Reenactment for Read-Committed Snapshot Isolation

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

★ Illinois Institute of Technology
{barab@hawk.,bglavic@}iit.edu

♦ Oracle
{dieter.gawlick,
vasudha.krishnaswamy}@oracle.com

□ LinkedIn
vradhakrishnan@linkedin.com

ABSTRACT

Provenance for transactional updates is critical for many applications such as auditing and debugging of transactions. Recently, we have introduced *MV-semirings*, an extension of the semiring provenance model that supports updates and transactions. Furthermore, we have proposed *reenactment*, a declarative form of replay with provenance capture, as an efficient and non-invasive method for computing this type of provenance. However, this approach is limited to the snapshot isolation (SI) concurrency control protocol while many real world applications apply the read committed version of snapshot isolation (RC-SI) to improve performance at the cost of consistency. We present non-trivial extensions of the model and reenactment approach to be able to compute provenance of RC-SI transactions efficiently. In addition, we develop techniques for applying reenactment across multiple RC-SI transactions. Our experiments demonstrate that our implementation in the GProM system supports efficient re-construction and querying of provenance.

1. INTRODUCTION

Tracking the derivation of data through a history of transactional updates, i.e., tracking the provenance of such operations, is critical for many applications including auditing, data integration, probabilistic databases, and post-mortem debugging of transactions. For example, by exposing data dependencies, provenance provides proof of how data was derived, by which operations, and at what time. Until recently, no solution did exist for tracking the provenance of updates run as part of concurrent transactions.

MV-semirings and Reenactment. In previous work [5], we have introduced **MV-semirings** (multi-version semirings). MV-semirings extend the semiring provenance framework [18] with support for transactional updates. We have introduced a low-overhead implementation of this model in our GProM system [7] using a novel declarative replay technique (**reenactment** [5]). This finally makes this type of provenance available to applications using the *snapshot isolation* (**SI**) concurrency control protocol. Figure 1 illustrates how reenactment is applied to retroactively compute provenance for updates and transactions based on replay with provenance capture. Consider the database states induced by a history of concurrently executed transactions. With our approach, a user can request the provenance of any transaction executed in the past, e.g., Transaction T_2 in the example. Using reenactment, a temporal query is generated that simulates the transaction's operations within the context of the transactional history and this query is instrumented for provenance capture. This so-called reenactment query is guaranteed to return the same results (updated versions of the relations modified by the transaction) as the original transaction. In the result of the reenactment query, each tuple is annotated with its complete derivation history: 1) from which previous tuple versions was it derived and 2) which updates of the transaction affected it. Importantly, reenactment only requires an audit log (a log of SQL commands executed in the past) and time travel (query access to the transaction time history of tables) to function. That is, no modifications to the underlying database system or transactional workload are required. Many DBMS including Oracle [1], DB2, and MSSQL [21] support a query-able audit log and time travel. If a system does not natively support this functionality we can implement it using extensibility mechanisms (e.g., triggers). Snapshot isolation is a widely applied protocol (e.g., supported by Oracle, PostgreSQL, MSSQL, and many others). However, the practical applicability of reenactment is limited by the fact that many real world applications use statement-level snapshots instead of transaction-level snapshots. Using statement-level snapshots improves performance and timeliness of data even though this comes at the cost of reduced consistency. In this work, we present the non-trivial extensions that are necessary to support statement-level snapshot isolation (isolation level `READ COMMITTED` in the aforementioned systems).

Snapshot Isolation (SI). Snapshot isolation [9] is a widely applied multi-versioning concurrency control protocol. Under SI each transaction T sees a private snapshot of the database containing changes of transactions that have committed before T started and T's own changes. SI disallows concurrent transactions to update the same data item. This is typically implemented by using write locks where transactions waiting for a lock have to abort if the transaction currently holding the lock commits.

Read Committed Snapshot Isolation (RC-SI). Under RC-SI each statement of a transaction sees changes of transactions that committed before the statement was exe-

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA

 ISBN 978-1-4503-4073-1/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2983323.2983825

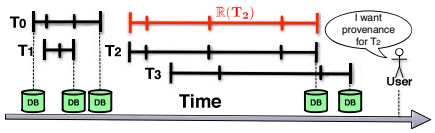


Figure 1: Reenactment

Employee

	ID	Name	Position	
$C^1_{T_0,6}(I^1_{T_0,2}(x_1))$	101	Mark Smith	Software Engineer	e_1
$C^2_{T_0,6}(I^2_{T_0,3}(x_2))$	102	Susan Sommers	Software Architect	e_2
$C^3_{T_0,6}(I^3_{T_0,4}(x_3))$	103	David Spears	Test Assurance	e_3

Bonus

	ID	EmpID	Amount	
$C^4_{T_1,10}(I^4_{T_1,8}(x_4))$	1	101	1000	b_1
$C^5_{T_2,14}(I^5_{T_2,12}(x_5))$	2	102	2000	b_2
$C^6_{T_4,18}(I^6_{T_4,16}(x_6))$	3	103	500	b_3

Figure 2: Running example database instance

Employee

	ID	Name	Position	
$C^1_{T_7,26}(U^1_{T_7,21}(C^1_{T_0,6}(I^1_{T_0,2}(x_1))))$	101	Mark Smith	Software Architecture	e_1'

Bonus

	ID	EmpID	Amount	
$C^4_{T_7,26}(U^4_{T_7,22}(C^4_{T_1,10}(I^4_{T_1,8}(x_4))))$	1	101	2000	b_1'
$C^7_{T_8,24}(I^7_{T_8,23}(C^1_{T_0,6}(I^1_{T_0,2}(x_1))))$	4	101	500	b_4

Figure 3: New and modified tuples after execution of the example history (version 26). Modified attribute values and new tuples are shown with shaded background.

	Bonus			Bonus Provenance			u_1	u_2
	ID	EmpID	Amount	P(B,ID)	P(B,EmpID)	P(B,Amount)	$\mathcal{U}_1$	$\mathcal{U}_2$
$C^4_{T_7,26}(U^4_{T_7,22}(b_1))$	1	101	2000	1	101	1000	F	T

Figure 4: Relational encoding of provenance restricted to Transaction T_7. Only tuples modified by this transaction are included and the derivation history of tuples is limited to updates of this transaction. Variables are encoded as actual tuples.

