Using Reenactment to Retroactively Capture Provenance for Transactions

Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, Boris Glavic

Abstract—Database provenance explains how results are derived by queries. However, many use cases such as auditing and debugging of transactions require understanding of how the current state of a database was derived by a transactional history. We present MV-semirings, a provenance model for queries and transactional histories that supports two common multi-version concurrency control protocols: snapshot isolation (SI) and read committed snapshot isolation (RC-SI). Furthermore, we introduce an approach for retroactively capturing such provenance using *reenactment*, a novel technique for replaying a transactional history with provenance capture. Reenactment exploits the time travel and audit logging capabilities of modern DBMS to replay parts of a transactional history using queries. Importantly, our technique requires no changes to the transactional workload or underlying DBMS and results in only moderate runtime overhead for transactions. We have implemented our approach on top of a commercial DBMS and our experiments confirm that by applying novel optimizations we can efficiently capture provenance for complex transactions over large data sets.

Index Terms—Databases, Provenance, Concurrency Control, Transaction Processing, Reenactment

1 Introduction

Provenance, information about the creation process and origin of data, is critical for many applications including auditing, debugging data by tracing erroneous results back to erroneous inputs, and understanding complex transformations. How to model and capture the provenance of database queries is relatively well understood. Most approaches model provenance as annotations on data (e.g., tuples) and propagate annotations to compute the annotation (provenance) of a query result. That is, the annotation of a tuple t in the result of a query records which input tuples are in tuple t's provenance and how these inputs were combined to derive tuple t. Annotation propagation techniques have been pioneered by systems such as Perm [1], DBNotes [2], Orchestra [3], and others. However, many use cases require the user to understand how data was derived by updates executed as part of concurrent transactions which is not supported by current approaches. For instance, tracing a query result tuple back to its provenance in the query input is not sufficient for auditing, because this type of provenance does not explain how the query inputs were created (i.e., inserted or updated by past transactions). Another motivating example is transaction debugging where a developer should be able to inspect the execution of a transaction after the fact to determine the cause of an erroneous outcome. This requires a provenance model that exposes the inner states of relations within transactions and their interactions with each other including concurrency anomalies which occur under non-serializable isolation levels.

- *B. S. Arab and B. Glavic are with the Department of Computer Science at the Illinois Institute of Technology, Chicago, IL, 60616, USA. E-mail: mail: barab@hawk.iit.edu, bglavic@iit.edu.*
- *D. Gawlick and V. Krishnaswamy are with Oracle, CA, 94065, USA. E-mail: {dieter.gawlick, vasudha.krishnaswamy}@oracle.com.*
- *V. Radhakrishnan is with LinkedIn, CA, 94043, USA. E-mail: vradhakrishnan@linkedin.com.*

Given the lack of support for transactional provenance, users resort to the *audit logging* and *time travel* functionality natively supported by many DBMS (e.g., Oracle, DB2, SQLServer) for their auditing and debugging needs. Time travel enables access to the transaction time history of relations, i.e., the user can query past committed versions of the database. An audit log records which SQL statements were executed by which user at which time and as part of which transaction. Note that *command logs* can serve the same purpose as audit logs in our framework as they both store which statement was executed by which transaction at what time. While these features can unearth facts about past operations and database states, there are limitations. For example, these features can not be used to track dependencies based on read operations, e.g., how the tuples created by an `INSERT INTO SELECT` ... depend on the tuples accessed by the `SELECT` query. Also, they can not expose which statements of a transaction affected a tuple which is important for debugging transaction execution. Our approach overcomes these limitations. We focus on transactions executed using the *snapshot isolation* (*SI*) concurrency control protocol and the *read committed* variant of this protocol (*RC-SI*).

Snapshot Isolation. Many DBMS such as PostgreSQL, Oracle, and MSSQL support SI. Under SI [4] each transaction T sees a snapshot of the database containing changes of transactions that have committed before T started and T's own changes. Using SI, reads never block concurrent reads or writes, because each transaction sees a consistent database version as of its start. To support snapshots, old tuple versions cannot be deleted until all transactions that may need them have finished. Typically, this is implemented by storing multiple timestamped versions of each tuple and assigning a timestamp to every transaction as of its start that determines which database version it will see (its snapshot). Concurrent writes are allowed under SI. However, if several concurrent transactions attempt to write the same data item,

T	SQL	Time
T_5	`UPDATE Account SET bal = bal + 100` `WHERE typ = 'Savings';`	10
T_6	`UPDATE Account SET bal = bal - 1500` `WHERE cust = 'Alice' AND typ = 'Checking';`	11
T_5	`UPDATE Account SET bal = bal + 300` `WHERE typ = 'Savings' AND bal > 5000 ;`	12
T_5	`COMMIT;`	13
T_6	`INSERT INTO Overdraft` `(SELECT cust, a1.bal + a2.bal` `FROM Account a1, Account a2` `WHERE a1.cust = 'Alice' AND a1.cust = a2.cust` `AND a1.typ≠a2.typ AND a1.bal + a2.bal < 0);`	14
T_6	`COMMIT;`	15

Fig. 1: Example audit log for a transactional history

(a) Database before execution of T_5 and T_6

Account

	cust	typ	bal
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400
$C^2_{T_1,4}(I^2_{T_1,3}(x_2))$	Alice	Savings	1000
$C^3_{T_2,3}(I^3_{T_2,1}(x_3))$	Peter	Savings	4990

Overdraft

cust	bal

(b) Database after execution of T_5

Account

	cust	typ	bal
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390

(c) Database after execution of T_6

Account

	cust	typ	bal
$C^1_{T_6,16}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1))))$	Alice	Checking	-1100
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390

Overdraft

	cust	bal
$C^4_{T_6,16}(I^4_{T_6,15}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)) \cdot C^2_{T_1,4}(I^2_{T_1,3}(x_2)))))$	Alice	-100

Fig. 2: Running example database states

only one will be allowed to commit. This is often implemented using write locks that are held until transaction commit. A transaction T waiting for a lock is aborted if the transaction T' holding the lock commits (and continues if T' aborts). SI corresponds to isolation level `SERIALIZE` in systems such as Oracle and older versions of PostgreSQL.

Read Committed Snapshot Isolation (RC-SI). Under statement-level snapshot isolation or RC-SI, each statement of a transaction T sees previous changes of T and of concurrent transactions that committed before the start of the statement. To guarantee that each statement sees a consistent snapshot, a statement waiting for a write-lock is restarted when the transaction holding the lock commits. We refer to this variant of SI as *read-committed snapshot isolation* (**RC-SI**), because it corresponds to isolation level `READ COMMITTED` in, e.g., Oracle and PostgreSQL. Note that some databases (e.g., PostgreSQL) resume the execution of the statement when a lock is released instead of restarting it.

Example 1. *Fig. 2a shows an example database storing information about banking accounts and overdrafts. Ignore the annotations to the left of each tuple for now. Suppose Bob executed the Transactions T_5 shown in Fig. 1 under SI. Bob implemented a policy of giving a \$100 bonus to all savings accounts and an additional \$300 bonus to all savings accounts with a balance higher than \$5000. The database instance after the execution of Transaction T_5 is shown in Fig. 2b. Attribute values affected by an update are highlighted in red. Meanwhile, Alice did withdraw money (\$1500) from her checking account which triggered Transaction T_6. This transaction inserts an overdraft record into the relation `Overdraft(cust,bal)` since the total balance of Alice's accounts is negative after the withdrawal. The states of the `Account` and `Overdraft` relations after the execution of both transactions are shown in Fig. 2c. Alice, surprised to receive an overdraft notice, checks her account. She observes that the total balance of her accounts is positive and, thus, she should not have received the \$100 overdraft. This unexpected result is caused by a concurrency anomaly called write-skew [4] which can occur under SI. Recall that under SI each Transaction T executes over a private snapshot which contains changes made by transactions that committed before T's start. Hence, Transactions T_6 sees the previous balance of \$1000 for Alice's savings account and after the withdrawal of \$1500 from her checking account, it computes a total balance of $1000 + (-1100) = -100 < 0$.*

Auditing or debugging errors such as the one illustrated in the example above is virtually impossible without access to past database states and operations. For the above example, an audit log would provide information as shown in Fig. 1 while time travel gives a user access to the database states as shown in Fig. 2. However, these database states are not very helpful in determining the cause of the overdraft, because Alice's total account balance is non-negative after the execution of both transactions. Technically, once the error is detected, a user with a deep understanding of SI may be able to recognize that this particular interleaving of operations can lead to a write-skew. However, even for a power user it would be challenging to determine the cause for such errors if several other transactions were run concurrently with the transactions involved in the error. Thus, this example motivates the need for capturing the provenance of tuples that are updated by concurrent transactions. We now give a brief introduction of our provenance model for transactions and then demonstrate how it can be used to understand unexpected results.

1.1 A Provenance Model For Transactions

Our provenance model called *Multi-version semirings* (*MV-semirings*) records provenance as annotations on tuples. While there are existing solutions for computing the provenance of updates [3], [5], [6], these approaches do not support transactions and are not integrated with provenance for queries. The annotation of a tuple t in our model is a symbolic expression that encodes **1)** which tuples where used to derive t (variables, e.g., x_1, x_2, $\ldots$ represent tuples) **2)** how these tuples have been combined (addition and multiplication represent alternative and joint use of inputs) **3)** which DML operations executed by which transactions at which time did create the annotated tuple version (represented by function symbols which we call *version annotations*). For example, the annotation $I^{id}_{T,\nu}(x)$ represents the fact that an insert of transaction T executed at time ν created a tuple with identifier id that is represented as variable x. Similarly, U, D, and C represent update, delete, and commit operations. Note that we do not consider provenance dependencies at

	Account			Provenance for the First Update			Provenance for the Second Update			u_1	u_2
	cust	**typ**	**bal**	**P(cust,u_1)**	**P(typ,u_1)**	**P(bal,u_1)**	**P(cust,u_2)**	**P(typ,u_2)**	**P(bal,u_2)**	$\mathcal{U}_1$	$\mathcal{U}_2$
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(c_2)))$	Alice	Savings	1100	Alice	Savings	1000	Alice	Savings	1100	T	F
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(c_3))))$	Peter	Savings	5390	Peter	Savings	4990	Peter	Savings	5090	T	T

Fig. 3: Relational encoding of the provenance and intermediate results for relation `Account` with respect to Transaction T_5.

the application side in this work. For instance, consider an application that runs a query, stores the result in a client-side variable, and then uses the variable in an update statement. Detecting such dependencies requires tracking provenance of procedural programming languages which is beyond the scope of this work. The annotations shown to the left of each tuple in Fig. 2 are the provenance annotations of these tuples in our model. For example, consider $C^1_{T_1,4}(I^1_{T_1,2}(x_1))$, the annotation of the first tuple shown in Fig. 2a. Based on this annotation we know that this tuple has tuple identifier 1 and was created by an insert of a Transaction T_1 executed at time 2. This transaction did commit this tuple version at time 3. Typically, a user would like to be able to drill down into a part of a history instead of tracing the origin of a tuple through the whole history of the database. Our model supports this type of drill down by replacing subexpressions in an annotation with fresh variables to prune parts of the history from a tuple's annotation. For example, if the user wants to focus her investigation on the operations of Transaction T_5 then this is achieved in our model by replacing subexpressions enclosed in commit operations by transactions which committed before T_5's start with fresh variables and removing tuples that were not affected by a transaction. For example, $C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$, the annotation of the second tuple in Fig. 2b, would be replaced with $C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(c_2)))$ for some fresh variable c_2. To ease understanding of provenance expressed in our model and to enable queries over provenance, we define relational encodings of such provenance annotations. These encodings are quite flexible in that the user can choose, e.g., whether 1) version annotations are shown and 2) how tuple versions are represented (either as a pair of tuple identifier and timestamp or using the tuple's attribute values). For instance, Fig. 3 shows the relational encoding of the provenance of the `Account` relation restricted to operations by Transaction T_5 (annotations in our model are shown on the left). Here the user has chosen the option to encode tuples in the provenance by their attribute values and to show intermediate tuple versions produced by the transaction. We explain this example in more detail below.

