Formal Foundations of Reenactment and Transaction Provenance

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

IIT DB Group Technical Report
IIT/CS-DB-2016-01

2016-03

http://www.cs.iit.edu/~dbgroup/

Abstract Provenance is essential for auditing, data debugging, understanding transformations, and many additional use cases. All these use cases would benefit from provenance for transactional updates. We present a provenance model for snapshot isolation transactions extending the semiring framework with version annotations and updates. Based on this model, we present the first solution for computing the provenance of transactions. Our approach retroactively traces provenance using an audit log and time travel functionality (supported by many DBMS) without having to store any additional information. For a given transaction, we construct a *reenactment query* that simulates the effect of the transaction. This query returns the updated versions of relations produced by the transaction and has the same provenance as the transaction. Interestingly, such reenactment queries can be expressed in relational algebra and, thus, be executed by standard DBMS. We have implemented a prototype on top of a commercial database system and our experiments confirm that by applying novel optimizations we can efficiently compute the provenance of large transactions over large data sets and our approach results in only moderate overhead for transactions when no provenance is requested.

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, understanding complex transformations, and as a supporting technology for integration and probabilistic databases. How to model and compute 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 each tuple in the result of a query will be annotated with input tuples that are in its provenance and, depending on the provenance model that is employed, also how these tuples were combined to derive the result. Such techniques have been pioneered by systems such as Perm [17], DBNotes [6], Orchestra [22], and others. While provenance for queries is important, many use cases (e.g., auditing) would benefit from provenance for update operations. 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 and/or updated by past transactions). Relational databases execute updates as part of transactions and apply concurrency control techniques to guarantee ACID properties for transactions. Provenance tracking for database updates needs to take into account the idiosyncrasies of concurrency control protocols to correctly describe the origin of data. We present the first solution to this problem. Specifically, we extend an existing provenance model for queries (the semiring model) to also support transactional semantics, demonstrate how to compute provenance according to this model using a relational database, and present an implementation in our GProM system. We introduce *reenactment*, a novel technique for replaying ("reenacting") a transactional history (or parts thereof) using queries. Reenactment queries enable us to retroactively compute the provenance of tuple versions produced by a transactional history using transaction time histories of relations and a log of SQL statements. Notably, our approach does not require any eager materialization of provenance during transaction execution nor any changes to the transactions themselves. Hence, we avoid paying the runtime and storage overhead of provenance computation for every transaction executed by the system. We focus on transactions executed under the *snapshot isolation* (*SI*) concurrency control protocol (used by, e.g., PostgreSQL, Oracle, and MSSQL) in this work.

Example 1.1 *Consider the database shown in Figure 1a. Relation* `Order` *stores orders submitted by customers. Relation* `Outstanding` *stores outstanding payments for orders and their due dates. Relation* `Collection` *stores outstanding payments that have not been payed by the due date and, thus, went to collection. A history of transactions for this database is shown in Figure 2. Here we assume a discrete time domain* Version *and show for each update the version at which it was executed. Transaction T_1 creates two orders for customer Peter and inserts the order amounts as outstanding payments. Peter recognizes an error in the order and contacts customer service. The service operator corrects the error (Transaction T_2) by updating the order table but forgets to modify Peter's outstanding payment accordingly. Some time later Peter decides to pay off his dept. Assuming that the error has been corrected, he pays \$250 for his first order and \$56 for his second order. This triggers transactions T_3 and T_4 which update the due amount (respectively delete outstanding amounts that have been fully payed). These transactions run a query that collects payments that are overdue and inserts them into the collections table.*

Assume that the database applies the snapshot isolation [5] concurrency control protocol. Figure 1 shows the states of the database after the commit of transactions T_1, T_2, T_3 and T_4, respectively. For convenience, we show tuple identifiers to the right of each tuple. Ignore the annotations shown on the left for now. Under snap-

(a) After Transaction T_1

Order

	id	customer	price	
$C^1_{T_1,6}(I^1_{T_1,2}(x_1))$	oid_1	Peter	300	o_1
$C^2_{T_1,6}(I^2_{T_1,4}(x_2))$	oid_3	Peter	56	o_2

Outstanding

	order	amount	due	
$C^3_{T_1,6}(I^3_{T_1,3}(I^1_{T_1,2}(x_1)))$	oid_1	300	2000-05	s_1
$C^4_{T_1,6}(I^4_{T_1,5}(I^2_{T_1,4}(x_2)))$	oid_3	56	2000-05	s_2

(b) After Transaction T_2

Order

	id	customer	price	
$C^1_{T_2,8}(U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1))))$	oid_1	Peter	250	o'_1
$C^2_{T_1,6}(I^2_{T_1,4}(x_2))$	oid_3	Peter	56	o_2

Outstanding

	order	amount	due	
$C^3_{T_1,6}(I^3_{T_1,3}(I^1_{T_1,2}(x_1)))$	oid_1	300	2000-05	s_1
$C^4_{T_1,6}(I^4_{T_1,5}(I^2_{T_1,4}(x_2)))$	oid_3	56	2000-05	s_2

(c) After Transaction T_3

Order

	id	customer	price	
$C^1_{T_2,8}(\dots)$	oid_1	Peter	250	o'_1
$C^2_{T_1,6}(\dots)$	oid_3	Peter	56	o_2

Outstanding

	order	amount	due	
$C^3_{T_3,13}(U^3_{T_3,9}(C^3_{T_1,6}(\dots)))$	oid_1	50	2000-05	s'_1
$C^4_{T_1,6}(\dots)$	oid_3	56	2000-05	s_2

Collection

	order	amount	
$C^5_{T_3,13}(I^5_{T_3,11}(U^3_{T_3,9}(C^3_{T_1,6}(\dots))))$	oid_1	50	c_1
$C^6_{T_3,13}(I^6_{T_3,11}(C^4_{T_1,6}(\dots)))$	oid_3	56	c_2

(d) After Transaction T_4

Order

	id	customer	price	
$C^1_{T_2,8}(\dots)$	oid_1	Peter	250	o'_1
$C^2_{T_1,6}(\dots)$	oid_3	Peter	56	o_2

Outstanding

	order	amount	due	
$C^3_{T_3,13}(\dots)$	oid_1	50	2000-05	s'_1
$C^4_{T_4,14}(D^4_{T_4,10}(C^4_{T_1,6}(\dots)))$	oid_3	56	2000-05	s'_2

Collection

	order	amount	
$C^5_{T_3,13}(\dots)$	oid_1	50	c_1
$C^6_{T_3,13}(\dots)$	oid_3	56	c_2
$C^7_{T_4,14}(I^7_{T_4,12}(C^3_{T_1,6}(\dots)))$	oid_1	300	c_3

Fig. 1: Running example database state after execution of each of the 4 transactions in the example history. Attribute values affected by an update are highlighted in red (🟥) and deleted tuples are highlighted in gray (▨).

Transaction T_1	Version
INSERT INTO Order VALUES (oid_1,Peter,300);	1
INSERT INTO Outstanding (SELECT id, price, '2005-05' FROM Order WHERE id=oid_1);	2
INSERT INTO Order VALUES (oid_2,Peter,56);	3
INSERT INTO Outstanding (SELECT id, price, '2005-05' FROM Order WHERE id=oid_2);	4
COMMIT;	5

Transaction T_2	Transaction T_3	Transaction T_4	
UPDATE Order SET price = 250 WHERE id = oid_1;			6
COMMIT;			7
	UPDATE Outstanding SET amount = amount − 250 WHERE order = oid_1;		8
		DELETE FROM Outstanding WHERE order = oid_3;	9
	INSERT INTO collection (SELECT order, amount FROM Outstanding WHERE due < '2000-06');		10
		INSERT INTO collection (SELECT order, amount FROM Outstanding WHERE due < '2000-06');	11
	COMMIT;		12
		COMMIT;	13

Fig. 2: Transactional history H of the running example, showing version identifiers for each statement.

shot isolation each transaction T operates on a private snapshot of the database that contains all tuple versions that were produced by transactions that committed before T started and versions created by T's own updates. This has no effect on transactions T_1 and T_2, because the execution of these transactions does not overlap with any other transaction. Transactions T_3 and T_4, however, do not see each other's updates. These transactions both operate on the version of relation `Outstanding` *produced by Transaction T_2.*[1] *This causes three (instead of one) collection tuples to be created (see Figure 1): Transaction T_3 sees a remaining balance of \$50 for order oid_1 and does not see the deletion of the outstanding payment for order oid_2. Transaction T_4, sees the previous balance of \$300 for order oid_1.*

Peter, surprised to receive a letter about outstanding payments, calls the customer service again. The representative will not be able to explain why the due amounts went to collection, because the current database state (Figure 1d) provides no hint at what caused the errors. A temporal database would reveal more information, e.g., showing the unmodified \$300 amount in the `Outstanding` *relation after the commit of Transaction T_2. However, critical information needed to understand the problem in this example is not available since a temporal database does not reveal which tuple versions the new collection tuples have been derived from and by which operations. For example, the representative needs to understand that tuple c_1 was derived from the tuple version s_1' produced by subtracting* \$250 *from the amount of the previous version of this tuple (s_1). Furthermore, this previous tuple version s_1 was derived from tuple version o_1. That is the update of Transaction T_2 (correction of the order price) was not taken into account. If a log of executed SQL statements is available (we call this an* audit log*), then the user may be able to correlate the transaction time history of the database with the audit log to infer such provenance dependencies. However, for any realistically complex SQL query and realistically sized database it would not be feasible to apply this inference manually.*

This example motivates the need for an approach for tracking the provenance of tuples that are updated by concurrent transactions. A provenance model for transactions would not just have to explain how tuple versions have been combined to produce new tuple versions, but also which SQL statements did create which tuple version (and how). Ideally, it should be possible to compute the provenance of any current or past tuple version without having to eagerly materialize provenance information during transaction execution. Note that in this work we do not consider provenance dependencies at the application side. 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. While there are existing solutions for computing the provenance of updates [22,27,8], none of these approaches support transactions and these approaches are not integrated with provenance for queries. As we will demonstrate in the following, naive combinations of existing provenance models with snapshot isolation do not fulfill our desiderata for a transaction provenance model. Techniques for replaying operations (e.g., [29]) are also not directly applicable to our problem because we do not want to pay the overhead of replaying DB updates (e.g., I/O caused by writing logs and changes to disk). The main contributions of this work are:

•We introduce the **multi-version semiring model**, a provenance model for queries and updates that extends the semiring annotation framework [21]. In particular, we support transactions executed using the snapshot isolation concurrency control protocol. The provenance of a tuple version in this model encodes its complete derivation history including previous tuple versions that were used to compute it and how tuple versions have been used by updates and/or queries involved in its creation.

•Based on this model, we introduce the novel concept of **reenactment queries**. Reenactment queries are queries that simulate the effect of an update, transaction, or even a whole history. Importantly, these queries are annotation equivalent to the operation(s) they are simulating, i.e., they produce the same result (updated relations) and have the same provenance. Reenactment is the main enabler of our approach for computing the provenance of transactions, because it enables us to compute provenance retroactively by running reenactment queries instead of having to compute and materialize it eagerly while transactions are running.

•We present a **relational encoding** of our provenance model and demonstrate how to implement provenance computation for transactions by translating reenactment queries into SQL queries using **time travel** to access past database states. By *time travel* we mean the ability to access past states of a relation in queries as supported by, e.g., Oracle, IBM, and MSSQL.[2] We

[1] Readers familiar with snapshot isolation may recognize that transactions T_3 and T_4 are an instance of the write-skew problem [5], i.e., this history is not serializable.

[2] If not natively supported, time travel can be implemented using triggers to maintain a transaction time history in separate history relations (e.g., see [26], Chapter 8).

use an **audit log** (log of executed SQL statements) to construct reenactment queries.

•We implement our techniques in the GProM system running on-top of DBMS X [3]. We discuss several optimizations including alternative ways of implementing reenactment queries and filtering unrelated data from the provenance computation early on.

•Our experiments demonstrate that 1) provenance computation 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 for running transactions when time travel and audit logging is activated is tolerable and significantly smaller than the overhead incurred by eagerly computing and materializing provenance for every transaction when it is executed.

The remainder of this paper is organized as follows. Section 2 presents an overview of our system. In Section 3, we review related work and then introduce background on provenance and concurrency control in Section 4. In Section 5, we introduce the multi-version provenance model. We study reenactment in Section 6 and demonstrate how to implement reenactment as standard relational queries in Section 7. In Section 8, we discuss our implementation and optimization techniques for reenactment. We present experimental results in Section 9 and conclude in Section 10.

2 System Overview

In this section, we give an end-to-end overview of our approach for computing provenance for transactions.

2.1 Multi-version Provenance Model

Our first contribution is to introduce MV-semirings, a provenance model that extends the well-known semiring annotation framework [19] to account for tuple derivations under transactional semantics. For any semiring $\mathcal{K}$ we can construct an MV-semiring $\mathcal{K}^{\nu}$. An annotation in $\mathcal{K}^{\nu}$ is a symbolic expression over elements from $\mathcal{K}$ recording the derivation history of a tuple. These expressions use *version annotations* to enclose part of the provenance of a tuple which encodes that this part of the provenance was processed by a certain update at a certain time. There is an intuitive correspondence between these version annotations and tuple versions: each version annotation wrapping the provenance of a tuple corresponds to the creation of a new tuple version. We define update operations and a *snapshot isolation* (SI) transactional semantics for this model. In the resulting semantics, each tuple in a version of a database produced by a history of transactions is annotated with its complete derivation history according to these transactions. Our model also supports provenance tracking for queries, i.e., the provenance a query result can not just be traced back to the inputs of the query, but also reaches back into the transactional history that produced these inputs. The model preserves a major advantage of the semiring framework: it generalizes standard set and bag semantics as well as types of annotated relations such as incomplete database. That is, we can derive a standard bag semantics database for a given snapshot isolation history from the annotated database for this history.

[3] Name omitted due to licensing restrictions.