T	SQL	Time
T_7	`UPDATE Employee SET Position='Software␣Architect' WHERE ID=101;`	20
T_7	`UPDATE Bonus SET Amount = Amount + 1000 WHERE ID=101;`	21
T_8	`INSERT INTO Bonus (EmpID, Amount) (SELECT ID, 500 FROM Employee WHERE Position='Software␣Engineer');`	22
T_8	`COMMIT;`	23
T_7	`SELECT Amount INTO amounts FROM Bonus WHERE ID=101;`	24
T_7	`COMMIT;`	25

Figure 5: Example Transactional History

cuted. In this paper we assume the RC-SI semantic as implemented by Oracle, i.e., a statement waiting for a write-lock is restarted once the transaction holding the lock commits. This guarantees that each statement sees a consistent snapshot of the database.

Example 1. *Consider the example database shown in Figure 2 storing information about employees and the bonuses they received. Ignore the annotations to the left of each tuple for now. Two transactions have been executed concurrently (Figure 5) using the RC-SI protocol. In this example, all software engineers got a bonus of $1000 while software architects received $2000. Suppose administrator Bob executed transaction T_7 to update the position of Mark Smith to reflect his recent promotion to architect and update his bonus accordingly (increasing it by $1000). Concurrently, user Alice executed Transaction T_8 to implement the company's new policy of giving an additional bonus of $500 to all* software engineers. *All new and updated tuples for relations* `Employee` *and* `Bonus` *after the execution of these transactions are shown in Figure 3 (updated attributes are marked in red). Bob has executed a query at the end of his transaction (T_7) to double check the bonus amount for Mark expecting a single bonus of $2000 instead of the actual result (a second bonus of $500). The unexpected second bonus is produced by Transaction T_8, because this transaction did not see the uncommitted change of T_7 reflecting Mark's promotion. Thus, Mark was considered to still be a software engineer and received the corresponding $500 bonus. This kind of error is hard to debug, because it only materializes if the execution of the two transactions is interleaved in a certain way and would not occur in any serializable schedule.*

By exposing data dependencies among tuple versions (e.g., the $500 bonus for Mark is based on the previous version of Mark's tuple e_1 in the `Employee` table) and by recording which operations created a tuple version (e.g., the updated $2000 bonus for Mark was produced by the second update of T_7), the MV-semiring provenance model greatly simplifies debugging of transactions. We now give an overview of our model and then present our extensions for RC-SI.

2. THE MV-SEMIRING MODEL

Using MV-semirings (multi-version semirings), provenance is represented as annotations on tuples, i.e., each tuple is annotated with its derivation history (provenance).

$\mathcal{K}$-relations. We briefly review the semiring provenance framework [17, 18] on which MV-semirings are based on. In this framework relations are annotated with elements from an annotation domain K. Depending on the domain K, the annotations can serve different purposes. For instance, natural number annotations ($\mathbb{N}$) represent the multiplicity of tuples under bag semantics while using polynomials over a set of variables (e.g., x_1, x_2, ...) representing tuple identifiers the annotations encodes provenance. Let $\mathcal{K} = (K, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ 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 Figure 6. 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 $\times_\mathcal{K}$ operations where addition represents alternative use of inputs (e.g., union) and multiplication denotes conjunctive use of inputs (e.g., join). The semiring $\mathbb{N}$, the set of natural numbers with standard arithmetics corresponds to bag semantics. For example, if a tuple t occurs twice in a relation R, then this tuple would be annotated with 2 in the $\mathbb{N}$-relation corresponding to R.

Provenance polynomials. Provenance polynomials (semiring $\mathbb{N}[X]$), polynomials over a set of variables X which represent tuples in the database, model a very 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 the results of queries are annotated with polynomials

Laws of commutative semirings

$$k + 0_{\mathcal{K}} = k \qquad k \times 1_{\mathcal{K}} = k \qquad \text{(neutral elements)}$$
$$k + k' = k' + k \qquad k \times k' = k' \times k \qquad \text{(commutativity)}$$
$$k + (k' + k'') = (k + k') + k'' \qquad \text{(associcitivity)}$$
$$k \times (k' \times k'') = (k \times k') \times k''$$
$$k \times 0_{\mathcal{K}} = 0_{\mathcal{K}} \qquad \text{(annihilation through 0)}$$
$$k \times (k' + k'') = (k \times k') + (k \times k'') \qquad \text{(distributivity)}$$

Evaluation of expressions with operands from K

$$k + k' = k +_{\mathcal{K}} k' \qquad k \times k' = k \times_{\mathcal{K}} k' \qquad \text{(if } k \in K \wedge k' \in K\text{)}$$

Equivalences involving version annotations

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

Figure 6: Equivalence relations for $\mathcal{K}^{\nu}$

over these variables. For example, if a tuple was derived by joining input tuples identified by x_1 and x_2, then it would be annotated with $x_1 \times x_2$. Since we are mainly concerned with provenance, we mostly limit the discussion to $\mathbb{N}[X]$ and its MV-semiring extension as explained below.

MV-semirings. In [5] we have introduced MV-semirings which 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. Figures 2 and 3 show examples of $\mathbb{N}[X]^{\nu}$ annotations on the left of tuples. In these symbolic $\mathbb{N}[X]^{\nu}$ expressions variables (e.g., x_1, x_2, ...) represent identifiers of freshly inserted tuples and uninterpreted function symbols called *version annotations* encode which operations (e.g., a relational update) were applied to the tuple. The nesting of version annotations records the sequence of operations that were applied to create a tuple version. For instance, consider the annotation of tuple e_1 in Figure 2. This tuple was inserted at time 2 by Transaction T_0 and was assigned an identifier 1 ($I^1_{T_0,2}$). The tuple became visible to other transactions after T_0's commit ($C^1_{T_0,6}$). Observe that these annotations encode what operations have been applied to tuples and from which other tuples they were derived.

Version Annotations. A version annotation $X^{id}_{T,\nu}(k)$ denotes that an operation of type X (one of update U, insert I, delete D, or commit C) that was executed at time $\nu-1$ by transaction T did 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 $\mathbb{T}$, we use $\mathbb{A}$ to denote the set of all possible version annotations. This set contains the following version annotations for each $id \in \mathbb{I}$, $\nu \in \mathbb{V}$, and $T \in \mathbb{T}$:

$$I^{id}_{T,\nu}, U^{id}_{T,\nu}, D^{id}_{T,\nu}, C^{id}_{T,\nu} \tag{1}$$

MV-semiring Annotation Domain. In the running example, the derivation history of each tuple is a linear sequence of operations applied to a single previous tuple version. However, in the general case a tuple can depend on multiple input tuples, e.g., a query that projects an input relation onto a non-unique column ($\Pi_{Position}(Employee)$) or an update that modifies two tuples that are distinct in the input to be the same in the output (e.g., `UPDATE Employee SET ID = 101, Name = Peter`). In MV-semiring annotations this is expressed by combining the variables representing input tuples using operations $+$ and $\times$ in the expressions. Fixing a semiring $\mathcal{K}$, the domain of $\mathcal{K}^{\nu}$ is the set of finite symbolic expressions P defined by the grammar shown below where $k \in K$ and $\mathcal{A} \in \mathbb{A}$.

