 1000000) IS NOT TRUE)

SELECT * FROM u2
WHERE (Balance < 100) IS NOT TRUE;
```

## A.5 Rewrite For Provenance Computation

The generated reenactment query is then rewritten for provenance computation according to the type of provenance that was re-

| xid | execOrder | SCN | sql |
|---|---|---|---|
| 0A0202F5 | 1 | 3652 | `INSERT INTO USacc (SELECT ID, Owner , Balance, 'Standard' AS Type FROM CanAcc WHERE Type = 'US␣dollar')` |
| 0A0202F5 | 2 | 3652 | `UPDATE USacc SET Type = 'Premium' WHERE Balance > 1000000` |
| 0A0202F5 | 3 | 3652 | `DELETE FROM USacc WHERE Balance < 100` |

Figure 4: Example Audit Log

**Transaction** / **Update Reenactment Queries**

$u_1$:
```
INSERT INTO USacc
   (SELECT ID,Owner,Balance,
           'Standard' AS Type
   FROM CanAcc
   WHERE Type = 'US␣dollar');
```

$q(u_1)$:
```
SELECT ID, Owner, Balance, 'Standard' AS Type
FROM CanAcc AS OF SCN 3652
WHERE Type = 'US␣dollar'
UNION ALL
SELECT * FROM USacc AS OF SCN 3652;
```

$u_2$:
```
UPDATE USacc SET Type = 'Premium'
 WHERE Balance > 1000000;
```

$q(u_2)$:
```
SELECT ID, Owner, Balance, 'Premium' AS Type
FROM USacc AS OF SCN 3652
WHERE Balance > 1000000
UNION ALL
SELECT *
FROM USacc AS OF SCN 3652
WHERE (Balance > 1000000) IS NOT TRUE;
```

$u_3$:
```
DELETE FROM USacc WHERE Balance < 100;
```

$q(u_3)$:
```
SELECT *
FROM  USacc AS OF SCN 3652
WHERE (Balance < 100) IS NOT TRUE;
```

Figure 6: Example Transaction and Translated Updates

quested. The query shown below has been rewritten to compute PI-CS provenance [11]. Note that the user has requested that only tuples modified by the transaction should be returned. This is realized by propagating an attribute `updated` which is set to 1 for updated tuples and to 0 for tuples which have not been updated. This attribute is used to select only updated tuples.

```
WITH
u1 AS
(SELECT ID, Owner, Balance, 'Standard' AS Type,
        ID AS prov_CanAcc_ID,
        Owner AS  prov_CanAcc_Owner,
        Balance AS prov_CanAcc_Balance,
        Type AS prov_CanAcc_Type,
        NULL AS prov_USacc_ID,
        NULL AS prov_USacc_Owner,
        NULL AS prov_USacc_Balance,
        NULL AS prov_USacc_Type,
        1 AS updated,
FROM CanAcc AS OF SCN 3652
WHERE Type = 'US␣dollar'
UNION ALL
SELECT ID, Owner, Balance, Type,
        NULL AS prov_CanAcc_ID,
        NULL AS prov_CanAcc_Owner,
        NULL AS prov_CanAcc_Balance,
        NULL AS prov_CanAcc_Type,
        ID AS prov_USacc_ID,
        Owner AS  prov_USacc_Owner,
        Balance AS prov_USacc_Balance,
        Type AS prov_USacc_Type
        0 AS updated
FROM USacc AS OF SCN 3652),

u2 AS
(SELECT ID, Owner, Balance, 'Premium' AS Type,
        prov_CanAcc_ID,
        prov_CanAcc_Owner,
        prov_CanAcc_Balance,
        prov_CanAcc_Type,
        prov_USacc_ID,
        prov_USacc_Owner,
        prov_USacc_Balance,
        prov_USacc_Type
        1 AS updated
FROM u1
WHERE Balance > 1000000
UNION ALL
SELECT * FROM u1
WHERE (Balance > 1000000) IS NOT TRUE)

SELECT *
FROM u2
WHERE (Balance < 100) IS NOT TRUE
      AND updated = 1;
```

### A.6 Heuristic Optimization

GProM features an optimizer for AGM queries. This optimizer is used to transform rewritten queries into queries that can be translated into efficient SQL code. Our current prototype implementation only supports some primitive simplification rules and heuristic choices between alternative rewrite methods. For example, we can reduce the number of set operations by using an alternative reenactment query generation for `UPDATE` statements. Instead of computing the union between the updated tuples and not updated tuples, we use the `CASE` construct to check the update's condition for each input tuple and only modify attribute values for tuples which fulfill this condition. The common table expression $u_2$ in the previous query can be optimized as follows:

```
...
u2 AS
```

**Result of Example Provenance Query**

| Updated USacc Tuples | | | | Provenance from CanAcc | | | | Provenance from USacc | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **ID** | **Owner** | **Balance** | **Type** | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | $P_8$ |
| 3 | Alice Bright | 1,500,000 | Premium | 3 | Alice Bright | 1,500,000 | US dollar | NULL | NULL | NULL | NULL |

Figure 7: Provenance for the Running Example Transaction

| Abbreviation | Provenance Attribute Name |
|---|---|
| $P_1$ | prov_USacc_ID |
| $P_2$ | prov_USacc_Owner |
| $P_3$ | prov_USacc_Balance |
| $P_4$ | prov_USacc_Type |
| $P_5$ | prov_CanAcc_ID |
| $P_6$ | prov_CanAcc_Owner |
| $P_7$ | prov_CanAcc_Balance |
| $P_8$ | prov_CanAcc_Type |

Figure 8: Provenance Attribute Names

```
(SELECT ID, Owner, Balance,
        CASE
          WHEN Balance > 1000000 THEN 'Premium'
          ELSE Type
        END AS Type,
        prov_CanAcc_ID,
        prov_CanAcc_Owner,
        prov_CanAcc_Balance,
        prov_CanAcc_Type,
        prov_USacc_ID,
        prov_USacc_Owner,
        prov_USacc_Balance,
        prov_USacc_Type,
        1 AS updated
FROM u1)
...
```

### A.7 Executing the Rewritten Query

Figure 7 shows the result of executing the rewritten query. In the result, the updated version of each tuple produced by the transaction is paired with its provenance in relations `USacc` and `CanAcc`. Full attribute names for provenance attributes are shown in Figure 8.

## B. Extensibility using RSL

The architecture of GProM, in particular the AGM model that we use as an internal representation of queries, makes it easy to extend the system with new types of rewrites no matter whether they are used to implement a new provenance type, heuristic optimization rule, provenance storage strategy, or provenance representation. Since query rewrites are such a fundamental part of our approach, we propose to develop a special-purpose language for specifying query rewrites. This language which we call *rewrite specification language* (*RSL*) will be a rule-based graph transformation language. It will enable concise specification of query rewrites over our AGM model and, thus, allow for rapid development of new extensions for GProM. Future versions of GProM will feature a parser and interpreter for this language, and an *RSL repository* for storing rewrite scripts. Rewrites can either be activated on a per query basis or triggered automatically based on conditions specified in the rules. Figure 9 shows how the RSL engine will interact with the other modules of GProM.

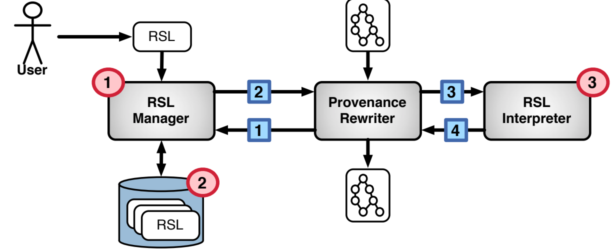


Figure 9: Interaction of a module (provenance rewriter) with the RSL engine. To process an input, the module retrieves applicable RSL scripts from the RSL repository ② using the RSL manager ①. The RSL scripts and input AGM are then send to the RSL interpreter ③. The interpreter applies the RSL rules and sends back a modified AGM model to the module.

EXAMPLE 1. *As an example, consider a simplification rewrite rule that uses a textbook relational algebra equivalence to merge adjacent selection operators into a single selection using conjunction. In relational algebra this rule can be expressed as* $\sigma_{C_1}(\sigma_{C_2}(q)) \rightarrow \sigma_{C_1 \wedge C_2}(q)$. *The RSL language design is based on lessons learned from pattern-based graph rewrites and tree query languages (e.g., XQuery). The properties of an AGM operator (node in the graph) are represented as a set of key-value pairs (e.g., the type of the operator) accessible using the* `->` *operator. The* `FOR` *clause of a rewrite rule matches subgraphs in the AGM model and binds node variables. In the implementation of the example rule shown below, we match a pattern of three nodes q, c, and g, where c is a child of q and g is a child of c. The* `WHERE` *clause filters out matches that do not fulfill the condition specified in this clause. In the example, both q and c have to be relational selection operators (checked by accessing the node's* `type` *field). The* `REWRITE` *clause defines how to rewrite each matched subgraph by constructing a subgraph that can contain both matched input graph nodes and newly constructed nodes. The semantics of RSL rules is that of a step-wise fix-point computation, i.e., as long as at least one match is found, we non-deterministically pick a match and apply the rewrite. Matches are reevaluated after each rule application. We plan to add additional language features for guiding computation in the future, e.g., defining rule precedence and calling rewrite rules from within a rule.*