Example 2.1 *Consider the annotations on the left of each tuple in Figure 1 showing the provenance for each tuple version according to our model. For now, we will only explain the meaning of these annotations - how to compute them will be covered later. As mentioned above, the provenance of a tuple version encodes its whole derivation history - from which tuple version was that tuple version derived and by which operations. As an extension of the semiring annotation framework, our model uses variables to denote tuples in the provenance. We wrap parts of a tuple's provenance in version annotations to denote that it was produced by a certain type of update of transaction T at time ν.*[4]

Transaction T_1. *For instance, consider the annotations on the* `Order` *relation tuples in Figure 1a. The first tuple was produced by an insert (I) of transaction T_1 executed at time (version) 1. Note that we assign a time stamp $\nu+1$ to tuples created by an update executed at time ν. We assign a fresh variable (x_1 in the example) to tuples created by an insert using a* `VALUES` *clause. Inserted tuples are assigned new tuple ids (id 1 shown as a superscript in the version annotation). When the transaction creating a tuple version commits then we wrap this tuple version in a commit version annotation (C). The resulting provenance expression for this tuple is $C^1_{T_1,6}(I^1_{T_1,2}(x_1))$. Transaction T_1 has also inserted two tuples into relation* `Outstanding`*. These tuples are the result of running queries over relation* `Order`*. We record which tuples of relation* `Order` *each of the new tuples in relation* `Outstanding` *depends on by wrapping the provenance of these* `Order` *tuples in a version annotation. For instance, the first tuple of relation* `Outstanding` *is annotated with $C^3_{T_1,6}(I^3_{T_1,3}(I^1_{T_1,2}(x_1)))$, i.e., it was produced by an insert of transaction T_1 executed at time 2 which used a tuple (represented by variable x_1) produced by the same transaction at time 1.*

[4] We assume that versions in the database are identified by values from a discrete time domain.

Based on the provenance annotations in Figure 1d, the service operator from our running example can explain the three tuples in the `Collection` relation. For instance, it is clear that the update to the `Order` tuple was not reflected in the corresponding `Outstanding` relation tuple (the provenance contains the annotation of o_1 before the update) which was used to create the first tuple of the `Collection` relation.

2.2 Provenance Filtering

For databases with long histories, a user is likely not interested in the complete derivation history of currently valid tuples. Thus, we need an approach for abstracting away unnecessary details in our provenance model. Since our model is composable we can remove irrelevant parts of a tuple's derivation history by replacing provenance subexpressions with new variables. We demonstrate how this approach can be applied to limit the provenance to a given transaction.

Example 2.2 *Assume the service operator wants to drill down into the provenance of Transaction T_3. That is, she is only interested in modifications of tuples by this transaction. This is naturally achieved in our model by replacing subexpressions in the provenance with variables that represent the input tuple versions as seen by transaction T_3. For instance, for tuple s'_1 in relation `Outstanding` (Figure 1c) we would replace version annotations from previous transactions (T_1 and T_2) in the annotation $C^3_{T_3,13}(U^3_{T_3,9}(C^3_{T_1,6}(I^3_{T_1,3}(I^1_{T_1,2}(x_1)))))$ with a plain variable x_3 wrapped in the commit annotation of T_1. The resulting annotation is $C^3_{T_2,13}(U^3_{T_3,9}(C^3_{T_1,6}(x_3)))$. Furthermore, we would remove the second tuple by setting its annotation to 0 (a 0-annotation denotes that the tuple is not in the relation), because this tuple was not affected by Transaction T_3. The resulting provenance only contains information about T_3's updates, e.g., the update of the first tuple of relation `Outstanding`.*

2.3 Reenactment

We next prove an important fact. If we extend our query model with a new operator that creates version annotations, then any update, transaction, or (partial) history in our model can be equivalently expressed as a query, e.g., from an update u we can derive a query $\mathbb{R}(u)$ which returns the same database state as the original update u (if executed over the same input). We call such queries *reenactment queries*. A history is a potentially concurrent execution of a set of transactions, e.g., Figure 2 shows a history. We will formally define histories in Section 5.3. Note that in this paper we are focusing on reenactment of updates or single transactions only. However, we allow these transaction to be part of a larger history. The equivalence under annotated semantics between an operation and its reenactment query has several important implications: instead of computing provenance eagerly during transaction execution we retroactively compute it by running reenactment queries. Furthermore, since our model generalizes bag-semantics snapshot isolation, we can use reenactment to recreate a database state valid at a particular time by simply running a query - including database states that were only visible within one transaction.

Example 2.3 *For simplicity we illustrate reenactment using SQL and standard relation query semantics. Our formal treatment of the subject in Section 6 uses an algebra of updates and queries defined for our provenance model. For instance, an SQL update* `UPDATE R SET a = a + 1 WHERE b = 3` *over a relation $R(a, b)$ can be reenacted as a query that runs over the database state valid before the update. Intuitively, we compute a union between the previous versions of tuples that were not affected by the update (tuples that do not fulfill the* `WHERE` *clause condition) and the new version of tuples that were updated (fulfill the* `WHERE` *clause condition). Thus, we can reenact the update shown above as:*

```
SELECT * FROM R WHERE b <> 3
UNION ALL
SELECT a + 1 AS a, b FROM R WHERE b = 3;
```

2.4 Relational Implementation using Time Travel

While our model fulfills our desiderata for a transaction provenance model, it would require major changes to implement it within a DBMS. Our third major contribution is a mapping of our provenance model to a standard relational representation and a method for using an *audit log* (a log of SQL statements executed on the database) to construct reenactment queries and *time travel* to access past database states when running reenactment queries. Thus, we can compute the provenance of past updates, transactions, and across transactions - without having to materialize any additional information, without modifying the DBMS, and without requiring changes to the transactional workload.

Example 2.4 *Figure 3 shows the relational representation of the provenance of relation `Collection` from the example history restricted to transaction T_3. The annotation of a tuple t is represented as several tuples by duplicating the tuple t and storing part of the annotation in additional attributes. Tuple variables are represented by actual tuples. Version annotations are represented as boolean attributes ($\mathcal{U}_i$ for update u_i) which*

	Collection		Provenance from Relation Outstanding			u_1	u_2
	order	amount	P(Outstanding,order)	P(Outstanding,amount)	P(Outstanding,due)	$\mathcal{U}_1$	$\mathcal{U}_2$
$C^5_{T_3,13}(I^5_{T_3,11}(U^3_{T_3,9}(x_3)))$	oid_1	50	oid_1	300	2000-05	T	T
$C^6_{T_3,13}(I^6_{T_3,11}(x_4))$	oid_3	56	oid_3	56	2000-05	F	T

Fig. 3: Relational encoding of the provenance of example transaction T_3

are true if this part of the provenance has this version annotation and false otherwise. In Figure 3 we show the annotation encoded by a tuple on the left of this tuple. The boolean attributes $\mathcal{U}_1$ and $\mathcal{U}_2$ represent the version annotations for the update (u_1) and insert (u_2) of Transaction T_3. For instance, consider the annotation $C^5_{T_3,13}(I^5_{T_3,11}(U^3_{T_3,9}(x_3)))$ of tuple t =(oid_1,50) derived by replacing $C^3_{T_1,6}(I^3_{T_1,3}(I^1_{T_1,2}(x_1)))$, the part of the annotation of t related to previous transactions, with a new variable x_3. This tuple was affected by both updates of T_3. Thus, both $\mathcal{U}_1$ and $\mathcal{U}_2$ are set to true. The tuple version t' =(oid_1,300,2000-05) from relation `Outstanding` from which tuple t was derived (represented by variable x_3 in the provenance annotation) is stored in additional attributes we add to the schema. Here P denotes a renaming function used to distinguish attributes storing provenance from attributes storing data.

Note that while the number of attributes in the relational encoding depends on how far back provenance is traced this representation is not materialized but constructed on the fly using reenactment when a user requests provenance. Using actual tuple values to represent variables in provenance expressions is often more meaningful to a user than other representations of the variables such as pairing tuple identifies with versions. Nonetheless, we let the user decide how tuples are represented (actual values, tuple id and timestamp, or both).

2.5 GProM

We have implemented the techniques discussed above in our **GProM** [3] provenance middleware. GProM works as a wrapper of a standard relational database. The user interacts with the system using the SQL dialect of the underlying DBMS. We support several extensions for computing provenance which are seamlessly integrated within SQL. For example, the user can request the provenance of a query, update, or single transaction, or for a certain time interval. To process a transaction provenance request we 1) query the audit log to gather sufficient information to be able to construct a reenactment query and 2) translate the reenactment query into an SQL query with time travel (i.e., querying the transaction time history of tables) which returns our relational encoding of provenance (e.g., as shown in Figure 3). From a language point of view, a provenance request is treated as a query that returns a relational provenance encoding, e.g., it can be used as a subquery (to query provenance). For instance, to return all tuples affected by an update u of a transaction T, the user would request the provenance of T and only keep tuples for which the annotation attribute of update u is true. To track the derivation history of a single tuple the user requests the provenance of the relation containing the tuple and applies a selection to the result to return only the provenance for the tuple she is interested in.

3 Related Work

3.1 Provenance Models

Provenance of relational queries has been studied extensively in the recent years leading to the development of several models including Why-provenance [9], Where-provenance [9], and Lineage [12]. See [11] for an overview. The seminal paper from Green et al. [19] introduced the $\mathcal{K}$-relational model, an extension of the relational model with annotations from a commutative semiring and has shown how such annotations propagate through positive relational algebra ($\mathcal{RA}^+$) queries. The semiring of provenance polynomials is the most general form of annotation in this model. Provenance polynomials generalize the relational datamodel (set and bag semantics), several extensions (e.g., trust), and less informative provenance models including Lineage and Why-provenance. See [21] for an overview of this model and its extensions beyond positive relational algebra (e.g., set difference [14] and aggregation [2]). Kostylev et al. [24] have studied data annotated with annotations from multiple semirings. Buneman et al. [10] relax the semiring model for a hierarchical data model where the distinction between data and annotation is flexible - allowing queries to treat part of a hierarchy as annotations and others as data. Oltenau et al. [25] discuss factorization of provenance polynomials and Amsterdamer et al. [1] rewrite queries into equivalent queries (under set semantics) with minimal provenance. Boolean Circuits can be used to compactly represent semiring expressions [13]. It has been proven that provenance polynomials can be extracted from the PI-CS [17] and Provenance Games [23] models. The latter also addresses negation. We extend the semiring framework with updates and transactional semantics. The idea of annotating parts of a provenance polynomial with function symbols was, to the best of our knowledge, first applied in the context of the Orchestra system to record applications of schema mappings [20]. The version annotations in our model were inspired by this idea. The major advances we made in developing our extension are 1) encode derivation under concurrent transactions and 2) model the visibility rules of the snapshot isolation concurrency control protocol. Our model is a strict generalization of the semiring model in the sense that we can derive the semiring annotations of a tuple from our model. As we will discuss further in Section 5, naive combinations of the semiring model with implementations of snapshot isolation which use additional attributes to store version information have the disadvantage that a tuple's provenance may be spread over multiple relations and database versions whereas in our model it is stored in the tuple's annotation.

3.2 Systems and Provenance for Past Operations

Systems such as DBNotes [6], Orchestra [22], LogicBlox [21], and Perm [17] encode provenance annotations as standard relations and use query rewrite techniques to propagate these annotations during query processing. We also implement provenance computation for transactions by propagating a relational encoding of provenance annotations. Similar to the Perm system, we refrain from eagerly computing provenance for all operations, but instead reconstruct provenance when requested. Zhang et al. [28] demonstrated that an audit log and time travel functionality is sufficient for computing the provenance of past queries. In this work, we prove that audit logging and time travel are also sufficient for computing the provenance of transactions. This idea of using a log of operations (and changes to data) to reconstruct provenance by replaying operations has also been applied in the DistTape system [29] (distributed datalog) and the Ariadne system [16] (stream processing). Such replay techniques could be applied to replay SI transactional histories as long as the replay mechanism implements snapshot isolation (or alternatively enforces the visibility rules of snapshot isolation) and ensures that the operations of transactions are executed in the same order as in the original history. The novelty of our reenactment mechanism lies in the fact that instead of replaying updates we construct a reenactment query that simulates the updates. The execution order of operations in the history is "hard-coded" into that query. Thus, we do not have to pay the overhead of DB update operations (caused by logging, concurrency control, and I/O of writing changes to disk) and can apply optimizations such as reordering updates and pushing selections through updates that are not available to a DBMS if the system replays updates one at a time.

3.3 Provenance for Updates

Provenance for updates has been studied in related work [22, 8, 27], but none of these approaches addresses the complications that arise when updates are run as parts of concurrent transactions. Note that the "transactions" from Archer et al. [4] are sequences of updates and not concurrent transactions. Buneman et al. [7] present a copy-based model of provenance for curated databases where sequences of updates are grouped into transactions to reduce the size of provenance at the cost of lossing information about intermediate states produced by updates. This work also did not consider concurrent transactions. Buneman et al. [8] have studied a copy-based provenance type for the nested update language and nested relational calculus. Vansummeren et al. [27] define provenance for SQL DML statements by modifying the updates to store provenance. Our approach differs in that we reconstruct provenance on demand instead of computing and storing provenance for all operations. Furthermore, we are the first to compute transactional provenance (for the snapshot isolation [5] concurrency control protocol) using a novel technique for query-based replay (reenactment). Extending approaches for updates to support transactions is nontrivial, because it requires tracking provenance through multiple operations taking the visibility of tuple versions into account (some of which only exist temporarily during the execution of a transaction).

4 Background

In this section we introduce necessary background on concurrency control and semiring annotated data.

Snapshot Isolation. Under *Snapshot isolation* (*SI*) [5] 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. 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 when it begins that determines which version of the database it will see (its snapshot). Concurrent writes are allowed under SI. However, if several concurrent transactions write the same data item d, only one will be allowed to commit. Under the *First Committer Wins (FCW)* rule, the transaction which tries to commit first is allowed to commit. Under the *First Updater Wins (FUW)* rule, the first transaction updating d is allowed to commit. SI corresponds to isolation level `SERIALIZE` in systems such as Oracle and older versions of PostgreSQL. These implementations neither apply the *FCW* nor the *FUW* rule, but instead use 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).