$$P := k \mid P + P \mid P \times P \mid \mathcal{A}(P) \tag{2}$$

For example, consider a query $\Pi_{Position}(Employee)$ evaluated over the instance from Figure 3. The result tuple `(Software Architect)` is derived from ${e_1}'$ or, alternatively, from e_2 (the two tuples with this value in attribute `position`) and, thus, would be annotated with

$$Employee({e_1}') + Employee(e_2)$$
$$= C^1_{T_7,26}(U^1_{T_7,21}(C^1_{T_0,6}(I^1_{T_0,2}(x_1)))) + C^2_{T_0,6}(I^2_{T_0,3}(x_2))$$

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 using K^{ν}, the set of congruence classes (denoted by $[]_{\sim}$) for expressions in P based on the equivalence relations as shown in Figure 6.

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

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

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

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

The definition of addition and multiplication has to be read as: create a symbolic expression by connecting the inputs with $+$ or $\times$ and then output the congruence class for this expression. For example, $k = U^1_{T,\nu}(10+5)$ is a valid element of $\mathbb{N}^{\nu}$, the bag semantics MV-semiring, which denotes that a tuple with identifier 1 was produced by an update (U) of transaction T at version ν. This element k is in the same equivalence class as $U^1_{T,\nu}(15)$ based on the equivalence that enables evaluation of addition over elements from $\mathcal{K}$.

Normal Form and Admissible Instances. We have shown in [5] that $\mathcal{K}^{\nu}$ 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. Intuitively, each summand corresponds to a tuple under bag semantics. Thus, we will sometimes refer to a summand as a tuple version in the following. Assuming an arbitrary, but fixed, order over such summands we can address elements in such a sum by position. Following [5] we use $n(k)$ to denote the number of summands in a normalized annotation k and $k[i]$ to refer to the i$^{\text{th}}$ element in the sum according to the assumed order. In the definition of updates we will make use of this normal form. Note that not all expressions produced by the grammar in Equation (2) can be produced by transactional histories. For instance, $U^1_{T,3}(C^1_{T,2}(\ldots)$ can never be produced by any history, because it would imply that an update of transaction T was applied after the transaction committed. An **admissible** $\mathcal{K}^{\nu}$ database instance is defined as an instance that is the result of applying a transactional history (to be defined later in this section) to an empty input database.

Example 2. *Consider the $\mathbb{N}[X]^{\nu}$-relation `Bonus` from the example shown in Figure 3. The first tuple ${b_1}'$ is annotated*

with $C^4_{T_7,26}(U^4_{T_7,22}\ (C^4_{T_1,10}(I^4_{T_1,8}(x_4))))$*, i.e., it was created by an update of Transaction* T_7*, that updated a tuple inserted by* T_1*. Based on the outermost commit annotation we know that this tuple version is visible to transactions starting after version* 25*. We use the relational encoding of* $\mathcal{K}^\nu$*-relations from [5] restricted to tuples affected by a given transaction to be able to compute provenance using a regular DBMS and to limit provenance to a transaction of interest for a user. Figure 4 shows the relational encoding of Bonus restricted to the part of the history corresponding to transaction* T_7*. We abbreviate relation* `Bonus` *as* B*. Version annotations are represented as boolean attributes (*$\mathcal{U}_i$ *for update* u_i*) which are true if this part of the provenance has this version annotation and false otherwise. The attributes* $\mathcal{U}_1$ *and* $\mathcal{U}_2$ *represent the version annotations for the first update (*u_1*) and second update (*u_2*) of* T_7*. The only tuple in the instance represents the annotation of tuple (1,101, 2000). The annotation contains only a single version annotation* $U^4_{T_7,22}$*. Thus, only the attribute* $\mathcal{U}_2$ *for update* u_2 *corresponding to this version annotation is true and the other attribute encoding a version annotation is set to false. Variables are encoded as the input tuple annotated with the variable (*b_1 *in the example).*

Queries and Update Operations. We use the definition of positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations of [5]. 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 a tuple 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,* $\text{Sch}(R)$ *denote the schema of relation* R*,* t*,* u *denote tuples, and* $k \in K$*. The operators of* $\mathcal{RA}^+$ *on* $\mathcal{K}$*-relations are defined as:*

$$\Pi_A(R)(t) = \sum_{u:u.A=t} R(u) \qquad (R \cup S)(t) = R(t) + S(t)$$

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

$$(R \bowtie S)(t) = R(t[R]) \times S(t[S])$$
$$\text{(for any } \text{Sch}(R) \cup \text{Sch}(S) \text{ tuple } t)$$

Updates are also defined using the operations of the MV-semiring, but updates add new version annotations to previous annotations. The supported updates correspond to SQL constructs `INSERT`, `UPDATE`, and `DELETE`, and `COMMIT`. An operation is executed at a time ν as part of a transaction T. Update operations take as input a normalized, admissible $\mathcal{K}^\nu$-relation R and return the updated version of this $\mathcal{K}^\nu$-relation. 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 tuple identifiers (id_{new}). An update operation $\mathcal{U}[\theta,A,T,\nu](R)$ applies the projection expressions in A to each tuple that fulfills condition θ. Both $\mathcal{U}[\theta,A,T,\nu](R)$ and $\mathcal{D}[\theta,T,\nu](R)$ wrap the annotations of all tuples fulfilling condition θ in version annotations. A commit $\mathcal{C}[T,\nu](R)$ adds commit version annotations.

Definition 3. *Let* R *be an admissible* $\mathcal{K}^\nu$*-relation. We use* $\nu(u)$ *to denote the version (time) when an update* u *was executed and* $id(k)$ *to denote the id of the outermost version annotation of* $k \in K^\nu$*. Let* A *be a list of projection expressions with the same arity as* R*, and* id_{new} *to denote a fresh id that is deterministically created as discussed below. Let* Q *be a query over a database* D *such that for every* $\{t \to k\}$ *operation in* Q *we have* $k \in \mathcal{K}$*. The update operations on* $\mathcal{K}^\nu$*-relations are defined as:*

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

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

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

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

$$\text{COM}[T,\nu](k) = \begin{cases} C^{id}_{T,\nu+1}(k) & \text{if } k = I/U/D^{id}_{T,\nu'}(k') \\ k & \text{else} \end{cases}$$

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, abbreviating *Software Architect* as *SA* the first update of example transaction T_7 would be written as

$$\mathcal{U}[ID = 101, 'SA' \to position, T_7, 20](Employee)$$

What tuple identifiers are assigned by inserts to new tuples is irrelevant as long as identifiers are deterministic and fulfill certain uniqueness requirements. Thus, we ignore identifier assignment here (see [5] for a detailed discussion).