Tracking Read and Write Dependencies of Tuples. One way to debug the example error is to determine which tuple versions were used to derive the erroneous overdraft tuple. This would unveil that it was computed based on Alice's savings account balance before the bonus was added by Transaction T_5. Note that this is a read dependency. The second account tuple version from Fig. 2a was read by the `INSERT INTO Overdraft SELECT ...` statement which was executed by Transaction T_6. In our model, this is encoded in the annotation of the new overdraft tuple which includes $C^2_{T_1,4}(I^2_{T_1,3}(x_2))$, the annotation of the account tuple from which it was derived. Time travel can expose write dependencies caused by updates if a tuple can be identified across versions (e.g., the DBMS uses immutable tuple identifiers). However, it cannot be used to track read dependencies.

Tracking Applications of Updates. Understanding which statements of which transactions were involved in the derivation of a tuple version is important to answer auditing questions such as "What data was affected by statements executed by a compromised user account?". Audit logs record which statements were executed and when they were executed. However, even when this information is correlated with a transaction time history using time travel, it is highly non-trivial to answer such questions since 1) only write tuple dependencies are available and 2) time travel only exposes committed database states. In our model, this information is encoded in the nesting of version annotations. For example, based on the annotation of the second tuple in the database state shown in Fig. 2b we know that this tuple version was created by an update of Transaction T_5 which was applied to a tuple created by an insert of Transaction T_1. Furthermore, we know when these operations were executed and when these transactions did commit. The annotation of the new overdraft tuple in the running example shows that none of the updates of Transaction T_5 (adding the account bonuses) did affect the tuples on which the overdraft is based on.

Exposing Intermediate States. Our relational encoding of provenance can expose intermediate states of relations produced by transactions, e.g., the state of a relation after a particular operation. Furthermore, the encoding records dependencies across such states. This is useful investigating whether and how an update modified a tuple.

Example 2. *Recall that Fig. 3 shows the provenance of the* `Account` *relation w.r.t. Transaction T_5. The provenance annotation of each tuple is encoded in additional attributes that are added to the schema. A* "provenance" *attribute $P(attr, u)$ stores the value of attribute* attr *for the version of a tuple in the provenance seen by the update statement u. We use u_1 and u_2 to denote the updates of Transaction T_5. For instance, attributes $P(bal, u_1)$ and $P(bal, u_2)$ store the balance of a tuple before the execution of update u_1 respective u_2. The boolean attribute $\mathcal{U}_i$ stores whether the version annotation for update u_i is part of the provenance, i.e., whether u_i affected a tuple. Suppose manager Tom wants to know which accounts received the \$300 bonus implemented by the update u_2 and what was the previous balance of these accounts before the bonus. This question can be answered by the SQL query shown below where* Prov *denotes the encoding from Fig. 3.*

```
SELECT P(cust,u₂), P(bal,u₂) FROM Prov WHERE 𝒰₂ = True
```

Understanding Errors Caused by Concurrency Anomalies. We have demonstrated in [7] how our model can be used to debug errors caused by concurrency anomalies such as the *write-skew* [4] in the running example. Errors caused by anomalies are common, but hard to debug since they may only occur for a particular interleaving of transactions.

1.2 Capturing Provenance With Reenactment

We have developed a provenance capture mechanism that produces the relational encoding of our provenance model for a provenance request and have implemented this mechanism in our provenance database middleware called *GProM* (https://github.com/IITDBGroup/gprom). We use *reenactment*, a novel technique for replaying a transactional history (or parts thereof) using queries instrumented to capture provenance. Reenactment retroactively captures the provenance of tuple versions produced by a history. Notably, our approach does not require any eager materialization of provenance during transaction execution. Hence, we avoid paying the runtime and storage overhead of provenance capture for every transaction executed by the system. Reenactment solely relies on the information provided by audit logs and time travel and is expressible in SQL. Many users that would be interested in provenance already use these features. Furthermore, as we demonstrate in Sec. 7.3, the overhead of activating these features is quite manageable (less than 20% for the workloads we considered). Importantly, our approach does not require any modifications of the underlying DBMS or transactional workload.

We have introduced our vision of GProM in [8] and have presented our approach for RC-SI in [9]. The main contributions of this work are:

- We introduce *multi-version semirings* (MV-semirings), a provenance model for database queries and transactions. In our model, tuples are annotated with symbolic expressions that model dependencies among tuples and which operations affected a tuple. We use a relational encoding of our model for querying provenance.
- We introduce *reenactment*, a technique for replaying a transactional history using queries. The reenactment query for a transaction T is equivalent to T within the context of a history under MV-semiring semantics, i.e., it returns the same database state and has the same provenance. We reduce reenactment queries with MV-semiring semantics to queries in standard SQL that return a relational encoding of provenance.
- We develop optimizations for reenacting SI and RC-SI transactions including alternative ways of encoding reenactment as SQL queries and filtering unrelated information from the provenance early on.
- Our experiments demonstrate that 1) provenance capture based on reenactment is very efficient and scales to large databases, complex transactions, and large number of updates; and 2) the storage and runtime overhead incurred by time travel and audit logging is tolerable and significantly smaller than the overhead of eagerly capturing provenance during transaction execution.

The remainder of this paper is organized as follows. We review related work in Sec. 2 and introduce our provenance model in Sec. 3. We define an annotated semantics of SI and RC-SI transactional histories in Sec. 4, cover reenactment in Sec. 5, discuss implementation and optimizations in Sec. 6, present experimental results in Sec. 7, and conclude in Sec. 8.

2 Related Work

Several provenance models for relational queries have been introduced in related work including Why-provenance, minimal Why-provenance [10], and Lineage [11]. Provenance polynomials introduced by Green et al. [12] generalize these provenance models for positive relational algebra queries ($\mathcal{RA}^+$). Green's semiring annotation framework has been the target of extensive research including relations annotated with annotations from multiple semirings [13], rewriting queries to minimize provenance [14], factorization of provenance polynomials [15], extraction of provenance polynomials from the PI-CS [1] model, and extensions for aggregation [16] and set difference [17]. Our MV-semirings generalize this model to support updates and transactions. Similar to our approach, Lipstick [18], LogicBlox [19], DBNotes [2], Perm [1], and many other systems encode provenance annotations in a standard data model and use query instrumentation to propagate these annotations. Several papers [5], [6] study provenance for updates, e.g., Vansummeren et al. [5] compute provenance for SQL DML statements. However, these approaches modify updates to eagerly capture provenance, do not track provenance through concurrent transactions, and are often not integrated with provenance for queries. We take interactions among transactions into account using a generalization of the semiring model for transactions. Command logs/audit logs provide information about the update statements and transactions that were executed, but they do not directly encode data dependencies and tuple versions.

3 The MV-Semiring Model

We now formally introduce our MV-semiring model that extends $\mathcal{K}$-relations with support for transactions.

$\mathcal{K}$-relations. MV-semirings are based on the semiring provenance framework [12]. In this framework, relations are annotated with elements from an annotation domain K. Let $\mathcal{K} = (K, +_{\mathcal{K}}, \cdot_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ be a commutative semiring. A $\mathcal{K}$-*relation* R is a (total) function that maps tuples to elements from $\mathcal{K}$ with the convention that tuples mapped to $0_{\mathcal{K}}$, the 0 element of the semiring, are not in the relation. A structure $\mathcal{K}$ is a commutative semiring if it fulfills the equational laws shown on the top of Fig. 4. Depending on the domain K, the annotations can serve different purposes. For instance, the semiring $\mathbb{N}$, natural numbers with standard arithmetics, corresponds to bag semantics. If a tuple t occurs twice in a bag semantics relation R, then this tuple would be annotated with 2 in the $\mathbb{N}$-relation corresponding to R. As we will see in the following, the operators of the positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations are defined by combining input annotations using the $+_{\mathcal{K}}$ and $\cdot_{\mathcal{K}}$ operations where addition represents alternative use of inputs (e.g., union) and multiplication denotes conjunctive use (e.g., join).

Provenance polynomials. Provenance polynomials (semiring $\mathbb{N}[X]$), polynomials over a set of variables X which represent tuples in the database, model an expressive type of provenance by encoding how a query result tuple was derived by combining input tuples. Using $\mathbb{N}[X]$, every tuple in an instance is annotated with a unique variable $x \in X$ and query results are annotated with polynomials over these variables. For example, a tuple that was derived by joining tuples identified by x_1 and x_2 would be annotated with $x_1 \cdot x_2$. Since our main concern is provenance, we mostly limit the discussion to $\mathbb{N}[X]$ and its MV extension.

Laws of commutative semirings

$$k + 0_{\mathcal{K}} = k \qquad k \cdot 1_{\mathcal{K}} = k \qquad \text{(neutral elements)}$$

$$k + k' = k' + k \qquad k \cdot k' = k' \cdot k \qquad \text{(commutativity)}$$

$$\begin{matrix} k + (k' + k'') = (k + k') + k'' \\ k \cdot (k' \cdot k'') = (k \cdot k') \cdot k'' \end{matrix} \qquad \text{(associtivity)}$$

$$k \cdot 0_{\mathcal{K}} = 0_{\mathcal{K}} \qquad \text{(annihilation through 0)}$$

$$k \cdot (k' + k'') = (k \cdot k') + (k \cdot k'') \qquad \text{(distributivity)}$$

Evaluation of expressions with operands from K

$$k + k' = k +_{\mathcal{K}} k' \quad k \cdot k' = k \cdot_{\mathcal{K}} k' \quad (\text{if } k \in K \wedge k' \in K)$$

Equivalences involving version annotations

$$\mathcal{A}(0_{\mathcal{K}}) = 0_{\mathcal{K}} \qquad \mathcal{A}(k + k') = \mathcal{A}(k) + \mathcal{A}(k')$$

Fig. 4: Equivalence relations for $\mathcal{K}^{\nu}$

MV-semirings. MV-semirings are a specific class of semirings that encode the derivation of tuples based on a history of transactional updates. For each semiring $\mathcal{K}$, there exists a corresponding semiring $\mathcal{K}^{\nu}$, e.g., $\mathbb{N}[X]^{\nu}$ is the MV-semiring corresponding to the provenance polynomials semiring $\mathbb{N}[X]$. Since $\mathbb{N}$ encodes bag semantic relations, $\mathbb{N}^{\nu}$ represents bag semantics with embedded history. Fig. 2 shows examples of $\mathbb{N}[X]^{\nu}$ annotations. In these symbolic expressions, variables (e.g., x_1, x_2, ...) represent freshly inserted tuples and uninterpreted function symbols (the aforementioned *version annotations*) encode which operations were applied to the tuple. The nesting of version annotations records the sequence of operations that created a tuple version.