The Semiring-Annotation Framework. Green et al. [19,21] have introduced the semiring annotation framework. In this framework [21] relations are annotated with elements from a commutative semiring $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$. Such relations are called $\mathcal{K}$-relations. Formally, 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. The operators of the positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations are defined by applying the $+_{\mathcal{K}}$ and $\times_{\mathcal{K}}$ operations of the semiring to input annotations. $\mathcal{K}$-relations generalize extensions of the relational model including bag semantics, incomplete databases, and various provenance models (e.g., Lineage). Intuitively, the $+_{\mathcal{K}}$ and $\times_{\mathcal{K}}$ operations of the semiring correspond to alternative and conjunctive use of tuples. For instance, if an output tuple t was produced by joining input tuples annotated with k and k', then the tuple t would be annotated with $k \times_{\mathcal{K}} k'$. Provenance polynomials (semiring $\mathbb{N}[X]$), polynomials over a set of variables X which represent tuples in the database, are the most general form of semiring annotation. Using $\mathbb{N}[X]$, every tuple in an instance is annotated with a unique variable $x \in X$. This semiring $\mathbb{N}[X]$ has the important property that for any semiring $\mathcal{K}$ the annotation of a query result t in $\mathcal{K}$ can be derived from the provenance polynomial for t. This is done by mapping each variable $x \in X$ to an element from $\mathcal{K}$ and interpreting the abstract $+$ and $\times$ operations in $\mathbb{N}[X]$ as the corresponding operations in $\mathcal{K}$. Formally, any valuation $\chi : X \to K$ of variables to elements from a semiring $\mathcal{K}$ can be lifted to a semiring homomorphism $Eval_{\chi} : \mathbb{N}[X] \to \mathcal{K}$. Semiring homomorphisms commute with queries. The table below shows some semirings and the extensions of the relational model they encode.

Semiring	Corresponding Model
$(\mathbb{B}, \vee, \wedge, false, true)$	Set semantics
$(\mathbb{N}, +, \times, 0, 1)$	Bag semantics
$(\mathcal{P}(X) \cup \{\perp\}, \cup_{+}, \cup_{\times}, \perp, \emptyset)$	Lineage
$(\mathbb{N}[X], +, \times, 0, 1)$	Provenance polynomials

The semiring $\mathbb{B}$ with elements *true* and *false* using $\vee$ as addition and $\wedge$ as multiplication corresponds to set semantics. The semiring $\mathbb{N}$, the set of natural numbers with standard arithmetics corresponds to bag semantics. In the Lineage provenance model, the provenance of a result tuple t of a query is a set of tuples from the input that were used to derive t. The semiring over the powerset of tuples in an instance (represented as variables X) using set union for addition and multiplication corresponds to Lineage [11].[5]

Example 4.1 *Consider the $\mathbb{N}[X]$-relation R_f shown below and the result of evaluating the query $Q = \Pi_A(R_f \bowtie \rho_{B,C}(R_f))$ (persons that have friends with friends) over this relation. The provenance polynomial for the query result tuple records that this tuple was produced by joining x_1 with x_2 ($x_1 \times x_2$) and by joining x_1 with x_3 ($x_1 \times x_3$). By mapping x_1, x_2, and x_3 to true and interpreting $+$ as $\vee$ and $\times$ as $\wedge$ we get a $\mathbb{B}$-annotation* true *indicating that the result tuple exists under set semantics. By mapping x_1 to x_3 to $1 \in \mathbb{N}$ and evaluating the resulting expression we get $1 \times 1 + 1 \times 1 = 2$, the multiplicity of the tuple under bag semantics. Finally, by mapping x_i to $\{x_i\}$ for $i \in \{1, 2, 3\}$, and by interpreting the expression in the lineage semiring we get $\{x_1, x_2, x_3\}$, the Lineage of the result.*

R_f

	A	B
x_1	*Pete*	*Bob*
x_2	*Bob*	*Alice*
x_3	*Bob*	*Gert*

Result

	A
$x_1 \times x_2 + x_1 \times x_3$	*Pete*

5 Multi-Version Provenance Model

We need a provenance model which is powerful enough to provide a full account of how tuple versions have been derived from other tuple versions and through updates in an SI history. A typical way of implementing SI (and transaction time databases in general) is to store multiple versions of each tuple in a relation and use additional attributes which are hidden from the user to store a unique tuple identifier, the time interval during which the tuple version was valid, and potentially the transaction which created the tuple version. Each update of a tuple creates a new tuple version with the same tuple identifier, the start time set to the current

[5] $\perp$ means not in the database and $\emptyset$ means no provenance. $\cup_{+}$ and $\cup_{\times}$ are both set union except for $\perp$ where these operations are defined as $k \cup_{+} \perp = k$ and $k \cup_{\times} \perp = \perp$.

time, the end time set to UC (until changed), and the transaction identifier set to the transaction updating the tuple. Such an update would also set the end time of the previous version of this tuple to the current time.

It is tempting to extend such a representation of snapshots with semiring annotations to represent provenance for snapshot isolation histories. However, we will demonstrate in the following that this approach has two major drawbacks: 1) the derivation history of a tuple is not fully contained in the tuple's annotation in this representation. In fact, tracing the origins of a tuple requires correlating annotations from multiple tuple versions - possibly across relations and several versions of the database; 2) even if we combine information from multiple tuple versions it may not be possible to reconstruct a tuple's complete derivation history.

Example 5.1 *As an example of the first problem consider how the instance of Figure 1a would be represented using $\mathcal{K}$-relations and a typical implementation of snapshot isolation using three additional attributes: the identifier of the transaction that created the tuple version (XID), the version when the tuple started to be valid (T_b), and the version when this tuple version is no longer valid (T_e). We show this instance below. Attribute T_b of tuple t is set to the commit time of the transaction that produced tuple t. Using this technique to store snapshot relations, the tuple versions visible to an update within a transaction T include all tuples committed before T started that were still valid when T started ($T_b \leq Start(T) < T_e$) plus all of T's own changes ($XID = T$).*

Order

	id	*customer*	*price*	*XID*	T_b	T_e
x_1	*oid*$_1$	*Peter*	*300*	T_1	*5*	*UC*
x_2	*oid*$_3$	*Peter*	*56*	T_1	*5*	*UC*

Outstanding

	order	*amount*	*due*	*XID*	T_b	T_e
x_1	*oid*$_1$	*300*	*2000-05*	T_1	*5*	*UC*
x_2	*oid*$_3$	*56*	*2000-05*	T_1	*5*	*UC*

The first difference of this representation to the instance annotated using our model is that the annotations of the tuples in relation `Outstanding` do not contain the whole provenance of such a tuple - part of its provenance is stored in a tuple from the `Order` relation. Thus, reconstructing the complete derivation history of a tuple requires correlating provenance across multiple tuples.

Compared to the instance in Figure 1a, we have lost information of how tuples have been derived, e.g., although we can infer that the two tuples annotated with x_1 are somewhat related, we do not know how. All we know is that they were both produced by Transaction T_1 and started to be valid at time 5 (the time when T_1 committed). Extending the model by adding additional attributes such as tuple identifiers and identifiers for the update operation creating a tuple would solve this problem for this particular example. However, this not true in the general case. Consider a relation $R(A,B,C) : \{(1,2,3) \rightarrow x\}$ (here we denote a tuple t annotated with k as $t \rightarrow k$) and an insert `INSERT INTO S (SELECT A,C FROM R UNION SELECT B,C FROM R)`. *This creates the following instance $S(A,C) : \{(1,3) \rightarrow x, (2,3) \rightarrow x\}$. The same transaction then executes an insert* `INSERT INTO T (SELECT C FROM S WHERE f(A))` *where function f's return type is boolean. Let us assume that $f(1) = true$ and $f(2) = false$. The new tuple $t_{new} = (3)$ inserted into table T will be annotated with x. Based on this annotation it is impossible to know whether this tuple was derived from tuple $(1,3)$ or $(2,3)$. Additional information that we can extract from the temporal attributes of the snapshot isolation implementation is not useful for resolving this ambiguity.*

These examples illustrate the need for a provenance model that can help us track the origin of tuple versions. We have developed an extension of the semiring model that fulfills this requirement. Given a semiring $\mathcal{K}$ we construct a new semiring $\mathcal{K}^\nu$ that represents $\mathcal{K}$ with embedded history. We call structures constructed in this fashion multi-version (MV) semirings. The elements of such a semiring are symbolic expressions over elements from K, version annotations, and semiring operations where the structure of an expression encodes the derivation history of a tuple. Recall from Example 1.1 that version annotations wrap a part of the provenance to indicate that a version of a tuple with identifier *id* (with the wrapped provenance) was modified by a certain type of update operation (**I**nsert, **U**pdate, or **D**elete), executed as part of a transaction T, at time $\nu - 1$. Furthermore, we use a version annotation C to denote that the transaction T creating a tuple version committed at time $\nu - 1$ and, thus, the tuple version will be visible to transaction starting at or after ν. The symbolic expressions that are the elements of an MV-semiring are uninterpreted with the exception of a set of equivalence relations which ensures that $\mathcal{K}^\nu$ obeys the laws of commutative semirings and addition as well as multiplication with operands from K can be evaluated.

Definition 5.1 *Let $\mathbb{T}$ be a domain of transaction identifiers, $\mathbb{V}$ a domain of version identifiers, $\mathbb{I}$ a domain of tuple identifier, and $\mathcal{K} = (K, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ a commutative semiring. The set $\mathbb{A}$ of version annotations contains the following elements for each transaction $T \in \mathbb{T}$, version $\nu \in \mathbb{V}$, and tuple identifier $id \in \mathbb{I}$: $I^{id}_{T,\nu}, U^{id}_{T,\nu}, D^{id}_{T,\nu}, C^{id}_{T,\nu}$. Consider the set of finite symbolic expressions P defined by the grammar shown below where $k \in K$ and $\mathcal{A} \in \mathbb{A}$. $P := k \mid P + P \mid P \times P \mid$*

Laws of commutative semirings

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

Evaluation of expressions with operands from K

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

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 $\mathcal{K}^{\nu}$

$\mathcal{A}(P)$. Furthermore, let K^{ν} be the set of congruence classes for expressions in P based on the equivalence relations shown in Figure 4. We use $[k]_{\sim}$ to denote the congruence class of $k \in P$. The multiversion semiring (MV-semiring) for semiring $\mathcal{K}$ is the structure $\mathcal{K}^{\nu} = (K^{\nu}, +_{\mathcal{K}^{\nu}}, \times_{\mathcal{K}^{\nu}}, [0_{\mathcal{K}}]_{\sim}, [1_{\mathcal{K}}]_{\sim})$. Here $\times_{\mathcal{K}^{\nu}}$ is defined as $[k]_{\sim} \times_{\mathcal{K}^{\nu}} [k']_{\sim} = [k \times k']_{\sim}$. Operation $+_{\mathcal{K}^{\nu}}$ is defined analogously.

Note that the structure $\mathcal{K}^{\nu}$ is a semiring. The elements of this structure are expressions build from version annotations, elements from K, and the operations $+$ and $\times$. In such expressions we are allowed to evaluate products and sums that only combine elements from K, but not allowed to interpret version annotations except for applying the equivalences used in the construction. 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 tuple 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 $\mathcal{K}$. The intuitive meaning of the equivalence for version annotations are: 1) update operations never create tuples from non-existing or deleted tuples (recall that if a tuple is annotated with $0_{\mathcal{K}}$ in relation R this denotes that the tuple is not in the relation R) and 2) alternative use of tuples distributes over updates (e.g., updating the result of a union query returns the same result as computing the union after updating its inputs). In the following we will omit the subscript of operations and neutral elements if the semiring is clear from the context or irrelevant to the discussion. Since we typically define a single semiring structure for a given set K, we will sometimes use $\mathcal{K}$ to refer both to the semiring $\mathcal{K}$ and its set K interchangeably.

Definition 5.2 *Let $\mathbb{D}$ be a universal domain of values and $\mathcal{K}$ a semiring. An n-nary $\mathcal{K}$-relation R is a function: $\mathbb{D}^n \rightarrow K$ that maps each tuple $t \in \mathbb{D}^n$ to an annotation from K. We require that R has finite support (number of tuples not mapped to 0). A $\mathcal{K}$-database is a set of $\mathcal{K}$-relations.*

There exists a strong connection between $\mathcal{K}$ and $\mathcal{K}^{\nu}$ relations: By evaluating the symbolic expression that make up an $\mathcal{K}^{\nu}$ element interpreting version annotations as functions $K \rightarrow K$, we transform an $\mathcal{K}^{\nu}$ relation into a corresponding $\mathcal{K}$ relation. Conceptually, this means we are removing the embedded history from the provenance. For example, if we apply this approach to derive provenance polynomials from their $\mathcal{K}^{\nu}$ counterpart, the result will record from which tuples a tuple was derived (and how), but no longer encode its update history. Below we define an operator UNV that implements this mapping based on a function $h_U : \mathcal{K}^{\nu} \rightarrow \mathcal{K}$. In Section 5.2 we will prove that h_U is a semiring homomorphism which as proven by Greene et al. [19] implies that it commutes with queries.