3. CHALLENGES AND CONTRIBUTIONS

Adapting the MV-semiring model and reenactment approach to RC-SI is challenging, because the visibility rules of RC-SI are more complex than SI, i.e., different statements within a transaction can see different snapshots of the database. Under SI, the first statement of a transaction T sees a snapshot as of the time when T started and later statements see the same snapshot and the modifications of previous updates from the same transaction. Under RC-SI, each statement u also sees modifications of earlier updates from the same transaction, but in addition sees updates of concurrent transactions that committed before u executed. This greatly complicates the definition of transactional semantics in the MV-semiring model. However, as we will demonstrate it is possible to define RC-SI semantics for MV-annotated databases without extending the annotation model (only the visibility rules have to be adapted). Under SI reenactment, queries for individual updates can simply be chained together to construct the reenactment query for a transaction. However, a naive extension of this idea to RC-SI would require us to generate the version of the database seen by a certain statement u by carefully merging the snapshot of the database at the time of u's execution with the previous changes by u's transaction. Thus, while the SI reenactment query for a transaction has to read each updated relation only once, a naive approach for RC-SI would have to read each relation R once for each update that affected it. We present a solution that only has to read each

relation once in most cases. Consequently, it significantly reduces the complexity of provenance computation for RC-SI transactions. The main contributions of this work are:

- We extend the **multi-version provenance model**, a provenance model for database queries, updates, and transactions to support RC-SI concurrency control protocol (Section 5).
- We extend our **reenactment** approach to support computing provenance of RC-SI workloads and present several novel optimizations that are specific to RC-SI including a technique for reducing the number of relation accesses in reenactment (Section 6).
- Our experimental evaluation demonstrates that reenactment for RC-SI is efficient and scales to large databases and complex workloads (Section 8).

4. RELATED WORK

Green et al. [17] have introduced provenance polynomials and the semiring annotation model which generalizes several other provenance models for positive relational algebra including Why-provenance, minimal Why-provenance [12], and Lineage [13]. This model has been studied intensively covering diverse topics such as relations annotated with annotations from multiple semirings [20], rewriting queries to minimize provenance [3], factorization of provenance polynomials [22], extraction of provenance polynomials from the PI-CS [15] and Provenance Games [19] models, and extensions to set difference [14] and aggregation [4]. Systems such as DBNotes [10], LogicBlox [16], Perm [15], Lipstick [2], and others encode provenance annotations as standard relations and use query rewrite techniques to propagate these annotations during query processing. Many use cases such as auditing and post-mortem transaction debugging require provenance for update operations and particularly transactions. In [5, 7], we have introduced an extension of the semiring model for SI transactional histories that is the first provenance model supporting concurrent transactions and have pioneered the reenactment approach for computing such provenance over regular relational databases. Several papers [11, 23, 8] study provenance for updates, e.g., Vansummeren et al. [23] compute provenance for SQL DML statements. This approach alters updates to eagerly compute provenance. However, developing a provenance model for transactional updates is more challenging as it requires to consider the complex interdependencies between tuple versions that are produced by concurrent transactions under different isolation levels. In this work we present the non-trivial extensions of our previous approach [5, 7] to efficiently support the RC-SI protocol which is widely used in practice.

5. READ-COMMITTED SI HISTORIES

We now define the semantics of RC-SI histories over $\mathcal{K}^{\nu}$-relations. Importantly, our extension uses standard MV-semirings and update operations. 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 the execution order of operations is encoded in the updates itself, because each update u in the MV-semiring model is associated with a version identifier $\nu(u)$ determining the order of operations.

Given a RC-SI history H we define $R[\nu]$, the annotated state of relation R at a time ν and $R[T, \nu]$, the annotated state of relation R visible to transaction T at time ν. Note that these two states may differ, because transaction T's updates only become visible to other transactions after T has committed. As in [5] we assume that histories are applied to an empty initial database. For instance, Figure 3 shows a subset of $D[26]$, the version of the example DB after execution of the history (Figure 5) over $D[18]$ (shown in Figure 2). The database state $D[18]$ is the result of running Transaction T_0 that inserted the content of the `Employee` relation and T_1, T_2, T_4 which created the tuples in relation `Bonus`.

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 Figure 7.*

Figure 7a: Relation Version in Transaction T at Time ν. To define the content of relation R at time ν within transaction T we have to distinguish between several cases: 1) per convention $R[T, \nu]$ is empty for any $\nu < Start(T)$; 2) at the start of transaction T, $R[T, \nu]$ is same as $R[\nu]$, the version of the relation containing changes of transactions committed before ν; 3) if an update was executed by transaction T at time $\nu - 1$ then its effect is reflected in $R[T, \nu]$. The update will see tuple versions created by transactions that committed before $\nu - 1$ and tuple versions created by the transaction's own updates. We use $R_{ext}[T, \nu - 1]$ to denote this version of R and explain its construction below; 4) right after transaction commit, the current version of the relation visible within T is the result of applying the commit operator to the previous version; and 5) as long as there is no commit or update on R at $\nu - 1$ then the current version of relation R is the same as the previous one.

Figure 7b: Relation Version Visible to Updates. As mentioned above we use $R_{ext}[T, \nu]$ to denote the version of relation R that is visible to an update of transaction T executed at time ν. This state of relation R contains all tuple versions created by committed transactions as long as they have not been overwritten by a previous update of transaction T (the first sum) and tuple versions created by previous updates of transaction T (the second sum). Here by overwritten we mean that a tuple version is no longer valid, because either it has been deleted or because it was updated and, thus, it has been replaced with a new updated version. Function validEx implements this check. It returns 1 if the tuple version has not been overwritten and 0 otherwise. This function uses a predicate $\text{UPDATED}(T, t, k, \nu)$ which is true if transaction T has invalidated summand k in the annotation of tuple t before ν by either deleting or updating the corresponding tuple version. The second sum ranges over tuple versions $R[T, \nu]$ excluding tuple versions not created by transaction T (function validIn).