Version Annotations. 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 $\nu - 1$ by transaction T affected a previous version of a tuple with identifier id and previous provenance k. Assuming domains of tuple identifiers $\mathbb{I}$, version identifiers $\mathbb{V}$, and transaction identifiers $\mathbb{T}$, let $\mathbb{A}$ denote the set of all version annotations:

$$I^{id}_{T,\nu}, U^{id}_{T,\nu}, D^{id}_{T,\nu}, C^{id}_{T,\nu} \quad \text{for} \quad id \in \mathbb{I}, \nu \in \mathbb{V}, T \in \mathbb{T} \qquad (1)$$

Example 3. *Consider the $\mathbb{N}[X]^{\nu}$-relation* `Account` *in Fig. 2b. The second tuple is annotated with $C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$, i.e., it was created by an update of Transaction T_5, which updated a tuple that was inserted by T_1. Based on the outermost commit annotation, this tuple version is visible to transactions starting after version* 13.

MV-semiring Annotation Domain. Fixing a semiring $\mathcal{K}$, we define the domain of semiring $\mathcal{K}^{\nu}$ based on the set of finite symbolic expressions P whose syntax is defined by the grammar shown below where $k \in K$ and $\mathcal{A} \in \mathbb{A}$.

$$P := k \mid P + P \mid P \cdot P \mid \mathcal{A}(P) \qquad (2)$$

The semantics of these expressions is defined in Def. 1 and Fig. 4. Note that $+$ and $\cdot$ in these expressions are used to encode that a tuple depends on multiple input tuples, e.g., a query such as the one used by the insert of example Transaction T_6 or an update that modifies two tuples that are distinct in the input to be the same in the output (e.g., `UPDATE Account SET typ = 'Savings'`). For example, consider a query $\Pi_{typ}(Account)$ evaluated over the instance from Fig. 2a. The result tuple `(Savings)` is derived from the second and third tuple in the `Account` table (the two tuples with this value in attribute `typ`) and, thus, would be annotated with $C^2_{T_1,4}(I^2_{T_1,3}(x_2)) + C^3_{T_2,3}(I^3_{T_2,1}(x_3))$ where addition represents alternative use of these two tuples. We would expect certain symbolic expressions produced by the grammar above to be equivalent, e.g., expressions in the embedded semiring $\mathcal{K}$ can be evaluated using the operations of the semiring ($k_1 + k_2 = k_1 +_{\mathcal{K}} k_2$) and updating a non-existing tuple does not lead to an existing tuple ($\mathcal{A}(0_{\mathcal{K}}) = 0_{\mathcal{K}}$). This is achieved by defining domain K^{ν} as the set of equivalence classes (denoted as $[]_{\sim}$) for expressions in P based on the equivalences shown in Fig. 4.

Definition 1. *Let $\mathcal{K} = (K, +_{\mathcal{K}}, \cdot_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ be a commutative semiring. The MV-semiring $\mathcal{K}^{\nu}$ is the structure*

$$\mathcal{K}^{\nu} = (K^{\nu}, +_{\mathcal{K}^{\nu}}, \cdot_{\mathcal{K}^{\nu}}, [0_{\mathcal{K}}]_{\sim}, [1_{\mathcal{K}}]_{\sim})$$

where $\cdot_{\mathcal{K}^{\nu}}$ and $+_{\mathcal{K}^{\nu}}$ are defined as

$$[k]_{\sim} \cdot_{\mathcal{K}^{\nu}} [k']_{\sim} = [k \cdot k']_{\sim} \qquad [k]_{\sim} +_{\mathcal{K}^{\nu}} [k']_{\sim} = [k + k']_{\sim}$$

Addition and multiplication output a symbolic expression by connecting the inputs with $+$ or $\cdot$ and then output the equivalence class for this expression. Consider semiring $\mathbb{N}$, which encodes bag semantics relations by annotating each tuple with a natural number representing its multiplicity. For example, assume a tuple t is annotated with the $\mathbb{N}^{\nu}$-expression $U^1_{T,\nu}(3 \cdot 6)$. Here 3 and 6, elements from the embedded semiring $\mathbb{N}$, represent multiplicities. Applying equivalence $k \cdot k' = k \cdot_{\mathcal{K}} k'$, we can evaluate $3 \cdot 6 = 3 \cdot_{\mathbb{N}} 6 = 18$. Thus, t appears with multiplicity 18 and was updated by an update (U) of transaction T. The update was run at time $\nu - 1$ and, thus, the tuple became valid at time ν.

Normal Form and Admissible Instances. $\mathcal{K}^{\nu}$ expressions admit a (non unique) normal form representing an element $k \in K^{\nu}$ as a sum $\sum_{i=0}^{n} k_i$ where none of the k_i contains any addition operations. Any $\mathcal{K}^{\nu}$ element can be brought into this normal form by applying the equivalences from Fig. 4. Intuitively, each summand in the normal form corresponds to a tuple under bag semantics. Thus, we will sometimes refer to a summand as a tuple version. Assuming an arbitrary, but fixed, order over such summands we can address elements in such a sum by position. We use $n(k)$ to denote the number of summands in a normalized annotation k and $k[i]$ to refer to the ith element in the sum according to the assumed order. We use this normal form to define updates and transactions. Note that some expressions produced by the grammar in Equation (2) can not be produced by any transactional history. For instance, $U^1_{T,11}(C^1_{T,10}(\ldots))$ is invalid, because it implies that Transaction T executed an update after its commit. A $\mathcal{K}^{\nu}$ instance is **admissible** if either 1) it is empty or 2) it is the result of evaluating a transactional history (formally defined later) over an admissible instance.

Queries and Update Operations. We extend the standard definition of positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations [12] with an operator $\{t \to k\}$ that creates a singleton relation (a tuple t annotated with k). Let $t.A$ denote the projection of a tuple t on a list of projection expressions A and $t[R]$ to denote the projection of t on the attributes of relation R. For a condition θ and tuple t, $\theta(t)$ denotes a function that returns $1_{\mathcal{K}}$ if $t \models \theta$ and $0_{\mathcal{K}}$ otherwise.

Definition 2. *Let R and S denote $\mathcal{K}$-relations, t, t' denote tuples, and $k \in K$. The operators of $\mathcal{RA}^+$ are defined as:*

$$\Pi_A(R)(t) = \sum_{t':t'.A=t} R(t') \qquad (R \cup S)(t) = R(t) + S(t)$$

$$\sigma_\theta(R)(t) = R(t) \cdot \theta(t) \qquad \{t' \to k\}(t) = \begin{cases} k & \text{if } t = t' \\ 0_{\mathcal{K}} & \text{else} \end{cases}$$

$$(R \bowtie S)(t) = R(t[R]) \cdot S(t[S]) \qquad (\text{for } R \cup S \text{ tuple } t)$$

Updates are also defined using semiring operations. However, in contrast to queries, they create version annotations. We support updates corresponding to SQL constructs `INSERT`, `UPDATE`, `DELETE`, and `COMMIT`. An operation is executed at a time ν as part of a transaction T. Updates take a normalized, admissible $\mathcal{K}^\nu$-relation R as an input and return an updated version of R. An insertion $\mathcal{I}[Q, T, \nu](R)$ inserts the result of query Q into relation R. The annotations of inserted tuples are wrapped in version annotations and are assigned fresh identifiers (id_{new}). An update operation $\mathcal{U}[\theta, A, T, \nu](R)$ applies projection expressions in A to tuples that fulfill condition θ. Both $\mathcal{U}[\theta, A, T, \nu](R)$ and delete $\mathcal{D}[\theta, T, \nu](R)$ wrap annotations of tuples fulfilling condition θ in version annotations. A commit $\mathcal{C}[T, \nu](R)$ adds commit version annotations. We use $\nu(u)$ to denote the version when an update u was executed and $id(k)$ to denote the id of the outermost version annotation of $k \in K^\nu$ (well-defined for admissible $\mathcal{K}^\nu$-relations).

Definition 3. *Let R be a normalized, admissible $\mathcal{K}^\nu$-relation. Let A be a list of projection expressions with the same arity as R and id_{new} to denote a fresh id. Let Q be a query over a database D such that for every $\{t \to k\}$ operation in Q we have $k \in \mathcal{K}$. We define updates over $\mathcal{K}^\nu$-relations as:*

$$\mathcal{U}[\theta, A, T, \nu](R)(t) = R(t) \cdot (\neg\theta)(t) + \sum_{t':t'.A=t} \sum_{i=0}^{n(R(t'))} U_{T,\nu+1}^{id(R(t')[i])}(R(t')[i]) \cdot \theta(t')$$

$$\mathcal{I}[Q, T, \nu](R)(t) = R(t) + I_{T,\nu+1}^{id_{new}}(Q(D)(t))$$

$$\mathcal{D}[\theta, T, \nu](R)(t) = R(t) \cdot (\neg\theta)(t) + \sum_{i=0}^{n(R(t))} D_{T,\nu+1}^{id(R(t)[i])}(R(t)[i]) \cdot \theta(t)$$

$$\mathcal{C}[T, \nu](R)(t) = \sum_{i=0}^{n(R(t))} \text{COM}[T, \nu](R(t)[i])$$

$$\text{COM}[T, \nu](k) = \begin{cases} C_{T,\nu+1}^{id}(k) & \text{if } k = X_{T,\nu'}^{id}(k') \wedge X \in \{U, I, D\} \\ k & \text{else} \end{cases}$$

As a convention, if an attribute a is not listed in the list of expressions A of an update then $a \to a$ is assumed. For instance, the first update of example transaction T_5 would be written as $\mathcal{U}[typ = \text{'}Savings\text{'}, bal + 100 \to bal, T_5, 10](Account)$. What tuple identifiers (id_{new}) are assigned by inserts to new tuples is irrelevant as long as identifiers are sufficient for uniquely identifying tuples (see [20]).