Definition 5.3 *Let R be a $\mathcal{K}^{\nu}$-relation. The unversioning operation* UNV(R)*: $\mathcal{K}^{\nu}$-relation $\rightarrow$ $\mathcal{K}$-relation applies the mapping $h_U : \mathcal{K}^{\nu} \rightarrow \mathcal{K}$ defined below to every tuple's annotation, i.e.,* UNV$(R)(t) = h_U(R(t))$.

$$h_U(k) = \begin{cases} k & \text{if } k \in K \\ h_U(k') & \text{if } k = I^{id}_{T,\nu}(k') / U^{id}_{T,\nu}(k') / C^{id}_{T,\nu}(k') \\ 0_{\mathcal{K}} & \text{if } k = D^{id}_{T,\nu}(k') \\ h_U(k_1) +_{\mathcal{K}} h_U(k_2) & \text{if } k = k_1 + k_2 \\ h_U(k_1) \times_{\mathcal{K}} h_U(k_2) & \text{if } k = k_1 \times k_2 \end{cases}$$

Note that we use $k = I^{id}_{T,\nu}(k') / U^{id}_{T,\nu}(k') / C^{id}_{T,\nu}(k')$ as a notational shortcut for $k = I^{id}_{T,\nu}(k') \vee k = U^{id}_{T,\nu}(k') \vee k = C^{id}_{T,\nu}(k')$ and will use similar notation throughout the paper, e.g., I/U denotes a version annotation that is either an insert or update. The application of UNV to an $\mathcal{K}^{\nu}$-database D is defined in the obvious way.

Example 5.2 *Reconsider the instance of relation* **`Outstanding`** *from the example shown in Figure 1d. This instance is annotated with $\mathbb{N}[X]^{\nu}$, the MV version of the provenance polynomial semiring. The first tuple s'_1 is annotated with $C^{3}_{T_3,13}(U^{3}_{T_3,9}(C^{3}_{T_1,6}(I^{3}_{T_1,3}(I^{1}_{T_1,2}(x_1)))))$, i.e., it was created by an update of Transaction T_3, that updated a tuple inserted by T_1 based on another previously inserted tuple by the same transaction. Based on the outermost commit annotation we know that this tuple version is visible to transactions starting after version 12. The second tuple s'_2 is annotated with $C^{4}_{T_4,14}(D^{4}_{T_4,10}(C^{4}_{T_1,6}(I^{4}_{T_1,5}(I^{2}_{T_1,4}(x_2)))))$, i.e., this tuple was deleted by Transaction T_4 (and was originally produced by* *a sequence of two inserts by Transaction T_1). If we apply* Unv *to relation* `Outstanding`*, then s'_1 is annotated with x_1 and s'_2 is annotated with* 0 *(indicating that the deleted tuple s'_2 is not in the instance).*

In the following we make use of a normal form for $\mathcal{K}^\nu$ elements that represents them as a sum of subexpressions which use multiplication and version annotations. This will simplify the definition of updates and transactional semantics in our model.

Definition 5.4 *An $\mathcal{K}^\nu$ element k is normalized if it is of the form: $\sum_{i=0}^{m} k_i$ where 1) none of the summands k_i contains addition and 2) all summands are non-zero.*

Note that any annotation k can be translated into this normal form by applying the equational laws of MV-semirings. For example, an annotation $I^3_{T,\nu_2}(U^2_{T,\nu_1}(x_1)+U^1_{T,\nu_1}(x_2))$ can be normalized based on distributivity of addition over version annotations into $I^3_{T,\nu_2}(U^2_{T,\nu_1}(x_1))+I^3_{T,\nu_2}(U^1_{T,\nu_1}(x_2))$. In the following, it will be helpful to introduce notation for accessing particular elements from the sum of a normalized $\mathcal{K}^\nu$ element. We use $n(k)$ to denote the number of summands of a normalized $\mathcal{K}^\nu$-element k and $k[i]$ to denote the ith element in the sum (assuming some order over the summands).

5.1 Queries

We use the standard definition of positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations with the exception that we add one operator $\{t \to k\}$ that creates a singleton relation containing the tuple t annotated with k. Note that this is an extension of the empty relation operator introduced in the original work on $\mathcal{K}$-relations [19]. For sake of completeness, we repeat the full definition of $\mathcal{RA}^+$ here. We use $t.A$ to 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 5.5 *Let $\mathcal{K}$ be a semiring, R, S denote $\mathcal{K}$-relations,* Sch(R) *denote the schema of relation R, t, u denote tuples, and $k \in K$. The positive relational algebra $\mathcal{RA}^+$ on $\mathcal{K}$-relations is defined as:*

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

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

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

Note that the singleton construction $\{t \to k\}$ introduced above does not affect the commutativity of semiring homomorphisms with queries. However, since this operator explicitly mentions a semiring element $k \in \mathcal{K}$, a homomorphism $h : \mathcal{K} \to \mathcal{K}'$ has to be applied to the query too to guarantee that it returns a $\mathcal{K}'$ relation (this is similar to the treatment of the constant annotation operator in [14]). Let $h(Q)$ denote the application of the homomorphism h to query Q, i.e., we replace every operator $\{t \to k\}$ in Q with $\{t \to h(k)\}$.

Theorem 5.1 *Let $h : \mathcal{K} \to \mathcal{K}'$ be a semiring homomorphism, then h commutes with any Q in the above algebra if h is applied to Q. Let I be a $\mathcal{K}$ database instance. Then, $h(Q)(h(I)) = h(Q(I))$*

Proof The proofs to all theorems presented in this report are given Appendix

The mapping $h_U : \mathcal{K}^\nu \to \mathcal{K}$ used in the definition of the Unv operator introduced above is a semiring homomorphism. Thus, the application of Unv commutes with queries. Practically, this means we can execute queries over relations with embedded history and then derive the corresponding relation without history or equivalently strip the history information upfront.

Theorem 5.2 *h_U is a surjective semiring homomorphism.*

Consider $\mathbb{N}[X]^\nu$, i.e., the MV-semiring version of the provenance polynomials semiring $\mathbb{N}[X]$. A variation of the fundamental property of the semiring framework still holds for $\mathcal{K}^\nu$-relations. That is, $\mathbb{N}[X]^\nu$ generalizes all other $\mathcal{K}^\nu$ semirings if we consider mappings that preserve embedded history. Any assignment $\chi : X \to K$ of elements from K to each variable from X extends to a homomorphism from $Eval_\chi{}^\nu : \mathbb{N}[X]^\nu \to \mathcal{K}^\nu$. Practically, this means that we can use the result of a query in $\mathbb{N}[X]^\nu$ to derive the query result in any MV-semiring $\mathcal{K}^\nu$ (and, thus also semiring $\mathcal{K}$ by applying Unv). In fact, we prove a more general result: any homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism $h^\nu : \mathcal{K}_1{}^\nu \to \mathcal{K}_2{}^\nu$ by applying h to each element from K_1 in an expression in $K_1{}^\nu$. We call this type of homomorphisms *history-preserving* because they do not change the embedded history (structure of the symbolic expression) of an MV-semiring element.

Theorem 5.3 *Any semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism $h^\nu : \mathcal{K}_1{}^\nu \to \mathcal{K}_2{}^\nu$ as defined below. If h is surjective then so is h^ν.*

$$h^\nu(k) = \begin{cases} h(k) & \textit{if } k \in K_1 \\ \mathcal{A}(h^\nu(k')) & \textit{if } k = \mathcal{A}(k') \\ h^\nu(k_1) + h^\nu(k_2) & \textit{if } k = k_1 + k_2 \\ h^\nu(k_1) \times h^\nu(k_2) & \textit{if } k = k_1 \times k_2 \end{cases}$$

Example 5.3 *Consider a query $Q = \Pi_{customer}(Order)$ run over the instance from Figure 1b. This query returns a single tuple t =(Peter) as shown below. The annotation of this tuple records that t was produced from two tuples in the input of the query and how these two tuples were created (e.g., $C^2_{T_1,6}(I^2_{T_1,4}(x_2))$). To compute the answer to this query under bag semantics we first apply the* Unv *operator which returns annotation $x_1 + x_2$ for tuple t and then apply an assignment $\mathbb{N}[X] \to \mathbb{N}$. If we assume that both input tuples have multiplicity 1, then tuple t will be annotated with $1 + 1 = 2$, the multiplicity of this query result under bag semantics.*

$$C^1_{T_2,8}(U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1)))) \\ + C^2_{T_1,6}(I^2_{T_1,4}(x_2))$$

customer
Peter

5.2 Update Operations

So far in our treatment of MV-relations we have considered instances annotated with arbitrary elements from a semiring $\mathcal{K}^\nu$, i.e., elements that can be constructed using the grammar shown in Definition 5.1. However, not every such element can be the result of a sequence of update operations or transactional history. For example, $k = U^1_{T,\nu+1}(C^1_{T,\nu}(x))$ is a valid $\mathbb{N}[X]^\nu$ element. Nonetheless, k cannot occur in a database created by a valid history because according to k, Transaction T did update a tuple after its commit. We define *admissible* MV-relations (databases) as instances that can be created from an empty relation (database) by a sequence of updates or a transactional history in our model.

Definition 5.6 *An $\mathcal{K}^\nu$-relation R is called admissible if there exists a (potentially empty) sequence of update operations (as defined in Definition 5.7) that if applied to an empty input produces R or if there exists a history H such that R is a relation in the database state produced by H (as defined in Definition 5.9).*

We now define an update language for $\mathcal{K}^\nu$-relations. We restrict the application of update operations to admissible instances to ensure that the input of the update has sufficient history embedded to correctly evaluate the update. In particular, summands in a normalized annotation in an admissible instance are of the form $\mathcal{A}(k)$ and we will use this fact in the definition of update operations. In contrast to queries which do not manipulate version annotations in $\mathcal{K}^\nu$ expressions, update operations add new versions annotations, i.e., they extend the history embedded in an $\mathcal{K}^\nu$ annotation to record the application of the update. We introduce three update operations and a commit operation for our model. For each operation, we consider it to be executed at a time ν as part of a transaction T. Update operations take as input a normalized $\mathcal{K}^\nu$-relation R and return the updated version of this $\mathcal{K}^\nu$-relation (recall that any $\mathcal{K}^\nu$-relation can be brought into normal form). An insert $\mathcal{I}[Q, T, \nu](R)$ inserts the result of query Q into relation R. Note that this operation can express SQL style `INSERT ... VALUES (...)` (singleton operator) and `INSERT ... (SELECT ...)` statements. Newly inserted tuples are wrapped in version annotations and are assigned a fresh tuple id (i_{new}). Note that we do not allow inserts to "forge" history. That is, if the query Q of an insert contains a singleton operator $\{t \to k\}$ then k should be an element of the embedded semiring and not contain any version annotations. An update operation $\mathcal{U}[\theta, A, T, \nu](R)$ modifies each tuple in R that matches condition θ by applying the projection expressions in A. These tuples will be wrapped in version annotations. A deletion $\mathcal{D}[\theta, T, \nu](R)$ wraps all tuples matching the condition θ in a delete annotation. Recall that Unv interprets delete annotations as functions mapping every input to 0. Thus, deleted tuples are removed when R is mapped to the corresponding $\mathcal{K}$-relation. A commit operation $\mathcal{C}[T, \nu](R)$ wraps every input affected by transaction T into a commit annotation.

Definition 5.7 *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 i_{new} to denote a fresh id that is deterministically created as discussed below. Let Q be a query 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:*

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

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

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

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

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

Note that for updates we will often only explicitly state the projection expressions for attributes that are updated and assume that other attributes are kept unmodified. For instance, for an update over a relation $R(a,b,c)$ we may write $(b+5 \rightarrow b)$ instead of $(a, b+5 \rightarrow b, c)$. What tuple identifiers are assigned by inserts to new tuples is immaterial to our approach. However, identifiers should be assigned deterministically (to ensure that they can be recreated during reenactment as will be explained in Section 6) and should be "unique enough" to enable elements in the provenance to be distinguished (as illustrated in Example 5.1). We use a skolem function $f_{id}(T, \nu, t, k)$ to assign new ids id_{new} that takes as input the transaction T, version ν, tuple t to be annotated, and $\mathcal{K}^{\nu}$-element k that is wrapped in the version annotation.

Example 5.4 *Consider the update operation of Transaction T_2 from the running example. This update runs over the version of relation* `Order` *shown in Figure 1a. We abbreviate the attributes of this relation as d (id), c (customer), and p (price). This update operation can be expressed in our model as:*

$$\mathcal{U}[d=1, p \rightarrow 250, T_2, 6](Order)$$

Tuple o_1: Tuple o_1 in the instance of relation `Order` *is annotated with $C^1_{T_1,6}(I^1_{T_1,2}(x_1))$. This tuple fulfills the condition $d=1$ of the update and, thus, the first expression $(R(t) \times (\neg\theta)(t))$ in the annotation created by the update evaluates to: $Order(o_1) \times (\neg(d=1))(o_1) = C^1_{T_1,6}(I^1_{T_1,2}(x_1)) \times 0 = 0$. Note that here we use concrete tuple identifiers instead of the f_{id} scheme. The second part of the expression sums the annotations over all tuples u such that if the update is applied to them the resulting updated tuples are equal to o_1'. Since the update sets attribute price to a constant value, these are all tuples (oid_1,Peter,p') for some price p'. However, all tuples except for o_1 and o_2 are annotated with 0 in the input (they are not part of this instance). For tuples u with $Order(u) = 0$ the inner sum evaluates to $U^1_{T_2,7}(Order(u)) \times \theta(u) = U^1_{T_2,7}(0) \times \theta(u) = 0$. Intuitively, this is the expected result, because an update is only creating new versions of existing tuples. Tuple o_2 is annotated $C^2_{T_1,6}(I^2_{T_1,4}(x_2))$ in the input, a single element sum. Since, this tuple does not fulfill the update's condition, the inner sum evaluates to $U^2_{T_2,7}(C^2_{T_1,6}(I^2_{T_1,4}(x_2))) \times (d=1)(o_2) = U^2_{T_2,7}(C^2_{T_1,6}(I^2_{T_1,4}(x_2))) \times 0 = 0$. Finally, o_1 is the only tuple which fulfills the condition of the update and is not annotated with 0 in the input. For o_1 the inner sum evaluates to $U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1))) \times (d=1)(o_1) = U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1)))$. As expected the annotation denotes that the resulting tuple was derived from tuple o_1 in the previous version of relation* `Order` *and was not affected by any other input tuple.*

Notably, the fundamental property of $\mathbb{N}[X]^{\nu}$, the MV-semiring of provenance polynomials, extends to updates. Recall that any valuation $\chi : X \rightarrow K$ can be lifted to a history-preserving homomorphism ${Eval_{\chi}}^{\nu}$ and the following theorem states that such lifted homomorphisms commute with updates. Note that the identifier generation scheme we have introduced for inserts uses an element k of $\mathcal{K}^{\nu}$ as one argument of the skolem function f_{id}. We extend lifted homomorphisms to also manipulate the arguments of this skolem function to ensure that they commute with updates. In particular $h^{\nu}(f_{id}(T, \nu, t, k)) = f_{id}(T, \nu, t, h^{\nu}(k))$.