Figure 7c: Committed Relation Version. The committed version $R[\nu]$ of a relation R at time ν contains all changes of transactions that committed before ν. That is, all tuple versions created by any such transaction unless the tuple version is no longer valid at ν, e.g., it got deleted by another transaction. Thus, this version of relation R can be computed as the sum over all annotations on tuple t in the

(a) Historic relation $R[T,\nu]$: version of R seen by Transaction T at Time ν

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

(b) $R_{ext}[T,\nu]$: Tuple versions visible within Transaction T at Time ν

$$R_{ext}[T,\nu](t) = \sum_{i=0}^{n(R[\nu](t))} R[\nu](t)[i] \times \textsc{validEx}(T,t,R[\nu](t)[i],\nu) + \sum_{i=0}^{n(R[T,\nu](t))} R[T,\nu](t)[i] \times \textsc{validIn}(T,t,R[T,\nu](t)[i],\nu)$$

(c) $R[\nu]$: Committed tuple versions at Time ν

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

(d) Validity of summands (tuple versions) within annotations

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

$$\textsc{validEx}(T,t,k,\nu) = 0 \text{ if } \textsc{updated}(T,t,k,\nu), 1 \text{ otherwise}$$

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

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

Figure 7: Historic relational instances induced by History H. $R[T,\nu]$ is the annotated instance visible by Transaction T at version ν. $R[\nu]$ is the instance containing all changes of transactions committed before version ν. Each update of a transaction sees all modifications of previous updates from the same transaction as well as modifications of transactions committed before the update was run ($R_{ext}[T,\nu]$).

versions of relation R created by past transactions. However, in addition to ensuring that outdated tuple versions are not considered we also need to ensure that every tuple version is only included once. Both conditions are modelled by function $\textsc{validAt}(T,t,k,\nu)$ that return 1 if k is a summand (tuple version) in the annotation of tuple t at time ν and was created by T (this ensures that each tuple version is only added once).

Example 3. *Consider the example transactional history from Figure 5. For instance, $Bonus[T_8,22]$ is the version of the* `Bonus` *relation seen by the insert operation of Transaction T_8 and is equal to $Bonus[22]$ (case 2, Figure 7a). It contains the tuples from the* `Bonus` *relation as shown in Figure 2, because these tuples were created by transactions that committed before time 22 (they are in $Bonus[T_8,22]$). Thus,* validAt *returns 1 for these tuples. For instance, tuple b_1 has been updated by Transaction T_7 (the new version is denoted as b_1') before version 22, but this transaction has not committed yet. Since T_8 has not updated b_1,* validEx *returns 1 and the full annotation of b_1 in $Bonus[22]$ is as shown in Figure 2.*

6. REENACTMENT

We have introduced reenactment [5] as a mechanism to construct a $\mathcal{K}^\nu$-annotated relation R produced by a transaction T that is part of a history H by running a so-called reenactment query $\mathbb{R}(T)$. We have proven [5] that $\mathbb{R}(T) \equiv_{\mathbb{N}[X]^\nu} T$, i.e., the reenactment query returns the same annotated relation as the original transaction ran in the context of history H (has the same result and provenance). In this work, we present reenactment for single RC-SI transactions as well as extensions necessary to reenact a whole history. The latter requires the introduction of a operator which merges the relations produced by the reenactment queries of several transactions. This operator is also needed to compute $R_{ext}[T,\nu]$ as introduced in the previous section. After introducing this operator, we first present a method to reenact RC-SI transactions that requires merging newly committed tuples into the version of a relation visible within the reenacted transaction after every update. We then present an optimization that requires no merging in most cases and uses another new operator - version filtering.

Version Annotation Operator. For reenactment of updates and transactions we need to be able to introduce new version annotations in queries. However, the operators of $\mathcal{RA}^+$ do not support that. To address this problem, we have defined the version annotation operator in [5]. For $X \in \{I,U,D\}$ the version annotation operator $\alpha_{X,T,\nu}(R)$ 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 $\textsc{com}[T,\nu](k)$ from Definition 3.

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

Reenacting Updates. Reenactment queries for transactions are constructed from reenactment queries for single update statements. The reenactment query $\mathbb{R}(u)$ for an update u returns the modified version of the relation targeted by the update if it is evaluated over the database state seen by u's transaction at the time of the update u ($R_{ext}[T,\nu(u)]$). The semantics of update operations is the same no matter whether SI or RC-SI is applied. Thus, we can use the technique we have introduced for SI in [5] to also reenact RC-SI updates. As we will see later, it will be beneficial to let up-

date reenactment queries operate over a different input for RC-SI than for SI which requires modifications to the update reenactment queries. Let H is a history over database D. Below we show the definitions of update reenactment queries from [5]. The reenactment query $\mathbb{R}(u)$ for operation u in H is:

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

For example, an update modifies a relation by applying the expressions from A to tuples that match the update condition θ. All other tuples are not affected. Thus, the result of an update can be computed as the union between these two sets. For instance, the reenactment query $\mathbb{R}(u_2)$ for the update u_2 of running example transaction T_7 is:

$$\alpha_{U,T_7,21}(\Pi_{ID,EmpID,Amount+1000\rightarrow Amount}(\sigma_{ID=101}(Bonus[T_7,21]))) \cup \sigma_{ID\neq 101}(Bonus[T_7,21])$$

Transaction and History Reenactment. To reenact a transaction T, we have to connect reenactment queries for the updates of T such that the input of every update u over relation R is $R_{ext}[T,\nu(u)]$. As discussed in Section 5, this instance of relation R contains tuple versions updated by previous updates of T which targeted R as well as tuple versions from $R[\nu]$. Hence, $R_{ext}[T,\nu(u)]$ can be computed as a union between these two sets of tuples as long as we can filter out tuple versions (summands in annotations) that are no longer valid. We now introduce a new query operator that implements this filtering and then define reenactment for RC-SI transactions using this operator.

Version Merge Operator. The version merge operator $\mu(R_1, R_2)$ is used to merge two version R_1 and R_2 of a relation R such that 1) tuple versions (summands in annotations) present in both inputs are only included once in the output and 2) if both inputs include different versions of a tuple, then only the newer version is returned. This operator is used to construct $R_{ext}[T,\nu]$ from a union of $R[\nu]$ and $R[T,\nu]$. The definition of $\mu(R_1,R_2)$ is shown below.