Properties of MV-semirings. We now discuss several properties of our model. A formal treatment including proofs is presented in our technical report [20]. An important property of provenance polynomials is that the result of a query Q in any semiring $\mathcal{K}$ can be computed from the $\mathbb{N}[X]$ result of Q by replacing variables in polynomials with elements from $\mathcal{K}$ and evaluating the resulting expression in $\mathcal{K}$. This property was proven by Green et al. [12] by demonstrating 1) that the process described above is a semiring homomorphism, i.e., a mapping $h : \mathbb{N}[X] \to \mathcal{K}$ that agrees with semiring operations; and 2) that homomorphisms commute with queries. In [20], we demonstrate that any homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a "history-preserving" homomorphism $h^\nu : {\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$ by applying h to each $\mathcal{K}_1$ element in a ${\mathcal{K}_1}^\nu$ element k. Lifted homomorphisms also commute with updates and transactional histories. Thus, $\mathbb{N}[X]^\nu$ enjoys the same generality property among MV-semirings as $\mathbb{N}[X]$ does for semirings. Any $\mathcal{K}^\nu$-relation can be transformed into a corresponding $\mathcal{K}$-relation, by "evaluating" the history embedded in a $\mathcal{K}^\nu$ element k. This is achieved through a homomorphism $h_U : \mathcal{K}^\nu \to \mathcal{K}$ that evaluates the symbolic expression k by interpreting version annotations as functions from $\mathcal{K} \to \mathcal{K}$ and by interpreting the operations $+$ and $\cdot$ in semiring $\mathcal{K}$. Insert, commit, and update annotations are interpreted as the identify function on K whereas deletion annotations are interpreted as the function that maps every input to $0_{\mathcal{K}}$. For example, consider $C_{T_6,16}^4(I_{T_6,15}^4(U_{T_6,12}^1(C_{T_1,4}^1(I_{T_1,2}^1(x_1)) \cdot C_{T_1,4}^2(I_{T_1,3}^2(x_2)))))$, the annotation of the overdraft tuple in Fig. 2c. By interpreting the version annotations as the identity function on $\mathbb{N}[X]$, this expression would be transformed into the $\mathbb{N}[X]$-expression $x_1 \cdot x_2$. Furthermore, we demonstrate that $Q \equiv_{\mathbb{N}[X]^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}^\nu} Q'$ for any MV-semiring $\mathcal{K}^\nu$. Thus, the equivalence between histories and reenactment queries that we prove for $\mathbb{N}[X]^\nu$ in Sec. 5 implies equivalence for any $\mathcal{K}^\nu$. In particular, reenactment works for bag semantics (semiring $\mathbb{N}^\nu$).

4 Transactions and Histories

We now define transactional histories for $\mathcal{K}^\nu$-databases under SI and RC-SI in a way that is backward compatible to the bag semantics version of SI/RC-SI. A **transaction** $T = \{u_1, \ldots, u_n, c\}$ is a sequence of update operations followed by a commit operation (c) with $\nu(u_i) < \nu(u_j)$ for $i < j$. A **history** $H = \{T_1, \ldots, T_n\}$ over a database D is a set of transactions over D with at most one operation at each version ν. We use $Start(T) = \nu(u_1)$ and $End(T) = \nu(c)$ to denote the time when transaction T did start (respective did commit). Note that an update u in our algebra records explicitly when it was executed in its version identifier $\nu(u)$. We use $R[\nu]$ to denote the state of relation R at time ν produced by the history. Note that $R[\nu]$ only contains committed changes. $R[T, \nu]$ denotes relation R as seen by transaction T at time ν. Our version annotations do not explicitly store when a tuple version was invalidated by an update. Invalidation is implicitly encoded in the nesting of version annotations.

Definition 4. *Let H be a history over a database D. The version $R[\nu]$ of relation $R \in D$ at time ν and the version $R[T, \nu]$ of relation R visible within transaction $T \in H$ at time ν are defined in Fig. 5 and 6 for SI and RC-SI, respectively.*

4.1 Snapshot Isolation Histories

A transaction T under SI sees 1) its own updates and 2) the updates of transactions that have committed before

(a) Historic Relation $R[T,\nu]$

$$R[T,\nu] = \begin{cases} \emptyset & \text{if } \nu < Start(T) \\ R[\nu] & \text{if } Start(T) = \nu \\ u(R[T,\nu-1]) & \text{if } \exists u \in T : \nu(u) = \nu - 1 \wedge u \text{ updates } R \wedge End(T) \neq \nu - 1 \\ \mathcal{C}[T,\nu-1](R[T,\nu-1]) & \text{if } End(T) = \nu - 1 \\ R[T,\nu-1] & \text{else} \end{cases}$$

(b) $R[\nu]$: Committed Tuple Versions at Time ν

$$R[\nu](t) = \sum_{T \in H \wedge End(T) < \nu} \sum_{i=0}^{n(R[T,\nu](t))} R[T,\nu](t)[i] \cdot \text{VALIDAT}(T, t, R[T,\nu](t)[i], \nu)$$

(c) Valid Tuple Versions from Transaction T at ν

$$\text{VALIDAT}(T,t,k,\nu) = 1 \text{ if } k = C^{id}_{T,\nu'}(k') \wedge (\neg\exists T' \neq T : End(T') \leq \nu \wedge \text{UPDATED}(T',t,k,\nu)), 0 \text{ otherwise}$$

(d) Tuple Versions Updated By Transaction T

$$\text{UPDATED}(T,t,k,\nu) \Leftrightarrow \exists u \in T, t', i, j : \nu(u) < \nu \wedge R[T,\nu(u)](t)[i] = k \wedge R[T,\nu(u)+1](t')[j] = X^{id}_{T,\nu(u)+1}(k) \wedge X \in \{U, D\}$$

Fig. 5: SI historic database definition

(a) Historic Relation $R[T,\nu]$

$$R[T,\nu] = u(R_{ext}[T,\nu-1]) \qquad \text{if } \exists u \in T : \nu(u) = \nu - 1 \wedge u \text{ updates } R \wedge End(T) \neq \nu - 1$$

(b) $R_{ext}[T,\nu]$: Tuple Versions Visible Within Transaction T at Time ν

$$R_{ext}[T,\nu](t) = \sum_{i=0}^{n(R[\nu](t))} R[\nu](t)[i] \times \text{VALIDEX}(T,t,R[\nu](t)[i],\nu) + \sum_{i=0}^{n(R[T,\nu](t))} R[T,\nu](t)[i] \times \text{VALIDIN}(T,t,R[T,\nu](t)[i],\nu)$$

(c) Validity of Summands (Tuple Versions) Within Annotations

$$\text{VALIDIN}(T,t,k,\nu) = 1 \text{ if } \exists \nu', k', id : k = X^{id}_{T,\nu'}(k') \wedge X \in \{U, D, I\}, 0 \text{ otherwise}$$

$$\text{VALIDEX}(T,t,k,\nu) = 0 \text{ if } \text{UPDATED}(T,t,k,\nu), 1 \text{ otherwise}$$

Fig. 6: RC-SI historic database definition

$Start(T)$. The first condition is encoded in the definition of $R[T,\nu]$ and the second one in the definition of $R[\nu]$.

Relation Versions Visible Inside an SI Transaction. $R[T,\nu]$ contains the result of applying the latest update of T before ν to the version valid before the update. As a convention, we define $R[T,\nu] = \emptyset$ if $\nu < Start(T)$. The 1st update in a transaction sees $R[Start(T)]$, the version of R containing all changes of transactions committed before T started (2nd case in Fig. 5a). We explain how to compute $R[\nu]$ below. Consider a transaction $T = u_1, \ldots, u_n, c$ and assume for simplicity that every update is modifying the same relation R. The 2nd update u_2 within the transaction will see the version of R produced by applying update u_1 to $R[Start(T)]$, the 3rd update u_3 will run over the version of R produced by u_2, and so on. This is encoded by the 3rd and 5th case in Fig. 5a. u denotes one of the operations as defined in Def. 3. If T executed an update on R at version $\nu - 1$ then $R[T,\nu]$ is the result of applying the update to $R[T,\nu-1]$. If transaction T committed at $\nu - 1$ then we apply a commit operation (Def. 3) to $R[T,\nu-1]$ (4th case). If the transaction did not execute any operation at $\nu - 1$ (including the case where $\nu > End(T)+1$) then $R[T,\nu] = R[T,\nu-1]$ (5th case).

Relation Versions Containing Committed Changes. Under SI, a transaction starting at ν will see a version of relation R that contains changes of transactions committed before ν. Recall that we use $R[\nu]$ to denote this version of R. Fig. 5b to 5d show the definition of $R[\nu]$. $R[\nu]$ can be written as a union (sum) over all tuple versions (annotations) created by transactions that committed before ν as long as the same tuple version is no included more than once. Furthermore, we should not include annotations that correspond to tuple versions which have been replaced with newer versions or were deleted. This is checked using a predicate VALIDAT.

Determining Valid Tuple Versions. VALIDAT(T, t, k, ν) evaluates to 1 if two conditions are met: 1) annotation k was produced by transaction T, i.e., the outermost version annotation in k is from T; 2) the tuple version corresponding to k was not updated (predicate UPDATED(T', t, k, ν)) by another transaction T' that committed before ν ($End(T') < \nu$).

Checking for Tuple Updates. UPDATED(T, t, k, ν) is true if transaction T has invalidated the tuple version corresponding to t annotated with k before ν. A transaction T has invalidated a summand k in an annotation of a tuple t if there exists an operation u (update or delete) within the transaction that has updated tuple t into tuple t' and $\nu(u) < \nu$. Thus, there has to exist i and j so that a summand $R[T,\nu(u)](t)[i] = k$ is in the annotation on t before the update and after the update the annotation of tuple t' contains a summand $R[T,\nu(u)+1](t')[j] = X^{id}_{T,\nu(u)+1}(k)$ where $X \in \{U, D\}$ (either a delete or update).

Example 4. *Consider $Account[T_6, 11]$, the version of relation* `Account` *from our running example visible to Transaction T_6 at version 11. Since $Start(T_6) = 11$, this version is equal to $Account[11]$. We construct $Account[11]$ by combining tuple annotations created by transactions that committed before T_6*

started as long as these tuple versions have not been invalidated by another already committed transaction. For instance, Transaction T_1 did create the annotation $k = C^1_{T_1,4}(I^1_{T_1,2}(x_1))$ on tuple $a_1 =$ (Alice, Checking, 400) as shown in Fig. 2a. VALIDAT *evaluates to* 1 *for this annotation of tuple a_1 if no transaction that committed before* 11 *has invalidated this version. Since there is no such transaction, we get $Account[11](a_1) = C^1_{T_1,4}(I^1_{T_1,2}(x_1))$.*

4.2 Read-Committed SI Histories

Under RC-SI, an update u of a transaction T sees 1) changes of previous updates of T and 2) changes of transactions that committed before $\nu(u)$. We use $R_{ext}[T,\nu]$ to denote the version of R seen by u.

Relation Versions Visible Inside a RC-SI Transaction. For RC-SI, we also apply the definition from Fig. 5a. The only difference is the 3^{rd} case: an update was executed by transaction T at time $\nu - 1$ and its modifications are reflected in $R[T,\nu]$. The modified 3^{rd} case is shown in Fig. 6a. The update sees tuple versions created by: 1) the transaction's own updates; 2) other transactions which committed before $\nu - 1$. We discuss how to compute this version of a relation R (denoted by $R_{ext}[T,\nu - 1]$) in the following.