Theorem 5.4 *Let h^{ν} be a lifted homomorphism as defined in Theorem 5.3. h^{ν} commutes with updates.*

5.3 Transactions and Histories

We now define transactional histories for $\mathcal{K}^{\nu}$-databases under SI. We will limit the discussion to histories that start from an empty database. The results naturally extend to histories that are applied to any admissible $\mathcal{K}^{\nu}$-database. We model transactions as sequences of update operations. Note that we do not consider transaction aborts and partially executed transactions, because this is unnecessary for the purpose of retroactively computing the provenance of transactions.

Definition 5.8 *A transaction $T = u_1, \ldots, u_n, c$ is a sequence of updates followed by a commit operation (c). We use $Start(T)$ to denote $\nu(u)$ where u is the first update in T. Similarly, $End(T)$ denotes the commit time of transaction T. A history $H = \{T_1, \ldots, T_n\}$ over a database D is a set of transactions over D such that $<_{\nu}$: $\{(u_i, u_j) \mid \nu(u_i) < \nu(u_j)\}$ is a total order.*

Recall that updates are explicitly part of a transaction in our model and we record when (ν) an update has been executed. This will allow us to determine the state of the database seen by each update of a transaction.

5.4 Historic Databases

An important property of histories in our model is that they completely determine what we refer to as a *historic database*. A historic database D_H for a history H executed under SI encodes the versions of D seen at each point in time by transactions from the history. Each transaction under SI sees a private version of the database. Our definition of a historical database takes this property of SI into account by defining transaction specific versions of each relation - $R[T, \nu]$ denotes relation R as seen by transaction T at time ν.

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

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

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

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

(c) Valid Tuple Versions from Transaction T at ν

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

(d) Tuple Versions Updated By Transaction T

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

Fig. 5: Historic database definition

In contrast to the standard implementations of SI, we do not need to store additional start and end timestamps for tuple versions, because this information is already encoded in the annotation of a tuple. Intuitively, the time ν recorded in a version annotation corresponds to the start time of a tuple version. Under standard SI, a system attribute recording the end time of a tuple version needs to be updated when a new version of this tuple is created. 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. Thus, tuple versions are immutable in our model in the sense that a part of an annotation wrapped in a version annotation may be used as part of a new more complex expression, but will not be modified itself. This greatly simplifies the reenactment approach presented in the next section, because we only need to deal with immutable data.

Definition 5.9 *Let H be a history over a database D, $\mathbb{T}$ the set of transactions in H, and $\mathbb{V}$ a domain of version identifiers. The historic database D_H based on H is a set of historic relations. An n-ary historic relation R^ν is a function $\mathbb{D}^n \times \mathbb{T} \times \mathbb{V} \to K^\nu$. We use $R[T,\nu]$ to denote the restriction of R^ν generated by fixing parameters $\mathbb{T}$ and $\mathbb{V}$ to T and ν and apply the same notation also for databases. Furthermore, we define $R[\nu]$, the snapshot of relation R visible at ν. The definitions of $R[T,\nu]$ and $R[\nu]$ are shown in Figure 5a.*

The complexity of the above definition stems from the fact that it needs to account for the visibility rules of SI. Recall that a transaction T under SI sees 1) its own updates and 2) the updates of transactions that have committed before $Start(T)$. The first condition is encoded in the recursive definition of $R[T,\nu]$ and the second one in the definition of $R[\nu]$.

Relation Versions Visible Inside a 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 first update in a transaction sees $R[Start(T)]$, i.e., the version of R containing all committed changes of transactions committed before T started (2nd case in Figure 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 second update u_2 within the transaction will see the version of R produced by applying update u_1 to $R[Start(T)]$, the third update u_3 will run over the version of R that is the result of applying update u_2 to the result of u_1 and so on. This is encoded by the 3rd and 5th case in Figure 5a. 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 to $R[T,\nu - 1]$ (4th case). If the transaction did not execute any operation at $\nu - 1$ then $R[T,\nu]$ is the same as $R[T,\nu-1]$ (5nd case). Note that this also includes the case where $\nu > End(T) + 1$.

Relation Versions Containing Committed Changes. Under SI, a transaction starting at ν will see a version of relation R that contains all changes of transactions committed before ν. Recall that we use $R[\nu]$ to denote this version of R. Figure 5b to 5d show the definition of $R[\nu]$. $R[\nu]$ can be expressed as a union (sum) over all tuple versions (annotations) created by committed past transactions as long as we make sure that we are not including the same tuple version more than once. Furthermore, we should not include annotations that correspond to tuple versions which have been replaced with newer versions or were deleted. We enforce these two conditions 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 which is the case if the outermost version annotation in k is from T; 2) the tuple version corresponding to k was not updated (predicate UPDATED(T', t, k)) by another transaction T' that committed before ν ($End(T') < \nu$).

Checking for Tuple Updates. UPDATED(T, t, k) is true if transaction T has overwritten the tuple version corresponding to t annotated with k. That is, T has updated or deleted this tuple version. A transaction T has overwritten 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'. Recall that U/D stands for an update, or delete version annotation. 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 on tuple t' contains a summand $R[T, \nu(u)+1](t')[j]$ is $U^{id}_{T,\nu(u)+1}(k)$ or $D^{id}_{T,\nu(u)+1}(k)$.

Example 5.5 *Consider the historic database states of Transaction T_4 from our running example.*

Relation **Order** *at Version 9: Consider the version Order$[T_4, 9]$ valid before T_4 started. Since $Start(T_4) = 9$, this version is equal to Order$[9]$. We construct Order$[9]$ by combining tuple annotations created by transactions that committed before T_4 started (Transactions T_1 and T_2 in the example) as long as these tuple versions have not been overwritten by another already committed transaction. For instance, consider tuple versions o_1 and o_1' which were created by T_1 and T_2. The annotation for o_1' = (oid$_1$,Peter,250) in Order$[9]$ is computed by summing up all annotations on this tuple in the versions of relation* **Order** *created by T_1 and T_2. These are 0 for T_1 and $k = C^1_{T_2,8}(U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1))))$ for T_2. The latter will be included in the annotation for o_1' if we can determine that it was not invalidated by another transaction that committed after T_2 and updated o_1'. This is checked by computing predicate* VALIDAT$(T_2, o_1', k, 9)$ *which returns 1 if there does not exist any such transaction. Since there is no such transaction in the example, we get Order$[9](o_1') = C^1_{T_2,8}(U^1_{T_2,7}(C^1_{T_1,6}(I^1_{T_1,2}(x_1))))$. Tuple $o_1 = (oid_1, Peter, 300)$ is annotated with $k' = C^1_{T_1,6}(I^1_{T_1,2}(x_1))$ in T_1 and 0 in T_2. Since Transaction T_2 updated o_1 and committed after T_1 and before version 9, the predicate* VALIDAT$(T_1, o_1, k', 9)$ *evaluates to 0 and we get Order$[9](o_1) = 0$.*

Importantly, lifted homomorphisms also commute with transactional histories.

Theorem 5.5 *Let h^ν be a lifted homomorphism (Theorem 5.3). h^ν commutes with histories.*

5.5 Provenance Filtering

The annotation of a tuple stores its derivation history since the origin of the database. This amount of information can be overwhelming to a user. We now define how to restrict the provenance to tuples versions related to one transaction. In the $\mathcal{K}^\nu$ model this can be achieved by filtering parts of the annotations (to only track the effect of a certain set of statements) and by replacing subexpressions in annotations that represent parts of the history the user is not interested in with by evaluating them using the homomorphism h_U of the UNV operator. Furthermore, for $\mathbb{N}[X]^\nu$-relations we replace the resulting polynomial with a fresh variable disambiguated by the tuple's identifier. For example, if a subexpression $C^{id}_{T',\nu'}(I^{id}_{T',\nu''}(I^{id_1}_{T',\nu_1}(x_1) \times I^{id_2}_{T',\nu_2}(x_2)))$ where T' is a transaction different from the transaction T we are interested in occurs in an annotation we would replaced it with $C^{id}_{T',\nu'}(x_{id})$. Reconsider Example 2.2 as an example for the application of the definition below.

Definition 5.10 *Let T be a transaction in a history H over database D. The provenance $D[T]$ restricted to T is derived from $D[T, End(T)]$ by replacing each relation $R[T, End(T)]$ with $R[T]$ as defined below.*

$$R[T](t) = \sum_{i=0}^{n(R[T,End(T)](t))} filt(R[T, End(T)](t)[i])$$

$$filt(k) = \begin{cases} C^{id}_{T,\nu}(h_f(k')) & \textit{if } k = C^{id}_{T,\nu}(k') \\ 0 & \textit{else} \end{cases}$$

$$h_f(k) = \begin{cases} k & \textit{if } k \in \mathcal{K} \\ U/I/D^{id}_{T,\nu}(h_f(k')) & \textit{if } k = U/I/D^{id}_{T,\nu}(k') \\ h_f(k_1) + h_f(k_2) & \textit{if } k = k_1 + k_2 \\ h_f(k_1) \times h_f(k_2) & \textit{if } k = k_1 \times k_2 \\ C^{id}_{T',\nu}(h_U(k')) & \textit{if } k = C^{id}_{T',\nu}(k') \wedge \mathcal{K} \neq \mathbb{N}[X] \\ C^{id}_{T',\nu}(x_{id}) & \textit{if } k = C^{id}_{T',\nu}(k') \wedge \mathcal{K} = \mathbb{N}[X] \end{cases}$$

6 Reenactment Queries

We now introduce *reenactment* which enables us to reconstruct the provenance of an update u or transaction T by executing a *reenactment query* $\mathbb{R}(u)$ respectively $\mathbb{R}(T)$. Such a query is annotation equivalent to u respectively T ($u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$), i.e., the operation and its reenactment query produce the same result and provenance. As we will demonstrate later this also implies equivalence for any other MV-semiring $\mathcal{K}^\nu$. We introduce a new operator that adds version annotations, because this is required for reenactment since the operators of $\mathcal{RA}^+$ do not introduce new version annotations.

Definition 6.1 *The operator $\alpha_{X,T,\nu}(R)$ for $X \in \{I, U, D, C\}$ takes as input a $\mathcal{K}^\nu$-relation R and returns a $\mathcal{K}^\nu$-relation where each summand in an annotation k is wrapped in $X^{i(k)}_{T,\nu}$. Here $id(k)$ denotes the identifier of the outermost version annotation in annotation k.*

$$\alpha_{U/D,T,\nu}(R)(t) = \sum_{i=0}^{n(R(t))} U/D^{id(R(t)[i])}_{T,\nu}(R(t)[i])$$

$$\alpha_{I,T,\nu}(R)(t) = \sum_{i=0}^{n(R(t))} I^{id_{new}}_{T,\nu}(R(t)[i])$$

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

Note that id_{new} is determined using skolem function f_{id} as described in Section 5.2 and that $\alpha_{C,T,\nu}$ uses $\text{COM}[T,\nu]()$ introduced Definition 5.7.

6.1 Update Reenactment

We first define reenactment for an update operation u that is executed over the historic database seen by u's transaction T at the time of the update ($R[T, \nu(u)]$). Note that 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 6.2 *Let H be a history over database D. The reenactment query $\mathbb{R}(u)$ for operation u in H is:*

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

$$\mathbb{R}(\mathcal{I}[Q, T, \nu](R)) = R[T,\nu] \cup \alpha_{I,T,\nu+1}(Q(D[T,\nu]))$$

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

An update modifies a relation by applying the expressions from A to all tuples matching condition θ. All other tuples are not modified. We can compute the result of an update as the union between these sets. An insert statement adds the result of a query to the affected relation. It can be reenacted as the union between the relation and the insertion query result. A deletion wraps tuples matching its condition in deletion annotations. Thus, it can be expressed as the union between the original tuples that do not match the condition and the deleted versions of tuples matching the condition.

Example 6.1 *Consider the reenactment query $\mathbb{R}(u_1)$ for the update $u_2 = \mathcal{U}[d = 1, p \to 250, T_2, 6](Order)$ of example Transaction T_2. We abbreviate relation `Order` as O and attributes like in previous examples.*

$$\alpha_{U,T_2,7}(\Pi_{d,c,250\to p}(\sigma_{d=1}(O[T_2,6]))) \cup \sigma_{\neg(d=1)}(O[T_2,6])$$

Theorem 6.1 *Let u be an update and $\mathbb{R}(u)$ its reenactment query. Then, $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$.*

Based on this theorem, reenactment queries can simulate the effect of any update expressible in our model.

6.2 Transaction Reenactment

To reenact a transaction, we merge the reenactment queries for the updates of the transaction in a way that respects the visibility rules enforced by the SI 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 a table R is evaluated over the annotated relation produced by the most recent update u_j with $j < i$ that updated R. 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. Technically, the reenactment of a transaction T is a set of queries. However, abusing terminology we refer to this set as the reenactment query of T and by $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$ denote that each reenactment query for a relation R is equivalent to the effect that transaction T has on this relation.