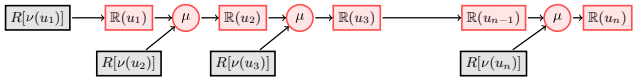
$$\mu(R_1,R_2)(t) = \sum_{i=0}^{n(R_1(t))} R_1(t)[i] \times isMax(R_2, R_1(t)[i]) + \sum_{i=0}^{n(S(t))} R_2(t)[i] \times isStrictMax(R_1, R_2(t)[i])$$

The operator uses two functions $isMax$ and $isStrictMax$. $isMax(R,k)$ returns 0 if relation R contains a newer version of the tuple version encoded as annotation k, i.e., if $\exists t', k', j : idOf(R(t')[j]) = idOf(k) \wedge versionOf(R(t')[j]) > versionOf(k)$. Function $isStrictMax$ is the strict version of $isMax$ which also returns 0 if the tuple version k is present in R, i.e., $versionOf(R(t')[j]) > versionOf(k)$ is replaced with $versionOf(R(t')[j]) \geq versionOf(k)$ in the condition. Here function $idOf(k)$ returns the tuple identifier in the annotation k and $versionOf$ returns the version encoded in the given annotation k. These functions are well defined if k is a summand in a normalized admissible $\mathcal{K}^\nu$-relation (see Section 2):

$$idOf(X^{id}_{T,\nu}(k')) = id \qquad versionOf(X^{id}_{T,\nu}(k')) = \nu$$

As an example consider computing $\mu(Bonus[26], Bonus[19])$. These relation versions are shown in Figure 2 and 3. The later only shows new or updated tuples. For instance, b_2 is present in both relations with the same annotation, a single summand. Thus, the first sum in $\mu(Bonus[26], Bonus[19])(b_2)$ will include this annotation (there is no newer version of this tuple in $Bonus[19]$) while it will be excluded from the second sum (the same annotation is found in $Bonus[26]$). As another example consider tuple b_1 which was updated to b_1' by Transaction T_7. Thus, $\mu(Bonus[26], Bonus[19])(b_1) = 0$, because a newer version of this tuple exists in $Bonus[26]$ and $\mu(Bonus[26], Bonus[19])(b_1') = Bonus[26])(b_1')$ (this is the newest version of this tuple found in $Bonus[19]$ and $Bonus[26]$).

Reenacting Transactions. For simplicity of exposition we present the construction of reenactment queries for transactions updating a single relation R. The construction for transactions updating multiple relations is achieved analog to [5]. The reenactment query for Transaction $T = (u_1, \ldots, u_n, c)$ executed as part of an RC-SI history H is recursively constructed starting with a commit annotation operator applied to the reenactment query $\mathbb{R}(u_n)$ for the last update of T. Then we replace $R[T,\nu(u_n)]$ in the query constructed so far with $\mu(\mathbb{R}(u_{n-1}), R[\nu(u_n)])$. The result of this version merge operator is $R_{ext}[T,\nu(u_n)]$, the input seen by u_n in the history H. This replacement process is repeated for $i \in n-1, \ldots, 1$ until every reference to a version of relation R visible within the transaction has been replaced with references to committed relation versions ($R[\nu]$ for some ν). The structure of the reenactment query is outlined below.



Reducing Relation Accesses. We would like reenactment queries for RC-SI to be defined recursively without requiring to recalculate the right mix of tuple versions from transaction T and from concurrent transactions after each update. To this end we introduce the version filter operator, that filters out summands k from an annotation based on the version encoded in the outermost version annotation of k. The filter condition θ of a version filter operator is expressed using a pseudo attribute V representing the ν encoded in version annotations. We use this operator to filter summands from annotations based on the version annotations they are wrapped in.

Version Filter Operator. The version filter operator removes summands from an annotation based on the time ν in their outermost version annotation. Let θ be a condition over pseudo attribute V. Given a summand $k = X^i_{T,\nu}(k')$ such a condition is evaluated by replacing V with ν in θ. The version filter operator using such a condition θ is defined as:

$$\gamma_\theta(R)(t) = \sum_{i=0}^{n(R(t))} R(t)[i] \times \theta(R(t)[i])$$

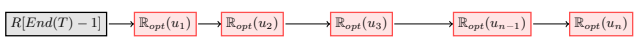
For example, we could use $\gamma_{V<11}(R)$ to filter out summands from annotations of tuples from a relation R that were added after time 10. In contrast to regular selection, a version filter's condition is evaluated over the individual summands in an annotation.

Our optimized reenactment approach for RC-SI is based on the following observation. Consider a tuple t updated by

Transaction T and let $u \in T$ be the first update of Transaction T that modified this tuple. Let t' denote the version of tuple t valid before u. Given the RC-SI semantics, t' is obviously present in $R[\nu(u)]$ and was produced by a transaction that committed before $\nu(u)$. Importantly, t' is guaranteed to be in $R[End(T)]$, i.e, the version of R immediately before the commit of Transaction T. To see why this is the case recall that T would have obtained a write-lock on this tuple to be able to update t' to t and this write-lock is held until transaction commit. Thus, it is guaranteed that no other transaction would have been able to update t' before the commit of T. Based on this observation, we can use $R[End(T)]$ as an input to the reenactment query as long as we ensure that the reenactment queries for other updates of T executed before u ignore t'. We achieve this using the version filter operator to filter out tuple versions that were not visible to an update u'. It is applied in the input of the part of the transaction reenactment query corresponding to the update u'. In the optimized reenactment query, the initial input of reenactment is $R[End(T)]$ instead of $R[Start(T)]$. Furthermore, the update reenactment queries are modified as shown below. An optimized reenactment query $\mathbb{R}_{opt}(u)$ for update u passes on unmodified versions of tuples that are not visible to update u. We use $\mathbb{R}_{opt}(T)$ to denote the optimized transaction reenactment query. In the formulas shown below, R denotes the result of the reenactment query for the previous update or $R[End(T)-1]$ (in case the update is the first update of the transaction). Note that this optimization is only applicable if the inserts in the transaction do not access the relation that is modified by the updates and deletes of the transaction. That is because the query of an insert may read tuple version that are not in $D[End(T)]$. Hence, we only apply this optimization if the inserts of Transaction T use the `VALUES` clause (the singleton operator $\{t \to k\}$ as defined in Section 2).

$$\begin{aligned}\mathbb{R}_{opt}(\mathcal{U}[\theta, A, T, \nu](R)) = \alpha_{U,T,\nu+1}(\Pi_A(\sigma_\theta(\gamma_{V \leq \nu(u)}(R))))\\ \cup\ \sigma_{\neg\theta}(\gamma_{V \leq \nu(u)}(R))\\ \cup\ \gamma_{V > \nu(u)}(R))\\ \mathbb{R}_{opt}(\mathcal{D}[\theta, T, \nu](R)) = \alpha_{D,T,\nu+1}(\sigma_\theta(\gamma_{V \leq \nu(u)}(R)))\\ \cup\ \sigma_{\neg\theta}(\gamma_{V \leq \nu(u)}(R))\\ \cup\ \gamma_{V > \nu(u)}(R)\end{aligned}$$

For example, the reenactment query for an update u distinguishes between three disjoint cases: 1) a tuple that is visible to the update ($V \leq \nu(u)$) and fulfills the update's condition, i.e., the tuple is updated by u; 2) a tuple that is visible to the update, but does not fulfill the condition θ; and 3) a tuple version that is not visible to u, because it was created by a transaction that committed after $\nu(u)$. The structure of the resulting reenactment query for transactions without inserts is shown below. Note that relation R is only accessed once by the reenactment query.