Relation Version Visible to Updates. Fig. 6b shows the definition of $R_{ext}[T,\nu]$, the version of relation R that is visible to an update of transaction T executed at time ν. The first sum computes a version of relation R that contains all tuple versions which were created by transactions that committed before $Start(T)$ and have not been modified by a previous update of transaction T. This is checked by function VALIDEX that returns 0 if the tuple version has been replaced with a new updated version and 1 otherwise. Function VALIDEX uses predicate UPDATED(T,t,k,ν) which was already introduced in Sec. 4.1 (Fig. 5d). The second sum only considers tuple versions $R[T,\nu]$ created by previous updates of T which is checked by function VALIDIN.

Relation Versions Containing Committed Changes. We use the same definition as for SI (Fig. 5b and Fig. 5c).

Example 5. *Assume T_5 and T_6 were executed under RC-SI instead of SI. Consider $Account[T_6,14]$, the version of* `Account` *visible to the insert of T_6 at time 14. As shown in Fig. 2c, the first update of Transaction T_6 did create the annotation $U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))$ on the tuple (Alice, Checking, -1100) of* `Account`*. Therefore,* VALIDIN *returns* 1 *whereas* VALIDEX *returns* 0 *and $Account_{ext}[T_6,14]$ contains this version.*

4.3 Provenance Filtering

A tuple's annotation stores its derivation history since the origin of the database. This amount of information can be overwhelming to a user. As mentioned in Sec. 1, we can restrict provenance to tuple versions affected by a transaction. To restrict $R[T,End(T)]$, the provenance of a Transaction T for relation R, we apply two filtering steps: 1) filter out tuples that were not affected by T. In this step, every summand is removed from the annotation of a tuple if it is not wrapped in a commit annotation of T, i.e., it is not of the form $C^{id}_{T,\nu}(k)$ 2) remove parts of the provenance that correspond to operations before the start of T. Each subexpression that is wrapped in the commit annotation of a transaction $T' \neq T$ is replaced with a variable disambiguated by tuple identifiers, i.e., every subexpression $C^{id}_{T',\nu}(k)$ is substituted with $C^{id}_{T',\nu}(x_{id})$. Assume we are interested in Transaction T and $T' \neq T$. An expression $C^{id}_{T',\nu'}(I^{id}_{T',\nu''}(I^{id_1}_{T',\nu_1}(x_1) \cdot I^{id_2}_{T',\nu_2}(x_2)))$ in the annotation of a tuple updated by T would be replaced with $C^{id}_{T',\nu'}(x_{id})$.

5 REENACTMENT

Reenactment captures provenance for an update u (or transaction T) within the context of a history H by executing an annotation equivalent *reenactment query* $\mathbb{R}(u)$ (or $\mathbb{R}(T)$). Annotation equivalent ($\equiv_{\mathbb{N}[X]^\nu}$) means that such a query produces the same result and provenance. Recall that this implies equivalence for any MV-semiring $\mathcal{K}^\nu$. Since $\mathcal{RA}^+$ operators do not introduce version annotations, we define an operator for this purpose.

Definition 5. *The operator $\alpha_{X,T,\nu}(R)$ for $X \in \{I,U,D\}$ takes as input a $\mathcal{K}^\nu$-relation R and wraps every summand in a tuple's annotation in $X_{T,\nu}$. The commit annotation operator $\alpha_{C,T,\nu}(R)$ only wraps summands produced by Transaction T using operator* COM$[T,\nu](k)$ *from Definition 3.*

$$\alpha_{X,T,\nu}(R)(t) = \begin{cases} \sum_{i=0}^{n(R(t))} \text{COM}[T,\nu](R(t)[i]) & \textit{if } X = C \\ \sum_{i=0}^{n(R(t))} X_{T,\nu}(R(t)[i]) & \textit{otherwise} \end{cases}$$

5.1 Update Reenactment

We first define reenactment for an update u that is executed over the historic database seen by u's transaction T at the time of the update ($R[T,\nu(u)]$). Here we abuse notation and treat $R[T,\nu]$ as a syntactic construct that we can substitute with an algebraic expression which computes this version of R. For example, $Q(D[T,\nu])$ denotes the query Q where every access to a relation R is substituted by $R[T,\nu]$.

Definition 6. *Let H be a history over database D. The reenactment query $\mathbb{R}(u)$ for an operation u in H is:*

$$\mathbb{R}(\mathcal{U}[\theta,A,T,\nu](R)) = \alpha_{U,T,\nu+1}(\Pi_A(\sigma_\theta(R[T,\nu]))) \cup \sigma_{\neg\theta}(R[T,\nu])$$
$$\mathbb{R}(\mathcal{I}[Q,T,\nu](R)) = R[T,\nu] \cup \alpha_{I,T,\nu+1}(Q(D[T,\nu]))$$
$$\mathbb{R}(\mathcal{D}[\theta,T,\nu](R)) = \alpha_{D,T,\nu+1}(\sigma_\theta(R[T,\nu])) \cup \sigma_{\neg\theta}(R[T,\nu])$$

An update applying the expressions from A to all input tuples matching condition θ and wraps the annotation of such tuples into an update annotation. All other tuples are not modified. We can compute the result of an update as the union between these sets. Similarly, a deletion wraps tuples matching its condition in delete annotations. Thus, it can be reenacted as the union between deleted (matching condition θ) and unmodified inputs. An insert statement adds the result of a query Q to relation R. It can be reenacted as the union between relation R and the result of Q.

Example 6. *Consider the reenactment query for the first update operation $\mathbb{R}(u_1)$ of Transaction T_6. $u_1 = \mathcal{U}[cust = 'Alice' \wedge typ = 'Checking', bal - 1500 \rightarrow bal, T_6, 11](Account)$ of example Transaction T_6. The reenactment query $\mathbb{R}(u_1)$ is:*

$$\alpha_{U,T_6,11}(\Pi_{cust,typ,bal-1500 \rightarrow bal}(\sigma_{cust='Alice' \wedge typ='Checking'}(Account[T_6,11])))$$
$$\cup \sigma_{\neg(cust='Alice' \wedge typ='Checking')}(Account[T_6,11])$$

Theorem 1. *Let u be an update. Then, $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$.*

Proof. We prove the theorem by substitution of operator definitions. We show the proof for an update $u = \mathcal{U}[\theta, A, T, \nu](R)$. The proofs for inserts and deletes are analogous. The reenactment query $\mathbb{R}(u)$ for u is:

$$\alpha_{U,T,\nu+1}(\Pi_A(\sigma_\theta(R[T,\nu]))) \cup \sigma_{\neg\theta}(R[T,\nu])$$

We have to show that $u(t) = \mathbb{R}(u)(t)$ for any $t \in R$. Let $Q' = \Pi_A(\sigma_\theta(R[T,\nu]))$. Substituting $\mathcal{RA}^+$ definitions we get:

$$\mathbb{R}(u)(t) = \sum_{i=0}^{n(Q'(u))} U_{T,\nu+1}^{id(Q'(u)[i])}(Q'(u)[i]) + (R(t) \cdot \neg\theta(t))$$

Now we substitute $Q'(t) = \sum_{u:u.A=t}(R(u) \cdot \theta(u))$ and apply commutativity of $+$ to get

$$= R(t) \cdot \neg\theta(t) + \sum_{i=0}^{n(Q'(t))} U_{T,\nu+1}^{id(Q'(t)[i])}((\sum_{u:u.A=t} R(u) \cdot \theta(u))[i])$$

Using the MV-semiring equivalence $\mathcal{A}(k + k') = \mathcal{A}(k) + \mathcal{A}(k')$, we can pull out the inner sum in the second part:

$$\ldots + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u)\cdot\theta(u))} U_{T,\nu+1}^{id((R(u)\cdot\theta(u))[i])}((R(u) \cdot \theta(u))[i])$$

Note that $n(R(u) \cdot \theta(u)) = n(R(u))$ if $\theta(u) = 1$. If $\theta(u) = 0$ then $n(R(u) \cdot \theta(u)) \neq n(R(u))$, but this does not affect the result, because then $R(u)[i] \cdot \theta(u) = 0$. An analog argument holds for $id(R(u) \cdot \theta(u))$. Applying distributivity and using the MV-semiring equivalence $\mathcal{A}(k \cdot k') = \mathcal{A}(k) \cdot k'$ for $k' = 1$ or $k' = 0$ to pull out the multiplication $\theta(u)$ we get:

$$= R(t) \cdot \neg\theta(t) + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u))} U_{T,\nu+1}^{id(R(u)[i])}(R(u)[i]) \cdot \theta(u)$$
$$= \mathcal{U}[\theta, A, T, \nu](R)(t)$$
□

5.2 SI Reenactment

To reenact a transaction, we merge the reenactment queries for updates of the transaction in a way that respects the visibility rules enforced by the concurrency control protocol. Under SI, each update u_i of a transaction T sees the version of the database at transaction start plus local modifications of updates u_j from T with $j < i$. Thus, effectively, each update u_i updating the relation R is evaluated over the annotated relation produced by the most recent update u_j that updated R with $j < i$. Since we have proven that $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$, each reference to a relation $R[T,\nu]$ produced by update u_j can be replaced with $\mathbb{R}(u_j)$ (as mentioned above we treat $R[T,\nu]$ as a symbolic expression in this context). Applying this substitution recursively and adding an annotation operator to wrap the final outputs in commit annotations results in a single query $\mathbb{R}^R(T)$ per relation R affected by T. We use $R(T)$ to denote all relations targeted by at least one update of T and $\textsc{Last}(T, R, \nu)$ to denote the last update executed before ν in T that updated relation R.

Definition 7. *Let T be a transaction in a history H. The reenactment query $\mathbb{R}(T)$ for T is:*

$$\mathbb{R}(T) = \{\mathbb{R}^R(T) \mid R \in R(T)\}$$
$$\mathbb{R}^R(T) = \alpha_{C,T,End(T)}(\mathbb{R}^R(\textsc{Last}(T, R, End(T))))$$

where query $\mathbb{R}^R(u)$ is computed as follows. We initialize $\mathbb{R}^R(u) = \mathbb{R}(u)$ and then apply the following substitution rule until a fix point is reached (for every relation S accessed by T, only references of the form $S[Start(T)]$ remain):
Pick a relation mention $S[T,\nu]$ in the current $\mathbb{R}^R(u)$

- *If $\exists u' \in T : u'$ updates $S \wedge \nu(u') < \nu$ then replace $S[T,\nu]$ with $\mathbb{R}(\textsc{Last}(T, S, \nu))$*
- *Otherwise, replace $S[T,\nu]$ with $S[Start(T)]$*

Technically, $\mathbb{R}(T)$ for a transaction T is a set of queries. However, abusing terminology we refer to this set as the reenactment query of T and by $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$ mean that for every $R \in R(T)$, the reenactment query $\mathbb{R}^R(T)$ for a relation R is equivalent to the effect that transaction T has on relation R. The structure of the reenactment query for SI transactions updating a single relation R is outlined below.

$$\begin{tikzcd} \boxed{R[Start(T)]} \arrow[r] & \boxed{\mathbb{R}(u_1)} \arrow[r] & \boxed{\mathbb{R}(u_2)} \arrow[r] & \boxed{\mathbb{R}(u_3)} \arrow[r, dotted] & \boxed{\mathbb{R}(u_{n-1})} \arrow[r] & \boxed{\mathbb{R}(u_n)} \end{tikzcd}$$