Definition 6.3 *Let $T = u_1, \ldots u_n, c$ be a transaction in a history H. We use $R(T)$ to denote all relations targeted by at least one update of T and $\text{Last}(R, T, \nu)$ to denote the last update executed before ν in T that updated table R. The reenactment query $\mathbb{R}(T)$ for T is:*

$$\mathbb{R}(T) = \{\mathbb{R}^R(T) \mid R \in R(T)\}$$

$$\mathbb{R}^R(T) = \alpha_{C,T,End(T)+1}(\mathbb{R}^R(\text{Last}(T, R, End(T))))$$

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 (only relation mentions of the form $S[Start(T)]$ remain):

- *Pick a relation mention $S[T, \nu]$ in the current version of $\mathbb{R}^R(u)$*
- *If $\exists u' \in T : u'$ updates $S \wedge \nu(u) < \nu$ then replace $S[T, \nu]$ with $\mathbb{R}^S(\textsc{Last}(T, R, \nu))$*
- *Else replace $S[T, \nu]$ with $S[Start(T)]$*

Example 6.2 *Consider the transaction T_3 from the running example. Let us refer to its operations as u_1 and u_2. We abbreviate relation names as O (Order), S (Outstanding), and C (Collection) and attribute names as in previous examples with the exception that attribute date is denoted as e. Consider the reenactment query for T_3 on C. The last update modifying C is u_2. Thus,*

$$\mathbb{R}^C(T_3) = \alpha_{C,T,13}(\mathbb{R}^C(u_2))$$

Operation u_2 is an insert into relation `Collection` using a query over the state of relation `Outstanding` valid at version 10. The reenactment query for this update is:

$$\mathbb{R}^C(u_2) = \alpha_{I,T_2,11}(\Pi_{o,a}(\sigma_{t<\text{'2000-06'}}(S[T_3, 10]))) \\ \cup C[T_3, 10]$$

The last update of Transaction T_3 that modified relation `Outstanding` before version 10 is u_1. Thus, the access to $S[T_3, 10]$ in $\mathbb{R}^C(u_2)$ is replaced with $\mathbb{R}^S(u_1)$. Relation `Collection` has not been modified by any other update of T_3. Thus, $C[T_3, 10]$ is replaced with $C[Start(T_3)]$.

$$\mathbb{R}^C(u_2) = \alpha_{I,T_2,11}(\Pi_{o,a}(\sigma_{t<\text{'2000-06'}}(\mathbb{R}^S(u_1)))) \cup C[8]$$

The access to relation `Outstanding` by update u_1 is not replaced in $\mathbb{R}^S(u_1)$, because there is no update operation in transaction T_2 that updated this relation before u_1 was executed. The final reenactment query $\mathbb{R}^C(T_3)$ is:

$$\mathbb{R}^C(T_3) = \alpha_{C,T,13}(\mathbb{R}^C(u_2))$$
$$\mathbb{R}^C(u_2) = \alpha_{I,T_2,11}(\Pi_{o,a}(\sigma_{t<\text{'2000-06'}}(\mathbb{R}^S(u_1)))) \cup C[8]$$
$$\mathbb{R}^S(u_1) = \alpha_{U,T_2,9}(\Pi_{o,(a-250)\to a,d}(\sigma_{d=1}(S[8]))) \\ \cup \sigma_{\neg(d=1)}(S[8])$$

We now prove that the reenactment query for a transaction is equivalent to this transaction.

Theorem 6.2 *Let T be a transaction and $\mathbb{R}(T)$ its reenactment query. Then: $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$.*

Green demonstrated [19] that $Q \sqsubseteq_{\mathbb{N}[X]} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}} Q'$ if $\mathcal{K}$ is *naturally ordered* and, thus $Q \equiv_{\mathbb{N}[X]} Q' \Rightarrow Q \equiv_{\mathbb{N}} Q'$. The result is based on the existence of surjective semiring homomorphisms. We do not define what it means for a semiring to be naturally ordered here, but note that many important semirings including all semirings considered here are naturally ordered. Based on the theorem shown below this result translates to queries using the annotation operation defined above and updates in MV-semirings. Thus, reenactment queries also produce the same updated relation as the original operation under bag semantics.

Theorem 6.3 *For Q and Q' be two $\mathcal{RA}^+$ queries and $\mathcal{K}$ a naturally ordered semiring. Then $Q \equiv_{\mathcal{K}^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}} Q'$. Let Q and Q' be two updates or $\mathcal{RA}^+$ queries that may use the annotation operator α and $\mathcal{K}$ a naturally ordered semiring, then $Q \equiv_{\mathbb{N}[X]^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}^\nu} Q'$.*

The theorem above implies that reenactment can be used to replay transactions over any $\mathcal{K}^\nu$-database not just a $\mathbb{N}[X]^\nu$-database. Furthermore, using Unv we can use reenactment compute the same $\mathcal{K}$-database as would have been produced by applying Unv to the result of the reenacted transaction.

7 Relational Reenactment using Time Travel and Audit Logging

We now introduce techniques for retrieving the provenance of transactions using standard DBMS based on a relational encoding of reenactment queries. Our approach uses an audit log to determine which SQL statements were executed when and by which transaction. We demonstrate how reenactment queries can be translated into standard relational algebra queries with time travel that produce a relational encoding of provenance restricted to a transaction (as explained in Section 5.5).

7.1 Relational Encoding of $\mathcal{K}^\nu$-Relations

We extend the relational encoding of provenance polynomials introduced for the Perm [17] project with additional columns that encode version annotations. To encode the filtered provenance $R[T]$ of a transaction T we: 1) normalize $\mathcal{K}^\nu$-expressions according to the operations that were applied to the data and 2) use additional attributes to represent a normalized $\mathbb{N}[X]^\nu$-polynomial.
Normal Form. The basic idea behind this encoding is to represent variables in a normalized polynomial by actual tuple values from the inputs of the query. In particular, we take a polynomial in the normal form introduced in Definition 5.3 that is a sum of products (and version annotations) and order the variables and version annotations in each summand according to the relation and update they belong to. Given the algebra tree for a reenactment query, variables in products mixed with version annotations are ordered according to the leaves of the algebra tree. We add additional attributes to be able to encode such a product and its version annotations, and represent each summand in a normalized polynomial as a separate tuple.

$$\textsc{Sch}(\textsc{Rel}(R[T])) = \textsc{Sch}(R) \triangleright ID_{\mathcal{P}}(\textsc{Sch}^{End(T)}(T,R)) \triangleright \mathcal{U}_C \qquad \textsc{Sch}^{\nu}(T,R) = \begin{cases} \mathcal{P}(T, \textsc{Last}(R,T,\nu)) & \text{if } \exists u \in T : u \text{ updates } R \wedge \nu(u) < \nu \\ \mathcal{P}(R) & \text{else} \end{cases}$$

$$\textsc{Sch}^{\nu}(T, \{t \to k\}) = const$$

$$\mathcal{P}(T,u) = \begin{cases} \textsc{Sch}^{\nu(u)}(T,R) \triangleright \textsc{Sch}^{\nu(u)}(T,X_1) \triangleright \ldots \triangleright \textsc{Sch}^{\nu(u)}(T,X_m) \triangleright \mathcal{U}_{pos(u)} & \text{if } u = \mathcal{I}[Q(X_1,\ldots,X_m),T,\nu](R) \text{ where either } X_i = R_i \text{ or } X_i = \{t_i \to k_i\} \\ \textsc{Sch}^{\nu(u)}(T,R) \triangleright \mathcal{U}_{pos(u)} & \text{else} \end{cases}$$

$$\textsc{Rel}(R[T]) = \bigcup_{t \in R} \bigcup_{i=0}^{n(R[T](t))} t \triangleright \textsc{Rel}^{End(T)}(T,R,R[T](t)[i]) \triangleright True \qquad \textsc{Rel}^{\nu}(T,R,k) = \begin{cases} \textsc{EncU}(T, \textsc{Last}(R,T,\nu), k) & \text{if } \exists u \in T : u \text{ updates } R \wedge \nu(u) < \nu \\ \textsc{EncR}(R,k) & \text{else} \end{cases}$$

$$\textsc{Rel}^{\nu}(T, \{t \to x\}, k) = \textsc{EncR}(\{t \to x\}, k)$$

$$\textsc{EncR}(R,k) = \begin{cases} T', \nu', id, t(x) & \text{if } k = C^{id}_{T',\nu'}(x) \\ \textsc{Null}(\mathcal{P}(R)) & \text{else} \end{cases} \qquad \textsc{EncR}(\{t \to x\}, k) = \begin{cases} id & \text{if } k = x_{id} \\ null & \text{else} \end{cases}$$

$$\textsc{EncU}(T,u,k) = \begin{cases} \textsc{Rel}^{\nu(u)}(T,R,k') \triangleright True & \text{if } u \text{ is a delete or update } \wedge k = \mathcal{A}_{pos(u)}(k') \\ \textsc{Rel}^{\nu(u)}(T,R,k) \triangleright False & \text{if } u \text{ is a delete or update } \wedge k \neq \mathcal{A}_{pos(u)}(k') \\ \textsc{Null}(\textsc{Sch}^{\nu(u)}(T,R)) \triangleright \textsc{Rel}^{\nu(u)}(T,X_1,k_1) \triangleright \ldots \triangleright \textsc{Rel}^{\nu(u)}(T,X_m,k_m) \triangleright True & \text{if } u \text{ is an insert } \wedge k = \mathcal{A}_{pos(u)}(k_1 \times \ldots \times k_m) \\ \textsc{Rel}^{\nu(u)}(T,R,k) \triangleright \textsc{Null}(Q) \triangleright False & \text{if } u \text{ is an insert with query } Q \wedge k \neq \mathcal{A}_{pos(u)}(k_1 \times \ldots \times k_m) \end{cases}$$

Fig. 6: Schema and instance of the relational encoding of $R[T]$

Definition 7.1 *Let T be a transaction. An $\mathbb{N}[X]^{\nu}$ annotation in $R[T](t)$ is in ordered normal form if it is normalized according to Definition 5.4 and variables in each summand k_i are sorted according to $\mathbb{R}^R(T)$.*

Schema. We first define the schema of a relational encoding $\textsc{Rel}(R[T])$ of the provenance for a transaction T using a renaming function P that maps a relation and attribute name to a provenance attribute name. In the following we use $\textsc{Sch}(R)$ to denote the schema of a relation R, and $P(R)$ to denote the list of attribute names containing $P(R,A)$ for each $A \in \textsc{Sch}(R)$ and $\mathcal{P}(R)$ to denote $P(R)$ plus three additional attributes Id, Xid and V that encode a tuples identifier, the transaction creating the tuple version, and the time at which the update creating the tuple version was created, respectively. Furthermore, let $ID_{\mathcal{P}}$ denote a function that takes a list of attribute names and adds unique identifiers to names that occur more than once in the list. We use $\triangleright$ to denote list concatenation, e.g., concatenating lists of attributes.

Definition 7.2 *Let T be a transaction, $\mathcal{U}_i$ denote a boolean attribute representing the version annotation of update $u_i \in T$, and $\mathcal{U}_C$ denote a boolean attribute representing the commit annotation of T. The name of attribute $\mathcal{U}_i$ encodes $\nu(u_i)$, the type of update u_i (insert, delete, update), and the query Q in case the update is an insert. We use $pos(u)$ to denote the position of update u in transaction T. The schema of the relational encoding $\textsc{Rel}(R[T])$ is defined in Figure 6.*

The schema is constructed recursively by tracing back from the last operation in the transaction that modified relation R. If this operation u is an update or delete then we add an attribute $\mathcal{U}$ for storing whether a version annotation for u is used in an annotation. For updates or deletes that are the first operation modifying a relation R, the schema will contain provenance attributes to store the tuple corresponding to a variable in an annotation. For instance, if a transaction T consists of two updates u_1 and u_2 which both updated relation R, then each annotation on a tuple in R will be of the form $U^{id}_{T,\nu(u_2)}(U^{id}_{T,\nu(u_1)}(x))$ where both version annotations are optional. Consequently, the schema of the relational encoding for such annotations will have two attributes $\mathcal{U}_1$ and $\mathcal{U}_2$ to denote which version annotation is present and provenance attributes for relation R to store the tuple corresponding to variable x in the annotation. Since insert operations insert the result of a query into a relation, we have to add provenance attributes to represent the provenance of input tuples to such a query. Assume that the query of an insert is defined over X_1 to X_m where each X_i is either an access to a relation R_i or a singleton operator $\{t \to x\}$. To be able to store the annotation of a tuple from relation R_i in the query's provenance, we have to add attributes to represent all previous updates of T on R_i. Such a list of attributes is then constructed in the same fashion as for the last update of the transaction using $\textsc{Sch}^{\nu(u)}(T, R_i)$. In case of a singleton operator $\{t \to x\}$ we have to add an attribute to store the variable assigned to the tuple t. Note that even though the definitions of $\mathcal{P}(T,u)$ and $\textsc{Sch}^{\nu}(T,R)$ are mutually recursive, these definitions are not circular because $\textsc{Sch}^{\nu}(T,R)$ only refers to $\mathcal{P}(T,u)$ for updates u with $\nu(u) < \nu$. Thus, $\textsc{Sch}^{\nu}(T,R)$ may only depend on $\textsc{Sch}^{\nu'}(T,R)$ if $\nu' < \nu$.

Instance. The instance $\textsc{Rel}(R[T])$ of the relational encoding of $R[T]$ is created by representing each summand k_i in a normalized annotation $R[T](t) = \sum_1^m k_m$ as a separate tuple. The construction of the schema guarantees that this will always be possible.

Definition 7.3 *Consider the provenance $R[T]$ of transaction T in ordered normal form and let* $\text{NULL}(\mathcal{P}(R))$ *and* $\text{NULL}(Q)$ *denote a list of null values with the same arity as $\mathcal{P}(R)$ and the provenance schema for Q (the list of* $\text{REL}^{\nu(u)}(R, X_j, k_j)$ *attributes), respectively. Let $t(x)$ denote the tuple corresponding to a variable x. The relational encoding* $\text{REL}(R[T])$ *is defined in Figure 6.*

The relational encoding of a normalized $\mathbb{N}[X]^{\nu}$-annotation of a tuple in $R[T]$ is constructed following the same procedure as for its schema. For each summand k in a ordered normalized annotation of a tuple t in $R[T]$ we create a tuple in $\text{REL}(R[T])$ by concatenating t and the relational encoding of k. The encoding of k is constructed iteratively by encoding and stripping of parts of k corresponding to updates in T.