For each insert using the `VALUES` clause a new tuple will be added to the relation R using `UNION`.

Reenactment queries for RC-SI transactions are equivalent to the transaction they are reenacting.

Theorem 1. *Let T be a RC-SI transaction. Then, $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T) \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}_{opt}(T)$.*

Proof. See the long version of this paper [6]. □

To create a reenactment query for a (partial) history, we combine the results of reenactment queries for all transactions in the history using the version merge operator. Each reference to a committed version of a relation $T[\nu]$ is replaced with a multiway merge of the results of reenactment queries for transactions $T \in H$ that committed before ν in the order of commit. For example, if two transactions T_1 and T_2 have committed before ν then $R[\nu]$ is computed as

$$q = \mu(\mathbb{R}(T_1), \mathbb{R}(T_2))$$

Later versions can then be computed by reusing this query result, e.g., if the next transaction to commit in the history was T_3, then the version of R at $End(T_3)+1$ is computed as $\mu(q, \mathbb{R}(T_3))$.

7. IMPLEMENTATION

GProM is a middleware that implements reenactment for SI over standard DBMS using a relational encoding of MV-relations [5, 7]. Reenactment is implemented as SQL queries over this encoding. We have extended the system to implement RC-SI reenactment using the same relational encoding. One advantage of this system is that provenance requests are considered as queries and can be used as subqueries in an SQL statement, e.g., to query or store provenance. In Section 8 we study the performance of queries over provenance. GProM assumes that the underlying database system on which we want to execute provenance computations keeps an audit log that can be queried and provides at least the information as shown in Figure 5. Furthermore, the DBMS has to support time travel for the system to query past states of relations (this is used to reenact single transactions and partial histories). For instance, Oracle, DB2, and MSSQL support both features. While a full description of the implementation and additional optimizations is beyond the scope of this paper, we give a brief overview of the additional optimizations that we have implemented: 1) as we observed in [5], reenactment queries can contain a large number of union operations that may lead to bad performance if they are unfolded by the DBMS. We extend our approach for using `CASE` to avoid union operations [5] to RC-SI; 2) if the user is only interested in the provenance of tuples modified by a particular transaction, then this can be supported by filtering tuples from the output of the transaction's reenactment query that were not affected by the transaction. We did present two methods for improving the efficiency of this filter step by either removing tuples from the input of the reenactment query which do not fulfill the condition of any update of the transaction or by retrieving updated tuple versions from the database version after transaction commit and using this set to filter the input using a join. We have adapted both methods for RC-SI; 3) the version merge operator is implemented using aggregation to determine the latest version of each tuple.

8. EXPERIMENTS

Using commercial DBMS X, we evaluate 1) the performance of provenance computation using reenactment for isolation level *RC-SI* and comparing it with *SI*, and 2) the performance of querying provenance. All experiments were run on a machine with 2 x AMD Opteron 4238 CPUs (12 cores

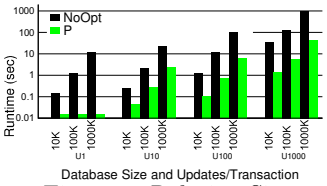


Figure 8: Relation Size

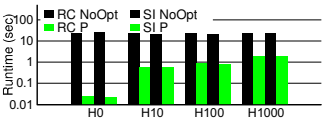


Figure 9: History Size

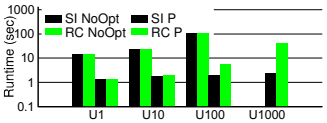


Figure 10: Isolation Levels

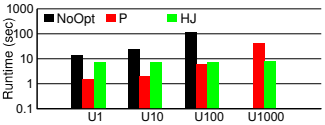


Figure 11: Optimization

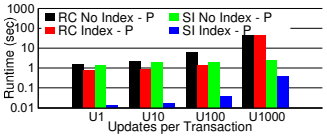


Figure 12: Index vs. No Index

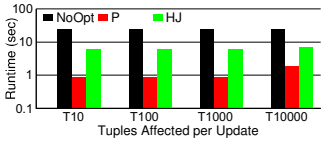


Figure 13: Affected Tuples

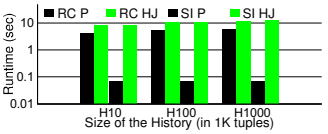


Figure 14: Inserts and Deletes

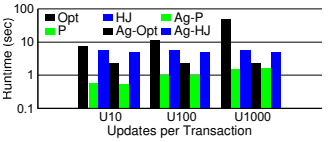


Figure 15: Aggregation

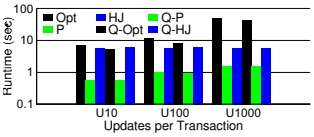


Figure 16: Query Provenance

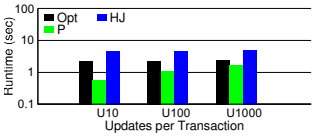


Figure 17: Query Vers. Ann.

total), 128 GB RAM, and 4 x 1TB 7.2K HDs in a hardware RAID 5 configuration. We have studied the runtime and storage overhead of DBMS X's build-in temporal and audit features in [5]. The results demonstrated that the runtime overhead for transaction execution is below 20% when audit logging and time travel are activated and it is more efficient than eager materialization of provenance during transaction execution (about 133% overhead and higher). We did confirm the same trend for RC-SI and, thus, do not present these results here.

Datasets and Workload. In all experiments, we use a relation with five numeric columns. Values for these attributes were generated randomly using a uniform distribution. Different variants $R10K$, $R100K$, and $R1000K$ with 10K, 100K, and 1M tuples and no significant history ($H0$) were created. Moreover, three variants of $R1000K$ with different history sizes $H10$, $H100$, and $H1000$ (100K, 1M, and 10M tuples of history) are used. In most experiments, transactions consist only of update statements. The tuple to be updated is chosen randomly by its primary key. The following parameters are used in experiments: U is the number of updates per transaction (e.g., *U100* is a transaction with 100 updates). T is the number of tuples affected by each update (default is $T1$). Transactions were executed under isolation level *RC-SI* (default) or *SI*. Experiments were repeated 100 times and the average runtime is reported.