Example 7. *Consider Transaction T_5 from the running example. Let us refer to its operations as u_1 and u_2. We use the following abbreviations in this example:* `Account` *= `A`, cust = c, typ = t, and bal = b. Consider the construction of the reenactment query for T_5 on `A`. The last update modifying `A` is u_2. Thus, $\mathbb{R}^A(T_5) = \alpha_{C,T,13}(\mathbb{R}^A(u_2))$. Operation u_2 updates relation `A` at version 12. The reenactment query for u_2 is:*

$$\mathbb{R}^A(u_2) = \alpha_{U,T_5,13}(\Pi_{c,t,(b+300)\to b}(\sigma_{t='Savings'\wedge b>5000}(A[T_5,12])))$$
$$\cup\, \sigma_{\neg(t='Savings'\wedge b>5000)}(A[T_5,12])$$

The last update of Transaction T_5 that modified relation `A` before version 12 is u_1. Thus, the access to $A[T_5,12]$ in $\mathbb{R}^A(u_2)$ is replaced with $\mathbb{R}^A(u_1)$. The access to relation `A` by update u_1 is not replaced in $\mathbb{R}^A(u_1)$, because there is no update operation in T_5 that updated this relation before u_1 was executed. The final reenactment query $\mathbb{R}^A(T_5)$ is:

$$\mathbb{R}^A(T_5) = \alpha_{C,T,13}(\mathbb{R}^A(u_2))$$
$$\mathbb{R}^A(u_2) = \alpha_{U,T_5,13}(\Pi_{c,t,(b+300)\to b}(\sigma_{t='Savings'\wedge b>5000}(\mathbb{R}^A(u_1))))$$
$$\cup\, \sigma_{\neg(t='Savings'\wedge b>5000)}(\mathbb{R}^A(u_1))$$
$$\mathbb{R}^A(u_1) = \alpha_{U,T_5,11}(\Pi_{c,t,(b+100)\to b}(\sigma_{t='Savings'}(A[10])))$$
$$\cup\, \sigma_{\neg(t='Savings')}(A[10])$$

Theorem 2. *Let T be a transaction. Then, $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$.*

Proof. We now prove the theorem by induction over the number of updates in transaction T. To simplify the exposition, assume WLOG that T updates a single relation R. Induction Start: For a transaction with a single update u_1, the theorem follows from equivalence for updates (Thm. 2) and the equivalence of the commit annotation operator and commit annotations produced by T (both are defined using COM). Induction Step: Assume that $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$ for transactions with up to i updates. We have to show that the same holds for any $T = u_1, \ldots, u_i, u_{i+1}, c$. Let $T_i = u_1, \ldots, u_i, c$. WLOG assume $End(T) = End(T_i)$. We know that $\mathbb{R}(T_i) \equiv_{\mathbb{N}[X]^\nu} T_i \Rightarrow R[T_i, End(T_i)] = R[T_i, \nu(u_i) + 1]$. Since T_i and T have executed the same updates over the same input, it follows that $R[T_i, \nu(u_i) + 1] = R[T, \nu(u_i) + 1]$. From the definition of $R[T,\nu]$ we know that $R[T, End(T)] = R[T, \nu(u_{i+1}) + 1] = u_{i+1}(R[T, \nu(u_{i+1})])$. Using the equivalences stated above we can deduce $u_{i+1}(R[T, \nu(u_{i+1})]) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}(\mathbb{R}^R(u_i)))$. We know that $\mathbb{R}(u_{i+1}) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}$ and, thus, it

(a) Version Merge Operator

$$\mu(R_1, R_2)(t) = \sum_{i=0}^{n(R_1(t))} R_1(t)[i] \times isMax(R_2, R_1(t)[i]) + \sum_{i=0}^{n(R_2(t))} R_2(t)[i] \times isStrictMax(R_1, R_2(t)[i])$$

(b) Check Tuple Versions of a Relation R

$$isMax(R,k) = 0 \quad \text{if } \exists t', k', j : idOf(R(t')[j]) = idOf(k) \wedge versionOf(R(t')[j]) > versionOf(k), 1 \text{ otherwise}$$
$$isStrictMax(R,k) = 0 \quad \text{if } \exists t', k', j : idOf(R(t')[j]) = idOf(k) \wedge versionOf(R(t')[j]) \geq versionOf(k), 1 \text{ otherwise}$$
$$idOf(X^{id}_{T,\nu}(k')) = id \qquad versionOf(X^{id}_{T,\nu}(k')) = \nu$$

Fig. 7: Definition of auxiliary operators used in RC-SI reenactment

follows that $R[T, End(T)] \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}^R(u_{i+1})$. Since $\mathbb{R}^R(T) = \alpha_{C,T,End(T)}(\mathbb{R}^R(u_{i+1}))$ this concludes the proof. □

5.3 RC-SI Reenactment

Based on our model for RC-SI histories that we did present in Sec. 4.2, we have to construct reenactment queries for updates of a transaction T such that $R_{ext}[T, \nu(u)]$ is the input of every update u over relation R. $R_{ext}[T, \nu(u)]$ contains tuple versions from $R[\nu]$ and also those tuples from R that have been modified by previous updates of the transaction T. Thus, we can compute it as a union between these two sets of tuple versions by filtering out invalid tuple versions.

Version Merge Operator. This operator [9] merges two version R_1 and R_2 of a relation R such that the output includes 1) each tuple version once that exists in both inputs and 2) the newer version of each tuple which exists as different versions in both inputs (shown in Fig. 7). We construct $R_{ext}[T, \nu]$ using this operator. Functions $isMax$ and $isStrictMax$ used to define μ are explained below.

Check Tuple Versions of a Relation. $isMax(R, k)$ returns 0 when relation R has a newer version of the tuple version encoded as annotation k. Function $isStrictMax$ is a strict version of $isMax$ function that also returns 0 when the tuple version k exists in R. These functions use $idOf(k)$ to access identifiers and $versionOf$ to retrieve the version identifiers from an annotation k. These functions are only defined for inputs from normalized, admissible $\mathcal{K}^\nu$-relation (see Sec. 3).

RC-SI transactions that modify multiple relations are handled analog to SI. Hence, we only present the construction of reenactment queries for RC-SI transactions that update a single relation R. The reenactment query for the Transaction $T = (u_1, \ldots, u_n, c)$ executed under RC-SI is defined recursively. It is constructed starting with a commit annotation operator applied to the reenactment query $\mathbb{R}(u_n)$ for the last update of T. Then for $i \in n-1, \ldots, 1$ we replace $R[T, \nu(u_{i+1})]$ in the query constructed so far with $\mu(\mathbb{R}(u_i), R[\nu(u_{i+1})])$. Operator μ computes $R_{ext}[T, \nu(u_{i+1})]$ which is the input seen by u_{i+1}. The structure of the reenactment query for RC-SI transactions for a single relation R is shown below.

$$\begin{tikzcd}
R[\nu(u_1)] \arrow[r] & \mathbb{R}(u_1) \arrow[r] & \mu \arrow[r] & \mathbb{R}(u_2) \arrow[r] & \mu \arrow[r] & \mathbb{R}(u_3) \arrow[r, dotted] & \mathbb{R}(u_{n-1}) \arrow[r] & \mu \arrow[r] & \mathbb{R}(u_n) \\
 & R[\nu(u_2)] \arrow[ur] & & R[\nu(u_3)] \arrow[ur] & & & R[\nu(u_n)] \arrow[ur] & &
\end{tikzcd}$$

Theorem 3. *If T is a RC-SI transaction, then $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$.*

Proof. Assume that transaction $T = u_1, \ldots, u_n, c$ is updating a single relation R. We need to show that the input $R[T, \nu(u)]$ for an update u is the same as the input produced for $\mathbb{R}(u)$ by the reenactment query for Transaction T. We prove this fact by induction over the number of updates in Transaction T. Induction Start: Let $T = u_1, c$. This case is analog to SI. Induction Step: Assume that $R[T, \nu(u_i)] = R_{ext}[T, \nu(u_i)]$ for any $i \leq n$ where i is the number of operations in Transaction T. We need to prove that for any transaction $T = u_1, \ldots, u_{n+1}, c$ we have that $R_{ext}[T, \nu(u_{n+1})]$ is equal to the input for the reenactment query $\mathbb{R}(u_{n+1})$ of u_{n+1} within the reenactment query $\mathbb{R}(T)$. In the reenactment query, the input to $\mathbb{R}(u_{n+1})$ is $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])$. Based on the induction hypothesis we have $\mathbb{R}(u_n) = R[T, \nu(u_{n+1})]$. Thus, denoting $\nu(u_{n+1})$ as ν_{n+1}:

$$\begin{aligned}
&\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t) \\
&= \sum_{i=0}^{n(R[T,\nu_{n+1}](t))} R[T, \nu_{n+1}](t)[i] \\
&\qquad \times isMax(R[\nu_{n+1}], R[T, \nu_{n+1}](t)[i]) \\
&+ \sum_{i=0}^{n(R[\nu_{n+1}](t))} R[\nu_{n+1}](t)[i] \\
&\qquad \times isStrictMax(R[T, \nu_{n+1}], R[\nu_{n+1}](t)[i])
\end{aligned}$$

$R_{ext}[T, \nu_{n+1}](t)$ is also defined as a sum over the elements from $R[T, \nu_{n+1}](t)$ and $R[\nu_{n+1}](t)$. Individual summands are filtered out using VALIDIN and VALIDEX. Thus, to prove that $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})]) = R_{ext}[T, \nu(u_{n+1})]$, we have to show that if the $isMax$ or $isStrictMax$ function returns 1 on a summand then the same is true for VALIDIN or VALIDEX, respectively. Fixing a tuple t and a tuple version (summand) k with tuple identifier id in its annotation, we have to distinguish between five cases based on whether such a tuple version occurs in $R[\nu(u_{n+1})]$ and/or in $R[T, \nu(u_{n+1})]$, and, if it occurs in both, whether one of these versions is newer. We show the proof for one of these cases. The remaining cases are similar in nature (see [21]). Case 1: For this case, we assume that the first tuple version with identifier $idOf(k)$ was created by an insert of Transaction T before ν_{n+1} and, thus k is only present in $R[T, \nu_{n+1}](t)$. Therefore, function $isMax(\ R[\nu_{n+1}], k)$ returns 1 and k is in $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Similarly, since k is the latest version, we have that VALIDIN($R[T, \nu_{n+1}], t, k, \nu_{n+1})$ returns 1 because k's outmost version annotation is from T. Thus, k is also present in $R_{ext}[T, \nu_{n+1}]$. Having proven that $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})]) = R_{ext}[T, \nu(u_{n+1})]$ it follows that $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$. □

6 Implementation and Optimizations

We have implemented provenance capture for transactions in our **GProM** (**G**eneric **Pro**venance **M**iddleware) system.