Example 7.1 *Figure 3 shows a simplified version of* $\text{REL}(\mathit{Collection}[T_3])$ *for example Transaction T_3. We abbreviate `Collection` as C and `Outstanding` as S and attribute names as in previous examples. Note that we have omitted the $\mathcal{U}_C$, Xid, V, Id, and $P(C)$ attributes to simplify the representation. MV-annotations are shown to the left of each tuple. We use u_1 and u_2 to denote the two operations of this transaction. The schema of this encoding is constructed as follows:*

$$\text{SCH}(\text{REL}(C[T_3])) = \text{SCH}(C) \triangleright ID_{\mathcal{P}}(\text{SCH}^{12}(T_3, C)) \triangleright \mathcal{U}_C$$
$$\text{SCH}^{12}(T_3, C) = \mathcal{P}(T_3, u_2) = \text{SCH}^{10}(T_3, C) \triangleright \text{SCH}^{10}(T_3, S) \triangleright \mathcal{U}_2$$
$$\text{SCH}^{10}(T_3, C) = \mathcal{P}(C) = Id, Xid, V, P(C, o), P(C, a)$$
$$\text{SCH}^{10}(T_3, S) = \mathcal{P}(T_3, u_1) = \text{SCH}^{8}(T_3, S) \triangleright \mathcal{U}_1$$
$$\text{SCH}^{8}(T_3, S) = \mathcal{P}(S) = Id, Xid, V, P(S, o), P(S, a), P(S, d)$$

In this schema, attributes $\mathcal{U}_1$ and $\mathcal{U}_2$ represent the version annotations for updates u_1 and u_2. The insert u_2 in T_3 uses a query $\Pi_{o,a}(\sigma_{d<\text{'2000-06'}}(\mathit{Outstanding}))$. Thus, the schema contains provenance attributes for relation `Outstanding`. Attribute $\mathcal{U}_1$ is added, because update u_1 has previously updated this relation. Consider the 1st tuple in Figure 3 which represents the single summand in the annotation of tuple (oid$_1$,50). The tuple was derived by updating the tuple annotated with x_3 in relation `Outstanding` and then inserting a new tuple into relation `Collection` based on this tuple. Both version annotation attributes are set to true (both updates were involved in the derivation) and the provenance attributes for relation `Outstanding` are used to store the tuple annotated with x_3 in the input.

Given this relational encoding we need to prove that it is lossless, i.e., the encoded MV-relation $R[T]$ can be recovered from $\text{REL}(R[T])$.

Theorem 7.1 *The* $\text{REL}(R[T])$ *operation is lossless.*

7.2 Audit Log and Time Travel

We require the DBMS we use for provenance computation to keep an audit log that stores the SQL code for each update plus 1) when the update was executed and 2) the identifier (*xid*) of its transaction. The audit log is used to determine the operations of a transaction and the database version they have accessed. We assume a standard SI based implementation of time travel as supported in similar fashion by multiple DBMS. Each tuple is annotated with a system time interval (transaction time) that encodes when this tuple version is valid in the database. Update operations create new tuple versions and invalidate tuple versions that are updated by setting their end time to the current time. These modifications are only visible in the updating transaction. When a transaction commits, then new tuple versions are created for all modified tuples with start time set to the transaction commit time. A *snapshot* R_{ν} of relation R contains all committed tuple versions valid at ν. Snapshots have additional attributes TT_b and TT_e storing validity time intervals as well as Xid and Id storing the transaction and tuple identifiers, respectively.

7.3 Relational Implementation of Reenactment

We now discuss how $\mathcal{K}^{\nu}$-relational reenactment queries can be rewritten as standard relational (bag semantics) queries which produce $\text{REL}(R[T])$ for a transaction T. This rewriting of $\mathcal{K}^{\nu}$-queries into bag semantics queries (expressible in SQL) extends previous results for rewriting $\mathcal{K}$-relational queries into bag semantics [17,15]. A query is rewritten by recursively applying rewrite rules for single operators in a top-down fashion. We apply a selection on the boolean version annotation attributes to only return tuple versions from $R[T]$, i.e., that were affected by at least one update of transaction T.

Definition 7.4 *Let $T = u_1, \ldots, u_n, c$ be a transaction and let $\mathcal{U}_i$ denote the version annotation created by the i^{th} update in T. The relational translation* $\text{TR}(\mathbb{R}^R(T))$ *of the reenactment query $\mathbb{R}^R(T)$ restricted to $R[T]$ is computed from $\mathbb{R}^R(T)$ as shown below. The rewrite operator* REW *is defined in Figure 7.* $\text{NULL}(\mathcal{P}(q))$ *denotes a singleton relation with null values for all provenance attributes of q except for version annotation attributes which are set to $false$. Furthermore, ID_x is a function that adds a suffix '`_x`' to attribute names in a list.*

$$\text{TR}(\mathbb{R}^R(T)) = \sigma_{\mathcal{U}_1 \vee \ldots \vee \mathcal{U}_n}(\text{REW}(\mathbb{R}^R(T)))$$

The query produced by the rewrite rules of Figure 7 returns the relational encoding of provenance introduced previously. There are two rules for the union

Structural Rewrite

$$\textsc{Rew}(\{t \rightarrow x_{id}\}) = \Pi_{\textsc{Sch}(t),Id}(\{t \triangleright id\})$$
$$\textsc{Rew}(R[\nu]) = \Pi_{\textsc{Sch}(R),Xid,TT_b \rightarrow V,Id,\textsc{Sch}(R)\rightarrow P(R)}(R_\nu)$$
$$\textsc{Rew}(\sigma_\theta(q)) = \sigma_\theta(\textsc{Rew}(q))$$
$$\textsc{Rew}(\Pi_A(q)) = \Pi_{A,\mathcal{P}(q)}(\textsc{Rew}(q))$$
$$\textsc{Rew}(q_1 \cup q_2) = \textsc{Rew}(q_1) \cup (\textsc{Rew}(q_2) \times \rho_{\mathcal{U}_i}(\{(false)\})) \quad (\text{if } \mathbb{R}^R(u_i) = q_1 \cup q_2 \wedge u_i = \mathcal{U}/\mathcal{D})$$
$$\begin{aligned}\textsc{Rew}(q_1 \cup q_2) = &(\rho_{\textsc{Sch}(q_1),ID_1(\mathcal{P}(q_1))}(\textsc{Rew}(q_1)) \times \textsc{Null}(ID_2(\mathcal{P}(q_2))))\\ &\cup (\Pi_{\textsc{Sch}(q_2),\mathcal{P}(q_1 \cup q_2)}(\rho_{\textsc{Sch}(q_2),ID_2(\mathcal{P}(q_2))}(\textsc{Rew}(q_2)) \qquad (\text{else})\\ &\times \textsc{Null}(ID_1(\mathcal{P}(q_1)))))\end{aligned}$$
$$\begin{aligned}\textsc{Rew}(q_1 \bowtie_\theta q_2) = &\Pi_{\textsc{Sch}(q_1),\textsc{Sch}(q_2),\mathcal{P}(q_1 \bowtie_\theta q_2)}(\rho_{\textsc{Sch}(q_1),ID_1(\mathcal{P}(q_1))}(\textsc{Rew}(q_1))\\ &\bowtie_\theta \rho_{\textsc{Sch}(q_2),ID_2(\mathcal{P}(q_2))}(\textsc{Rew}(q_2)))\end{aligned}$$
$$\textsc{Rew}(\alpha_i(q)) = \Pi_{\textsc{Sch}(\textsc{Rew}(q)),true \rightarrow \mathcal{U}_i}(\textsc{Rew}(q))$$
$$\textsc{Rew}(\alpha_{C,T,End(T)}(q)) = \rho_{\textsc{Sch}(\textsc{Rel}(R[T]))}(\textsc{Rew}(q) \times \rho_{\mathcal{U}_C}(\{(true)\})) \quad (\text{for q} = \mathbb{R}^R(T))$$

Annotation Attributes

$$\mathcal{P}(\{t \rightarrow x_{id}\}) = Id$$
$$\mathcal{P}(R[\nu]) = \mathcal{P}(R)$$
$$\mathcal{P}(\sigma_C(q)) = \mathcal{P}(q)$$
$$\mathcal{P}(\Pi_A(q)) = \mathcal{P}(q)$$
$$\mathcal{P}(q_1 \bowtie q_2) = ID_1(\mathcal{P}(q_1)) \triangleright ID_2(\mathcal{P}(q_2))$$
$$\mathcal{P}(\alpha_i(q)) = \mathcal{P}(q) \triangleright \mathcal{U}_i$$

if $\mathbb{R}^R(u) = q_1 \cup q_2 \wedge u = \mathcal{U}/\mathcal{D}$

$$\mathcal{P}(q_1 \cup q_2) = \mathcal{P}(q_1)$$

else:

$$\mathcal{P}(q_1 \cup q_2) = ID_1(\mathcal{P}(q_1)) \triangleright ID_2(\mathcal{P}(q_2))$$

Fig. 7: Rewrite rules for translating $\mathcal{K}^\nu$-semantics reenactment queries into standard relational semantics (bag)

$$\mathrm{TR}(\mathbb{R}^C(T_3)) = \sigma_{\mathcal{U}_1 \vee \mathcal{U}_2}(\rho_{\textsc{Sch}(\textsc{Rel}(C[T_3]))}(\textsc{Rew}(\mathbb{R}^C(u_2) \times \rho_{\mathcal{U}_C}(\{(true)\}))))$$
$$\textsc{Rew}(\mathbb{R}^C(u_2)) = (\rho_{o,a,ID_1(\mathcal{P}(C[8]))}(\textsc{Rew}(C[8])) \times \textsc{Null}(ID_2(\mathcal{P}(q_2)))) \cup (\Pi_{o,a,\mathcal{P}(\mathbb{R}^C(u_2))}(\rho_{o,a,ID_2(\mathcal{P}(q_2))}(q_2) \times \textsc{Null}(ID_1(\mathcal{P}(C[8])))))$$
$$q_2 = \Pi_{o,a,P(S,o),P(S,a),P(S,d),\mathcal{U}_1,true \rightarrow \mathcal{U}_2}(\sigma_{d<'2000-06'}(\textsc{Rew}(\mathbb{R}^S(u_1))))$$
$$\textsc{Rew}(\mathbb{R}^S(u_1)) = \Pi_{o,(a-250)\rightarrow a,d,Xid,V,Id,P(S,o),P(S,a),P(S,d),true \rightarrow \mathcal{U}_1}(\sigma_{o=1}(\textsc{Rew}(S[8])))) \cup (\sigma_{\neg(o=1)}(\textsc{Rew}(S[8])) \times \rho_{\mathcal{U}_1}(\{(false)\}))$$
$$\textsc{Rew}(C[8]) = \Pi_{o,a,Xid,TT_b \rightarrow V,Id,o \rightarrow P(C,o),a \rightarrow P(C,a)}(C_8)$$
$$\textsc{Rew}(S[8]) = \Pi_{o,a,d,Xid,TT_b \rightarrow V,Id,o \rightarrow P(S,o),a \rightarrow P(S,a),d \rightarrow P(S,d)}(S_8)$$

Fig. 8: Relational translation of the reenactment query for transaction T_3 from the running example

operator. The first one deals with reenactment of an update or delete operation u_i where the annotation attributes of the left union input are the same as for the right input except for the version annotation attribute $\mathcal{U}_i$. The renaming applied in the rewriting of the annotation operator for the commit of the transaction ensures that the schema is that same as defined in Figure 6.

Example 7.2 *Consider the reenactment query for transaction T_3 (Example 6.2) and its standard relational algebra version shown in Figure 8. Accesses to relations Collection (C) and Outstanding (S) are replaced with snapshots and the attributes of these relations are duplicated to encode the variables in the annotations of tuples of C and S. Recall that we represent variables in annotations using the tuples they are annotating using the Xid, TT_b, and V to encode the commit annotation of the past transaction creating the tuple. The part of the query corresponding to update u_1 ($\mathbb{R}^S(u_1)$) has been rewritten by replacing $S[8]$ with its rewritten counterpart. The version annotation operator has been replaced with a projection adding true as the value for annotation attribute $\mathcal{U}_1$. For attribute $\mathcal{U}_1$, which is not in the right input (tuples that were not updated by u_1), we add $\rho_{\mathcal{U}_1}(\{(false)\})$ (we use $\rho_{\mathcal{U}_1}$ to denote renaming of the single attribute of $\{(false)\}$). Update u_2 is an insert that accesses relation `Outstanding`. Recall that in $\mathbb{R}(u_2)$ the access to this relation was replaced with $\mathbb{R}(u_1)$. The rewritten version of $\mathbb{R}^C(u_2)$ applies crossproducts with singletons for union compatibility and uses ID_i to append a suffix to every annotation attribute to prevent name clashes between the two inputs of the union. The annotation operator in $\mathbb{R}(u_2)$ is replaced with a projection $\Pi_{\ldots,true \rightarrow \mathcal{U}_2}$. Finally, the selection $\sigma_{\mathcal{U}_1 \vee \mathcal{U}_2}$ ensures that only tuples that are affected by at least one operation from T_3 are returned.*

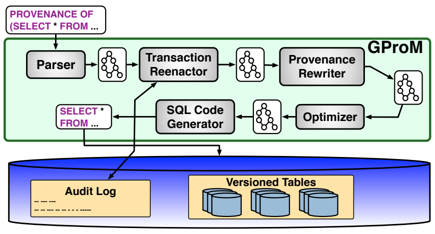

Fig. 9: GProM architecture

This translation of reenactment queries returns the encoding of $R[T]$ that as defined in Section 7.1.

Theorem 7.2 *Let T be a transaction. Then:*

$$TR(\mathbb{R}^R(T)) = \textsc{Rel}(R[T])$$

8 Implementation

We have implemented provenance computation for transactions in **GProM** (**G**eneric **Pro**venance **M**iddleware), a provenance middleware for standard DBMS. Figure 9 shows an overview of the system. GProM translates SQL statements with provenance requests into relational algebra. Requests for transaction provenance are processed by the *reenactor* that constructs a reenactment query using the audit log of the backend DBMS. The *provenance rewriter* rewrites this reenactment query to compute its provenance. Afterwards, we optimize the algebra expression using heuristic rules. The simplified expression is then compiled into SQL code. For an overview of GProM and its unique features see [3].