```
RULE mergeSelections {
  FOR q => c => g
  WHERE q->type = selection
        AND c->type = selection
  REWRITE INTO
    selection[pred = q->pred AND c->pred] => g
}
```

## C. Database Independence

One of the goals of GProM is to support a wide variety of database backends. To support this goal, we encapsulate database-specific functionality in pluggable modules. Query rewrites related to provenance computation, optimization, provenance translation, and update and transaction reenactment all operate on our AGM model. These rewrites do not need to be modified to support additional

database backends. What needs to be adapted are 1) the parser (each database vendor supports a different SQL dialect), 2) the SQL code generator (again for dialect compliance), and 3) metadata access, 4) audit log access, and 5) time travel activation. Furthermore, we may want to tweak the optimization for each individual system.

## D. Translating between Provenance Representations

As mentioned before, we will support multiple provenance models and multiple representations and variants for these models. Often it is possible to express conversion between these provenance models and their representations as queries. For example, we can use this approach to translate between provenance models such as Why-provenance or Lineage (using the definition from [6]) that can be expressed in the semi-ring framework has been discussed by Karvounarakis et al. [14].

In GProM, we implement translation between provenance models and representations as additional query rewrites. By decoupling provenance computation from translation, we significantly reduce the number of rewrite rules that need to be implemented. In particular, implementing a translation usually requires much less work than implementing the rewrites for a new provenance type. There may exist several options for translating between two different provenance types and these options may have different performance characteristics. We will use the optimizer proposed in Appendix F to determine which rewrite option to use.

EXAMPLE 2. *Consider a provenance representation similar to the one used by an early version of the Trio system. This format represents the provenance of the result of a query q as pairs of tuple identifiers. The identifier for a query result tuple t is paired with all identifiers of tuples from t's provenance. This is a concise representation of provenance, but such identifiers are meaningless to a user. GProM, and other approaches such as Karvounarakis et al. [14] or Perm [11], use attribute values from the tuples as a more informative representation of provenance. As an example for rewrite based translations consider two rewrites that translate from the identifier into the full tuple representations. The first method (assuming that provenance is computed by propagation of a relational encoding of annotations) modifies the query used for provenance computation by propagating additional attributes used in the full tuple representation. The second method joins the original query which computes the provenance as tuple identifiers with the input relations to replace tuple identifiers with the corresponding full tuples.*[7]

## E. Importing and Exporting Provenance

Several standard formats for representing provenance have been proposed (e.g., PROV [16]). To make our system interoperable with other provenance systems, we would like to be able to export provenance into these formats. The heavy lifting of this translation can be implemented as query rewrites. We would also like to be able to import provenance produced manually or by other provenance systems and propagate imported provenance in an provenance-independent way (as pioneered in Perm [11]). The user will be able to inform the system about available provenance information during import. GProM will import the provenance, transform it using the techniques discuss above, and record the association between the data and its provenance. GProM will automatically use the imported provenance for all operations accessing the imported data. The user will also be able to retroactively associate provenance with existing data.

[7] this is similar to a technique applied in Ariadne [10] for computing provenance of continuos queries.

## F. Heuristic and Cost-based Optimization

Previous projects using query rewrites, e.g., for provenance [11] or compiling non-relational languages into SQL (e.g., Pathfinder [12]), have demonstrated that queries produced by such rewrites often contain atypical access patterns and operator sequences (e.g., large number of unions over subqueries accessing the same input and long chains of operators). Such queries "confuse" even sophisticated DBMS optimizers.

We propose to build a general purpose heuristic optimizer for our AGM model inspired by Grust et al. [12]. Note that the goal of such an optimizer is not to compete with mature database optimizers, but rather to transform the input query into a form that can be successfully optimized by the database optimizer. For example, we may want to cluster joins in the query or remove redundant duplicate removal operators and analytical functions. Since all rewrite operations in our system operate on the AGM representation of queries, such an optimizer would benefit all provenance computations. The optimizer will also be used to decide which rewrites to apply if multiple equivalent rewrites are available. For the reasons outlined above and confirmed by our experience on rewriting nested subqueries in the Perm project [9], we cannot rely on the database optimizer to be able to detect these alternatives. Finally, we do not have to rely on purely heuristic optimization. For example, we could use the DBMS optimizer to decide between alternative rewrite options at critical points in the optimization process as long as the underlying database system supports inspection of query plans (standard database systems do support this).

One application domain for GProM are systems such Hive, Pig, Shark, Tenzing, Asterix, and many others which provide SQL or SQL-like query capabilities for BigData workloads. Query optimization in these systems is currently rather limited. For example, Hive only applies a set of heuristic plan rewrites. This is likely to change over time while these systems are becoming more mature. Until then, however, we expect our optimizer to be quite effective. The example shown in Appendix A discusses some simple heuristic optimizations.

## G. Provenance Generation and Storage Policies

We let the user decide when to store which types of provenance and how to store it. The default in GProM is to not compute any provenance information unless it is explicitly requested by the user. This is reasonable, because we can always reconstruct provenance of past operations as explained in Section 3. However, the underlying database system may not support keeping an audit log and versioned relations, or the user may not be willing to tolerate the storage overhead entailed by these features. To support provenance tracking under these conditions, we enable the user to register storage policies that define when to store which type of provenance (e.g., store the provenance of queries accessing a certain relation as a map between input and output tuple identifiers). For each executed statement, we check whether the statement matches an existing storage policy. If the conditions of a policy are fulfilled, then, in addition to executing the input operation unmodified, the provenance storage manager will call the provenance rewriter to create a query for computing the provenance of the operation and store the result according to the policy. This whole process is transparent to the user executing the statement.