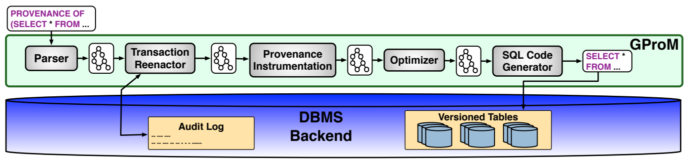


Fig. 8: GProM architecture

Fig. 8 shows how the system processes a transaction provenance request. GProM translates SQL statements with provenance requests into a relational algebra with annotated semantics. Transaction provenance requests are processed by the *reenactor* module that constructs the reenactment query for a transaction using the audit log of the backend DBMS to determine which statements were executed by the transaction. The *provenance instrumentation* module rewrites the reenactment query with annotated semantics into a relational algebra expression that produces our relational encoding of MV-semiring annotations. This query uses time travel to access past database states. We return provenance restricted to a transaction T using the method discussed at the end of Sec. 4. Afterwards, we optimize the algebra expression and then compile it into SQL code. We present optimizations specific to reenactment in the following and refer the reader to [22] for a detailed discussion of GProM's heuristic and cost-based optimization framework.

6.1 Reducing MV to Standard Relational Semantics

We encode a normalized MV-semiring annotation of a tuple as a set of tuples - one for each summand. We add provenance attributes to the result schema to store tuples in the provenance (variables in MV-semiring expressions) and version annotations for each summand. For instance, a tuple t annotated with $x + y$ would be encoded as two tuples encoding the summand x and y, respectively. When computing the provenance of a Transaction T, initial annotations for a relation R are created to represent variables in annotations. We access the snapshot of R as of the start of Transaction T and create annotations by duplicating attribute values using projection. Note that the relational encoding of annotations produced by this step corresponds to the result of applying the filtering step 2 in Sec. 4.3 to $R[Start(T)]$. Here we assume a standard SI based implementation of time travel that allows us to access a *snapshot* R_ν of relation R containing all committed tuple versions valid at ν. Furthermore, we expect a snapshot to store the following information for each tuple version: 1) the transaction that created the tuple version (attribute Xid) and 2) a unique tuple identifier (attribute Id). We instrument the remaining operators to propagate annotations from their inputs. Consider a transaction $T = (u_1, \ldots, u_n, c)$. We apply a selection $\mathcal{U}_1 \vee \ldots \vee \mathcal{U}_n$ to the result to implement filtering step 1 that removes tuples that were not affected by Transaction T (see Sec. 4.3). The details of our encoding and instrumentation are presented in our technical report [20].

Reenacting With CASE. Our reenactment approach translates an `UPDATE` into a union between two accesses of the input relation. For a sequence of updates in a transaction this leads to queries where both inputs of such a union are again unions. Unless intermediate results are reused, this leads to an exponential number of unions (in the number of updates). Instead of computing the union between the set of updated tuples and non-updated tuples, we can use the SQL `CASE` construct to decide for each tuple whether it should be updated. We can reenact an update $\mathcal{U}[\theta, A, T, \nu](R)$ using a projection constructed as follows. We replace each expression $e \rightarrow a$ in A with `CASE WHEN` θ `THEN` e `ELSE` a `END AS` a. Version annotation attributes ($\mathcal{U}_i$) are computed in a similar fashion. This approach is also applicable for deletes.

Example 8. *Consider a Transaction T with a single update:*

```
UPDATE Account SET bal = bal + 100 WHERE typ = 'Savings';
```

Reenactment produces the following query (for simplicity we omit instrumentation for propagating annotations). SQL construct `R AS OF t` *denotes the use of time travel to compute snapshot R_t. Using* `CASE` *instead of union we get:*

```
SELECT cust, type, (CASE  WHEN (typ = 'Savings')
                          THEN bal + 100 ELSE bal END) AS bal
FROM Account AS OF Start(T);
```

6.2 Prefiltering Provenance

Recall that we apply a selection on $\mathcal{U}_1 \vee \ldots \vee \mathcal{U}_n$ to the result of reenactment to filter out tuples that were not affected by any update of the transaction. Thus, the reenactment query is evaluated over all tuples from $R_{Start(T)}$. We now discuss two optimizations that filter out tuples early on.

Prefiltering With Update Conditions. The naive method can be improved if we can determine upfront which tuples will be affected by a transaction. Consider a transaction $T = u_1, \ldots, u_n, c$ where each u_i is an `UPDATE` and a tuple t valid at transaction start. Tuple t was modified by a subset (potentially empty) of the updates of T. If t is affected, then there has to exist a first update u_t in T that modified tuple t. Thus, t has to fulfill the condition of u_t. This observation can be used to characterize the set of tuples affected by the transaction. In particular, this is the set fulfilling the condition $\theta_1 \vee \ldots \vee \theta_n$ where θ_i is the condition of the ith update operation. Hence, it is safe to apply a selection on this condition to the input of reenactment. This approach is not applicable to a relation R if one of the transaction's inserts uses a query that accesses relation R. Delete operations can be handled like update operations whereas inserts create new tuples and there is no need for prefiltering.

Join With Committed Tuple Versions. The version of the database at commit of transaction T contains all tuple versions that were created by T. Recall that snapshots use a column Xid to store the updating transaction. Thus, we can determine which tuple versions were created by a transaction T by running a query $\sigma_{Xid=T}(R_{End(T)+1})$. To retrieve the version of these tuples valid at transaction start, we can join the result of this query with $R_{Start(T)}$. Here, we assume that the database uses unique immutable tuple identifiers stored in attribute Id. We join on this identifier, i.e., in the reenactment query we replace $R_{Start(T)}$ with $R_{Start(T)} \bowtie \Pi_{Id}(\sigma_{Xid=T}(R_{End(T)+1}))$. This approach is only applicable to relations that are not accessed by any insert's query in the transaction.

7 Experiments

Our experimental evaluation studies 1) the performance of provenance capture and 2) the overhead for transaction execution comparing our approach (using reenactment, audit

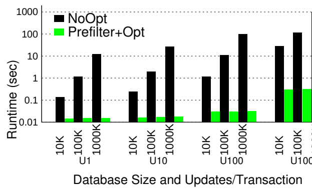


Fig. 9: Relation size

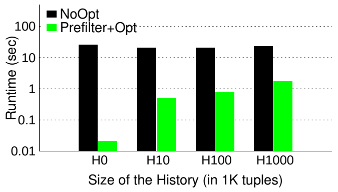


Fig. 10: History size

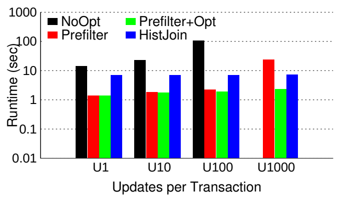


Fig. 11: Optimization methods

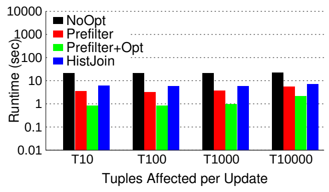


Fig. 12: Affected tuples

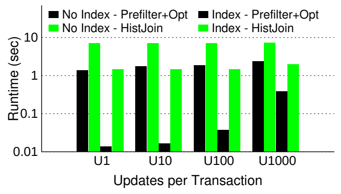


Fig. 13: Index vs. no index

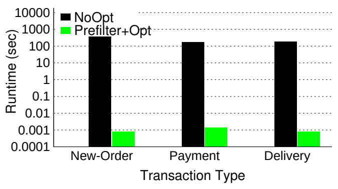


Fig. 14: Provenance for TPC-C

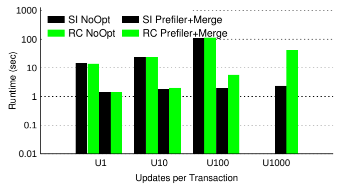


Fig. 15: Isolation Levels

logging and history maintenance) with an approach that directly stores provenance. All experiments are run with DBMS X as a backend (name omitted due to licensing restrictions). We use a synthetic workload to evaluate how our approach scales in various parameters and a TPC-C workload to test its performance for realistic transactions. All experiments were executed on a machine with 2 x AMD Opteron 4238 CPUs (12 cores in total), 128 GB RAM, and 4 x 1TB 7.2K HDs in a hardware RAID 5 configuration.

7.1 Setup and Workload

Datasets and Workload. We use a relation with five uniformly distributed, numeric columns. We created variants $R10K$, $R100K$, and $R1000K$ with 10K, 100K, and 1M tuples and no significant history ($H0$). Additionally, we generated three variants of $R1000K$ with different history sizes $H10$, $H100$, and $H1000$ (100K, 1M, and 10M tuples history). At first, we only consider transactions that consist solely of update statements. We vary the following parameters: U is the number of updates per transaction, e.g., $U10$ is a transaction with 10 updates. T is the number of tuples affected by each update. Unless stated otherwise, we use $T1$. The tuple to be updated is selected randomly using the primary key (uniform distribution). The default isolation level used in the experiments is `SERIALIZABLE` (SI).
Compared Methods. We compare different configurations for capturing provenance for a transaction - each using a subset of the optimizations described in Sec. 6. Experiments were repeated 100 times and we report the average runtime. **NoOpt (N)**: Computes the provenance of all tuples in a relation, even tuples that were not affected by the transaction, i.e., we do not apply the filter condition on the version annotation attributes. **Prefilter (P)**: Only returns provenance of tuples affected by the transaction using a selection on the disjunction of the conditions of the transaction's updates (see Sec. 6.2). The database system was instructed to materialize the intermediate result corresponding to each update in the reenactment query using temporary relations. **Prefilter+Opt (PO)**: This is the same as *Prefilter*, but we merge operators (particularly, projections) to reduce the number of query blocks. **HistJoin (HJ)**: We use a join to compute partial provenance as described in Sec. 6.2. This configuration merges operators where possible. The maximum allocated execution time for each method is 1,000 sec.