Compared Methods. We apply different configurations for computing provenance of transactions using a subset of the optimizations outlined in Section 7. **NoOpt (N)**: Computes the provenance of all tuples in a relation including tuples that were not affected by the transaction. **Opt (O)**: Like the previous option but GProM's heuristic relational algebra optimizations are activated. **Prefilter (P)**: Only returns provenance of tuples affected by the transaction by prefiltering (Section 7). **HistJoin (HJ)**: Same as P, but using the join method as described in Section 7.

Provenance Computation. For the following experiments we have executed the transactional workload beforehand and measure performance of provenance capture.

Relation Size and Updates/Transaction. We consider relations of different size ($R10K$, $R100K$, and $R1000K$) that do not have any significant history ($H0$). Figure 8 shows performance of computing provenance of transactions with different number of updates (*U1* up to $U1000$). We applied N and P. We scale linearly in R and U. By reducing the amount of data to be processed, the P approach is orders of magnitude faster than the N configuration.

History Size. Figure 9 shows the results for relations with 1M tuples ($R1000K$) and varying history sizes ($H0$, $H10$, $H100$, and $H1000$). We compute provenance of transactions with 10 updates (*U10*). Method N has almost constant performance for both isolation levels *RC-SI* and *SI*. The P approach displays better performance as it has to process less tuples. Its performance decreases for relations with a large history size.

Isolation Levels. Figure 10 compares the result of transactions under isolation levels *SI* and *RC-SI* with varying number of updates per transaction (*U1* to *U1000*). This experiment was conducted over table *R1000K-H1000*. The runtime of N is not affected by the choice of isolation level, because the main difference between *SI* and *RC-SI* reenactment is that we need to check whether a row version is visibile for each update. However, the impact of these checks is negligible for N as the major cost factors are scanning the table and large parts of its history as well as producing 1M output rows. For the more efficient P configuration this effect is more noticeable, especially for larger number of updates per transaction. Note that for *U1000* the N method did not finish within the allocated time budget (1000 sec.).

Comparing Optimization Techniques. Figure 11 compares different optimization methods (N, P, and HJ) for varying number of updates (*U1*, *U10*, *U100*, and *U1000*) using *R1000K-H1000*. Both P and HJ outperform N with a more pronounced effect for larger number of updates per transaction. P outperforms HJ for *U1* by a factor of 5 whereas this result is reversed for *U1000*. The runtime of HJ is almost not affected by parameter U, because it is dominated by the temporal join.

Index vs. No Index. We have studied the effect of using indexes for the relation storing the history of a relation. We use *R1000K-H1000* and vary U (*U1* to *U1000*). Figure 12 compares the effect of indexes for isolation levels *RC-SI* and *SI* using P. The results demonstrate that using indexes improves execution time of queries that apply P considerably. Provenance computation for *SI* benefits more from indexes, because the prefilter conditions applied by the P method are simpler for *SI*.

Affected Tuples Per Update. We now fix *U10* and *R1000K-H1000*, and vary the number of tuples (T) affected by each update from 10 to 10,000. The runtime (Figure 13) is dominated by scanning the history and filtering out updated tuples (P) or the self-join between historic relations (HJ). Increasing the T parameter by 3 orders of magnitude increases runtime by about 120% (P) and 9% (HJ) whereas it does not effect runtime of queries using N.

Inserts and Deletes. We now consider transactions that use inserts, deletes, and updates over $R1000K$ varying history size ($H10$ to $H1000$). Each statement in a transaction is chosen randomly with equal probability to be an insert, update, or delete. Figure 14 presents the result for $U20$. Performance is comparable to performance for updates for *RC-SI*. This aligns with our previous findings for *SI*.

Querying Provenance. In GProM, provenance computations can be used as subqueries of a more complex SQL query. We now measure performance of querying provenance (the runtimes include the runtime of the subquery computing provenance). All experiments of this section are run over relation $R1000K - H0$ and transactions with *U10* to *U1000*.

Aggregation of Provenance Information. Figure 15 shows the results for running an aggregation over the provenance computation (denoted as *Ag-*). These results indicate that the performance of aggregation on provenance information is comparable to provenance computation. Even more, aggregation considerably improves performance for (O). For *U1000*, *Ag-O* results in 95% improvement over O (because it reduces the size of the output) while *Ag-HJ* improves performance by $\sim$ 13% compared to *HJ*.

Filtering Provenance. A user may only be interested in part of the provenance that fulfills certain selection conditions, e.g., bonuses larger than a certain amount. Figure 16 shows the runtime of provenance computation and querying (denoted as *Q-*). Performance of querying the results of provenance capture is actually slightly better than just computing provenance, because it reduces the size of the output and selection conditions over provenance are pushed into the SQL query implementing the provenance computation.

Querying Versions Annotations. A user can also query version annotations which are shown as boolean attributes in the provenance, e.g., to only return provenance for tuples that were updated by a certain update of the transaction. Figure 17 shows the performance results for such queries. We fix an update $u \in T$ and only return provenance of tuples modified by this update. This reduces the runtime of O queries significantly by reducing the size of the output.

9. CONCLUSIONS

We have presented an efficient solution for computing the provenance of transactions run under RC-SI by extending our MV-semiring model and reenactment approach. Our experimental evaluation demonstrates that our novel optimizations specific to RC-SI enables us to achieve performance comparable to SI reenactment. In future work, we would like to explore the application of reenactment for post-mortem debugging of transactions which is particularly important for lower isolations level such as RC-SI.

10. REFERENCES

[1] Oracle - FBA Whitepaper. http://www.oracle.com/technology/products/database/oracle11g/pdf/flashback-data-archivewhitepaper.pdf. Accessed: 2015, Jul 1.

[2] Y. Amsterdamer, S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.

[3] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. In *PODS*, pages 141–152, 2011.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for Aggregate Queries. In *PODS*, pages 153–164, 2011.

[5] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Formal foundations of reenactment and transaction provenance. Technical report, Illinois Institute of Technology, 2016.

[6] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation (long version). Technical report, Illinois Institute of Technology, 2016. http://cs.iit.edu/%7Edbgroup/pdfpubls/AG16a.pdf.

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

[8] 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. 2013.

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

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

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

[12] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.

[13] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *TODS*, 25(2):179–227, 2000.

[14] F. Geerts and A. Poggi. On database query languages for K-relations. *Journal of Applied Logic*, 8(2):173–185, 2010.

[15] 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. 2013.

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

[17] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.

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

[19] S. Köhler, B. Ludäscher, and D. Zinn. First-order provenance games. In *In Search of Elegance in the Theory and Practice of Computation*, pages 382–399. 2013.

[20] E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *ICDT*, pages 196–207, 2012.

[21] D. B. Lomet and F. Li. Improving transaction-time dbms performance and functionality. In *ICDE*, pages 581–591, 2009.

[22] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.

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