7.2 Performance of Provenance Capture

In this set of experiments we execute the transactional workload beforehand and measure the performance of capturing provenance for transactions from this workload. We study how our reenactment approach scales in database and history size as well as complexity of the transaction (number of operations, amount of modified tuples, types of updates).
Relation Size and Updates/Transaction. We compute the provenance of transactions varying the number of updates per transaction ($U1$ up to $U1000$) and the size of the database ($R10K$, $R100K$, and $R1000K$) without significant history ($H0$). Fig. 9 shows the runtime of capturing provenance for one transaction. We scale linearly in R and U. By reducing the amount of data to be processed by the reenactment query and by merging operators, the *PO* approach is up to three orders of magnitude faster than the naive *N* configuration.
History Size. We capture provenance for transactions with 10 updates ($U10$) over relations with 1M tuples ($R1000K$) and history sizes: $H0$, $H10$, $H100$, and $H1000$. As shown in Fig. 10, *N* exhibits almost constant performance. The runtime is dominated by evaluating the reenactment query over 1M tuples (all tuples in one version of the relation) hiding the impact of scanning the history. Since we have not created any indexes on the history relations, the *PO* approach only has the advantage of processing less tuples in the provenance computation, but still has to scan most of the history to find tuples that were updated.
Comparing Optimization Techniques. Fig. 11 shows results for varying the number of updates ($U1$ to $U1000$) using *R1000K-H1000*. Compared with *P*, *PO* benefits from avoiding materialization. This optimization is more effective for larger transactions, as reenactment queries for such transactions are increasingly complex. While resulting in ~20% improvement for U100, it improves the runtime by a factor of roughly 10 for U1000. The cost of *PO* is affected by the first selection that is applied to 1M tuples (no index on the history relation). The size of this condition is linear in the number of update operations. The runtime of *HJ* is almost not affected by parameter U, because it is dominated by

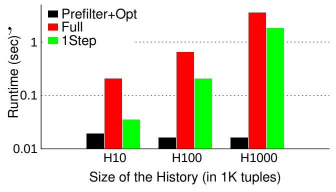


Fig. 16: Provenance Retrieval

W/O	History	History+Audit
27.46 sec	30.94 sec	32.59 sec

Fig. 17: Runtime Overhead

Total Runtime (sec) + Relative Overhead (rel)

Method	sec	rel
Reenact	**32.59**	**19%**
1Step	67.18	145%
Full H10	64.02	133%
Full H100	71.98	162%
Full H1000	220.16	702%

Storage Size (MB)

#Tuples / Update	Method	H10	H100	H1000
T1	History	41	97	655
	Audit Log	36	360	3600
	Total	77	457	4255
	Full	62	**181**	**1245**
	1Step	**45**	191	1658
T10	Audit Log	4	36	360
	Total	45	**133**	**1015**
T100	Audit Log	0.3	4	36
	Total	**41.3**	**98**	**691**

Fig. 18: Eager vs. reenactment

the join between historic relations. *PO* outperforms *HJ* by a factor of about 3. For $U1000$, the *N* method did not finish within the allocated time slot (1000 sec.).

Affected Tuples Per Update. Fig. 12 shows results for $U10$ where each update modifies 10, 100, 1000, or 10000 tuples of the *R1000K-H1000* relation. As evident from Fig. 12, the runtime is not significantly affected when increasing the number of affected tuples per update. It is dominated by scanning the history and filtering out updated tuples (*PO*) or the self-join between historic relations (*HJ*). Increasing the T parameter by 3 orders of magnitude results in a runtime increase of about 150% (*PO*) and 20% (*HJ*).

Index vs. No Index. Fig. 13 shows the effect of replicating the indexes defined for the *R1000K-H1000* relation to its corresponding history relation. We vary U ($U1$ to $U1000$). We omit the *N* (no benefit from indexes) and *P* (consistently outperformed by *PO*) configurations. Using indexes improves execution time of queries that apply *PO* considerably.

TPC-C. We capture provenance for the TPC-C benchmark. We execute a TPC-C workload over an instance with 32 warehouses. The resulting database is roughly 16GB large. The benchmark defines 5 transaction types, out of which 2 are read-only. We compare the *N* and *PO* methods for the 3 transaction types that execute updates. Fig. 14 shows the result for computing the provenance of a single transaction of each type. Each of these transactions only modifies a few tuples. Thus, the cost for *PO* is quite low. The cost for *N* is dominated by scanning the full input relation.

Isolation Levels. Fig. 15 compares the performance of capturing provenance for *SI* and *RC-SI*. We use *R1000K-H1000* and vary the number of updates per transaction (*U1* to *U1000*). As expected, *SI* reenactment is more efficient than *RC-SI* reenactment, because for *RC-SI* we have to check for each tuple and update whether the tuple is visible to the update. The impact is more noticeable for efficient configurations such as *P* and larger number of updates (*U1000*). The runtime of *N* is dominated by a full scan of the large input and history tables and by having to produce 1M output rows. For *U1000*, the method *N* did not finish within the allocated time slot (1000 sec.).

7.3 Overhead and Eager Provenance Capture

We use audit logging and time travel to reconstruct provenance of past transactions. We now quantify the runtime and storage overhead of DBMS X's built-in temporal and audit features. We measure the execution time of 10,000 transactions with $U10$ and $T1$ run over the $R1000$ instance. Fig. 17 shows the total runtime for three configurations: without temporal and audit logging features (W/O), with temporal features, and with both the temporal and audit logging features. If history maintenance is activated then this results in about 12% runtime overhead (see Fig. 17). This result agrees with DBMS X's documentation which states 5% overhead for mixed read-write workloads. Also activating audit logging results in a total overhead of $\sim$ 19%.

We now compare our approach with eager provenance capture during transactions execution. We consider two configurations: **1Step** stores a separate provenance record for each tuple version and statement in an extra relation. Each record is linked to the provenance record for the previous tuple version. The provenance of a transaction is reconstructed by recursively joining these provenance records; **Full** stores the complete derivation history of each tuple in an additional column. Results are shown in Fig. 18.

Transaction Execution Overhead. Using the workload from Sec. 7.3, we compare the overhead for transaction execution incurred by these two eager methods with our method. The performance of our method and *1Step* remains stable when increasing the size of the history. In contrast, the overhead of *Full* increases with the history size, as the size of provenance per tuple increases and the attribute storing provenance has to be updated by every operation. Both *1Step* and *Full* do significantly slow down the transaction processing showing up to a factor of 7 higher overhead than our approach.

Storage Size. We compare the storage size used by the three methods for a table with $1M$ rows varying the size of the history ($H10$, $H100$, and $H1000$) and number of tuples affected by each update ($T1$, $T10$, and $T100$). For our method we show the total storage space as well as the breakdown into a relation plus history and the audit log. Only the size of the audit log is affected by the T parameter. Thus, we only show our method for $T10$ and $T100$ since the other methods require the same storage for all T values. The results shown in Fig. 18 demonstrate that in the worst case (1 tuple affected per update), our method requires up to $\sim$4 times more storage than the best approach. This overhead is caused by the audit log storing one SQL statement per modified tuple. However, if more tuples are affected by each statement then our method requires about the same or less space than the alternatives.

Retrieving Provenance. We now compare the performance of using reenactment (the *PO* method) for retrieving provenance with *1Step* and *Full*. Fig. 16 shows the result for capturing provenance of transactions with $U10$ and $T1$ varying the history size (H). We created relevant indexes for each method. Optimized reenactment outperforms both alternatives, because *Full* requires filtering tuples based on the transaction identifier that is stored in the provenance column and *1Step* requires a recursive query or multi-way join to reconstruct the provenance of a transaction from provenance records for each update.

8 Conclusions

We present the first solution for capturing provenance for transactions run under SI and RC-SI. Our approach is based on *reenactment*, i.e., replaying updates and transactions as queries with annotated semantics. Using audit logging, time travel, and a relational encoding of reenactment, we retroactively capture the provenance of tuples produced by transactional histories using a standard DBMS. In future work, we will study reenactment for more expressive query languages (e.g., aggregation [16]). Reenactment has many potential applications such as answering historic What-If queries (e.g., "What would have happened if we had updated accounts using 10% interest?").

Acknowledgments

This work was supported in part by NSF Award #1640864.

References

[1] 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*, 2013, pp. 291–320.

[2] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "An Annotation Management System for Relational Databases," *VLDB Journal*, vol. 14, no. 4, pp. 373–396, 2005.

[3] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen, "Collaborative data sharing via update exchange and provenance," *TODS*, vol. 38, no. 3, p. 19, 2013.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[5] S. Vansummeren and J. Cheney, "Recording Provenance for SQL Queries and Updates," *Data Eng. Bull*, vol. 30, no. 4, pp. 29–37, 2007.

[6] P. Buneman, J. Cheney, and S. Vansummeren, "On the Expressiveness of Implicit Provenance in Query and Update Languages," *TODS*, vol. 33, no. 4, pp. 1–47, 2008.

[7] X. Niu, B. S. Arab, S. Lee, S. Feng, X. Zou, D. Gawlick, V. Krishnaswamy, Z. H. Liu, and B. Glavic, "Debugging transactions and tracking their provenance with reenactment," *PVLDB*, vol. 10, no. 12, pp. 1857–1860, 2017.

[8] B. S. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic, "A generic provenance middleware for database queries, updates, and transactions," in *TaPP*, 2014.

[9] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic, "Reenactment for read-committed snapshot isolation," in *CIKM*, 2016, pp. 841–850.

[10] P. Buneman, S. Khanna, and W.-C. Tan, "Why and Where: A Characterization of Data Provenance," in *ICDT*, 2001, pp. 316–330.

[11] Y. Cui, J. Widom, and J. L. Wiener, "Tracing the Lineage of View Data in a Warehousing Environment," *TODS*, vol. 25, no. 2, pp. 179–227, 2000.

[12] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance Semirings," in *PODS*, 2007, pp. 31–40.

[13] E. V. Kostylev and P. Buneman, "Combining dependent annotations for relational algebra," in *ICDT*, 2012, pp. 196–207.

[14] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen, "On provenance minimization," in *PODS*, 2011, pp. 141–152.

[15] D. Olteanu and J. Závodný, "On factorisation of provenance polynomials," in *TaPP*, 2011.

[16] Y. Amsterdamer, D. Deutch, and V. Tannen, "Provenance for Aggregate Queries," in *PODS*, 2011, pp. 153–164.

[17] F. Geerts and A. Poggi, "On database query languages for K-relations," *Journal of Applied Logic*, vol. 8, no. 2, pp. 173–185, 2010.

[18] Y. Amsterdamer, S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting Lipstick on Pig: Enabling Database-style Workflow Provenance," *PVLDB*, vol. 5, no. 4, pp. 346–357, 2011.

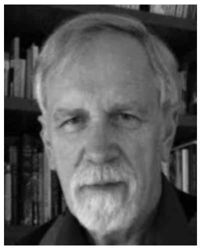
[19] T. J. Green, M. Aref, and G. Karvounarakis, "Logicblox, platform and language: A tutorial," in *Datalog in Academia and Industry*. Springer, 2012, pp. 1–8.

[20] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic, "Formal foundations of reenactment and transaction provenance," IIT, Tech. Rep., 2016. [Online]. Available: http://cs.iit.edu/%7edbgroup/pdfpubls/AG16.pdf

[21] ——, "Reenactment for read-committed snapshot isolation (long version)," *CoRR*, vol. abs/1608.08258, 2016. [Online]. Available: http://arxiv.org/abs/1608.08258

[22] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, and V. Radhakrishnan, "Provenance-aware query optimization," in *ICDE*, 2017, pp. 473–484.

Bahareh Sadat Arab received the master in computer science from University Putra Malaysia. She is now a PhD student at the Illinois Institute of Technologycollaborating with Oracle on topics related to provenance.



Dieter Gawlick is an architect at Oracle. He has developed key concepts for high-end OLTP, storage management, messaging, workflow, and information dissemination.

Vasudha Krishnaswamy received the PhD in computer science from the University of California, Santa Barbara working on semantics based concurrency control. Currently, she is a Consulting Member of Technical Staff at Oracle.

Venkatesh Radhakrishnan received the PhD in computer science from SUNY Albany. Currently, he is a Staff Software Engineer at LinkedIn, working on Pinot, a realtime distributed OLAP datastore.

Boris Glavic received the PhD in computer science from the University of Zurich. Currently, he is an Assistant Professor at the Illinois Institute of Technology focusing on data provenance and integration.