8.1 Reenacting With CASE

One disadvantage of the reenactment queries produced by our approach is that each `UPDATE` is translated into a union between two accesses to the input relation. For a sequence of updates in a transaction this leads to queries where the left and right input of each such union is again a union operation. Unless intermediate results are reused, this leads to an exponential number of union operations (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 or not. We can reenact update $\mathcal{U}[\theta, A, T, \nu](R)$ using a projection (`SELECT`) constructed as follows. For each expression $e \rightarrow a$ in A we add `CASE WHEN` θ `THEN` e `ELSE` a `END AS` a to the projection. The values of version attributes can be computed in a similar fashion. This approach is also applicable for deletes.

8.2 Prefiltering Provenance

The relational encoding of reenactment introduced in Section 7.3 filters out tuples that were not affected by any update by applying a selection on $\mathcal{U}_1 \vee \ldots \vee \mathcal{U}_n$ to the result of reenactment. 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 $T = u_1, \ldots, u_n, c$ 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 i^{th} 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 will contain all tuple versions created by T. Recall that that snapshots use a column Xid to store the updating transaction. Thus, we can determine which tuple versions got created by a transaction T by running a query $\sigma_{Xid=T}(R_{End(T)})$. To retrieve the versions of these tuples valid at transaction start, we can join the result of this query with $R_{Start(T)}$. Recall that we assume that the database system 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 $\Pi_{\text{SCH}(R_{Start(T)})}(R_{Start(T)} \bowtie_{Id=Id'} \Pi_{Id \rightarrow Id'}(\sigma_{xid=T}(R_{End(T)})))$. This approach is only applicable to relations that are not accessed by any insert's query in the transaction.

9 Experiments

In our experiments we study 1) the performance of provenance computation and 2) the overhead for transaction execution comparing our approach (using reenactment, audit logging and history maintenance) against an approach that directly stores provenance. 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.

9.1 Setup and Workload

Datasets and Workload. We use a relation with five numeric columns. Values for these attributes are chosen from a uniform distribution. 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

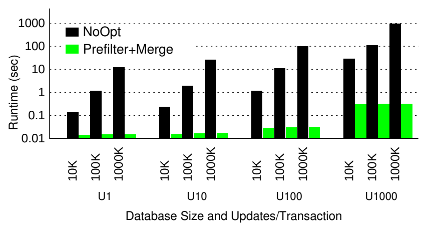

Fig. 10: Relation size

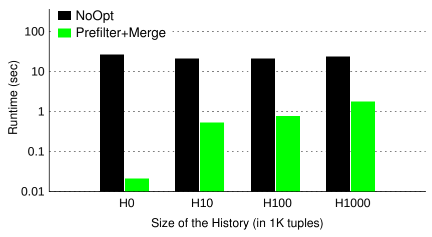

Fig. 11: History size

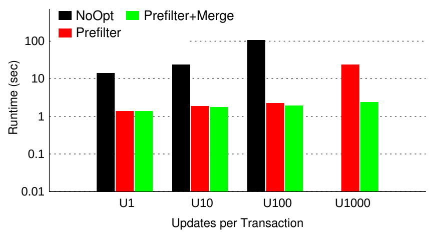

Fig. 12: Optimization methods

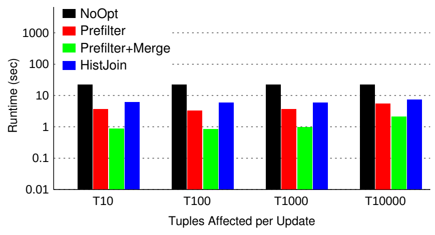

Fig. 13: Affected tuples/update

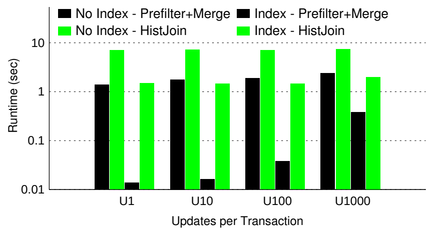

Fig. 14: Index vs. no index

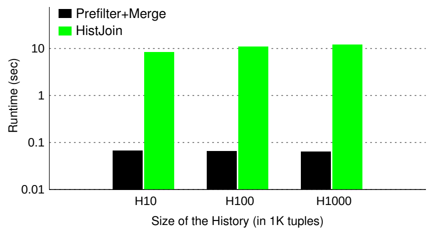

Fig. 15: Inserts, deletes, updates

transaction, e.g., $U10$ is a transaction with 10 updates. T is the number tuples affected by each update. Unless stated otherwise, we use $T1$. The tuple to be updated is selected using the primary key. All transactions were executed under isolation level `SERIALIZABLE` (SI).

Compared Methods. We compare different configurations for computing provenance of a single transaction - each using a subset of the optimizations described in Section 8. 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 disjunctions of all conditions for the transaction's updates (see Section 8.2). The database system was instructed to materialize the intermediate result corresponding to each update in the reenactment query using temporary relations. **Prefilter+Merge (PM)**: 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 Section 8.2. This configuration merges operators where possible.

9.2 Performance of Provenance Computation

In the first set of experiments we have executed the transactional workload beforehand and measure the performance of computing provenance for transactions from this workload to study how our reenactment approach scales in database size, size of the history, and complexity of the transaction (number of operations, amount of modified tuples, mix of update types).

Relation Size and Updates/Transaction. We compute the provenance of transactions varying the number of updates per transaction ($U1$, $U10$, $U100$, and $U1000$) and the size of the database ($R10K$, $R100K$, and $R1000K$). We use the relation without significant history ($H0$) and ran N and PM. Figure 10 shows the runtime of these provenance computations. 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 PM approach is up to three orders of magnitude faster then the naive N configuration.

History Size. We have computed the provenance for transactions with 10 updates ($U10$) over relations with 1M tuples ($R1000K$) and history sizes: $H0$, $H10$, $H100$, and $H1000$. As shown in Figure 11, 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 PM 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. Figure 12 shows results for varying the number of updates ($U1$, $U10$, $U100$, and $U1000$) using *R1000K-H1000*. Compared with P, PM benefits from avoiding materialization. This optimization is more effective for larger transactions, because reenactment queries for such transactions are increasingly complex. While resulting in $\sim$20% improvement for U100, it improves the runtime by a factor of roughly 10 for U1000. The cost of PM is affected by the first selection that is applied to 1M tuples (no index on the history relation). 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 the join between historic relations. PM outperforms HJ by a factor of about 3.

Affected Tuples Per Update. Figure 13 shows results for $U10$ where each update modifies 10, 100, 1000, or 10000 tuples from relation *R1000K-H1000*. As evident from Figure 13, 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 (PM) or the self-join between historic relations (HJ). Increasing the T parameter by 3 orders of magnitude results in an runtime increase of about 150% (PM) and 20% (HJ).

Index vs. No Index. We study the effect of replicating the indexes defined for a relation to its corresponding history relation. Figure 14 shows the results with and without indexes. We have used *R1000K-H1000* and have varied U from $U1$ to $U1000$. We omit the N (no benefit from indexes) and P (consistently outperformed by PM) configurations. Using indexes improves execution time of queries that apply PM considerably.

Inserts and Deletes. We now also use inserts and deletes in addition to updates. We have used the $R1000K$ relation in this experiment. Each operation is chosen randomly (25% probability) from: **1)** An update as used in the previous experiments ($T1$); **2)** an insert that inserts one new tuple; **3)** an insert that inserts the result of a query over a different relation (1 tuple inserted); and **4)** a delete that removes 1 randomly chosen tuple. Figure 15 shows the results for $U20$ varying history size ($H10$ to $H1000$). The results indicate that performance is comparable to performance for updates.

TPC-C. In this experiment, we compute provenance for the TPC-C OLTP benchmark. We executed a TPC-C transactional 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 compared the N and PM methods for the 3 transaction types that execute updates. Figure 16 shows the results for computing the provenance of a single transaction of each type. Each of these transactions only manipulate a few tuples each. Thus, the cost for PM is quite low. The cost for N is dominated by scanning large input relations (millions of tuples).

9.3 History and Audit Logging Overhead

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. The table below 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 for this workload. This seems to align with the performance numbers from DBMS X's documentation which states 5% overhead for mixed read-write workloads. Activating audit logging in addition results in a total overhead of $\sim 19\%$.

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

Tuples in the relation without history are 21 byte large. Activating time travel results in an overhead of 37 bytes for currently valid tuples. Outdated tuple versions occupy 65 bytes on average. The average audit log entry size is 378 bytes (per executed statement).

9.4 Eager Provenance Computation

We now compare our approach with eager provenance computation during transactions execution. We consider two configurations: **1Step** stores a separate provenance record for each tuple version and statement in an extra relation. Each such record is linked to the provenance record for the previous version of the tuple. 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.

Transaction Execution Overhead. Using the workload from Section 9.3 we compare the overhead for transaction execution incurred by these two eager methods with our method (*Reenact*). The results are shown in Figure 18. The performance of our method and *1Step* remains stable when increasing the size of the history. In constrast the overhead of *Full* increases with the history size, because 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 transaction processing showing about a factor 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 tuple affected by each update ($T1$, $T10$, and $T100$). For our method we show the total storage space as well as the breakdown into regular 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

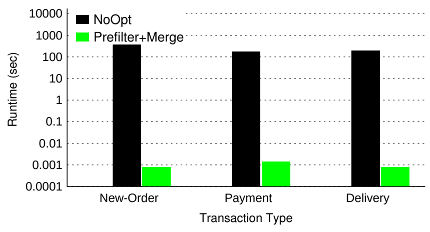

Fig. 16: Provenance for TPC-C

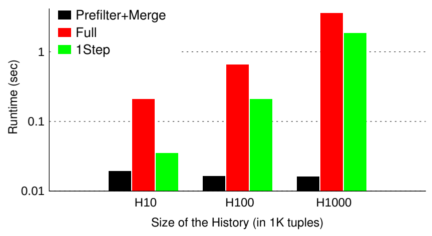

Fig. 17: Provenance computation

Method	Exec. (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%

Fig. 18: Transaction exec. overhead

shown in the table below demonstrate that in the worst case (1 tuple affected per update) our method requires up to ∼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 storage space than the alternatives. Note that in other two approaches, there is no record of the SQL statements that were executed.

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**

Retrieving Provenance. We now compare the performance of reenactment (the *PM* method) for retrieving provenance with *1Step* and *Full*. Figure 17 shows the result for computing 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 the transaction identifier contained in the attribute storing the provenance and *1Step* requires a recursive query or multi-way join to reconstruct the provenance of a transaction from provenance for each update.

9.5 Summary

Our evaluation confirms the efficiency and scalability of our approach - the presented techniques for retroactively computing provenance scale to relations with millions of tuples, large transactions (1000 update statements), large number of updated tuples, and large histories. The proposed optimizations increase performance by several orders of magnitude. The runtime overhead for transaction execution incurred by our approach due to auditing and history maintenance is below 20% for our experimental workloads - a small price to pay compared to eager materialization of provenance while transactions are executed (about 133% and higher).

10 Conclusions

We have presented the first solution for computing the provenance of transactions run under SI. Our approach is based on the novel concept of reenactment queries, i.e., queries that simulate the effect of updates and transactions. We have extended the semiring annotation framework with updates and transactional semantics using version annotations. Using audit logging, time travel, and a relational encoding of reenactment we can retroactively compute the provenance of tuples produced by transactional histories using a standard DBMS. Our experiments confirm that our implementation scales to large databases, histories, and transactions. In future work, we will study reenactment for other concurrency control protocols and more expressive query languages (e.g., aggregation [2]). 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 research is partially supported by a gift from the Oracle Corporation.

References

1. Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. In *PODS*, pages 141–152, 2011.
2. Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for Aggregate Queries. In *PODS*, pages 153–164, 2011.
3. B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for database queries, updates, and transactions. In *TaPP*, 2014.
4. 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.

5. 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.
6. D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4):373–396, 2005.
7. P. Buneman, A. Chapman, and J. Cheney. Provenance Management in Curated Databases. In *SIGMOD*, pages 539–550, 2006.
8. P. Buneman, J. Cheney, and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. *TODS*, 33(4):1–47, 2008.
9. P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.
10. P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *ICDT*, pages 177–188, 2013.
11. J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
12. Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *TODS*, 25(2):179–227, 2000.
13. D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for datalog provenance. In *ICDT*, pages 201–212, 2014.
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 and G. Alonso. Provenance for Nested Subqueries. In *EDBT*, pages 982–993, 2009.
16. B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul. Efficient stream provenance via operator instrumentation. *TOIT*, 14(1):7:1–7:26, 2014.
17. 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.
18. T. J. Green. Containment of Conjunctive Queries on Annotated Relations. In *ICDT*, pages 296–309, 2009.
19. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *PODS*, pages 31–40, 2007.
20. G. Karvounarakis. *Provenance in collaborative data sharing.* PhD thesis, University of Pennsylvania, 2009.
21. G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.
22. G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *TODS*, 38(3):19, 2013.
23. 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.
24. E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *ICDT*, pages 196–207, 2012.
25. D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.
26. R. T. Snodgrass, J. Gray, and J. Melton. *Developing time-oriented database applications in SQL*, volume 42. Morgan Kaufmann Publishers San Francisco, 2000.
27. S. Vansummeren and J. Cheney. Recording Provenance for SQL Queries and Updates. *IEEE Data Engineering Bulletin*, 30(4):29–37, 2007.
28. J. Zhang and H. Jagadish. Lost source provenance. In *EDBT*, pages 311–322, 2010.
29. W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. Loo, and M. Sherr. Distributed time-aware provenance. *PVLDB*, 6(2):49–60, 2013.

A Glossary

The table below given an overview of the notation applied in the paper.

Symbol	Meaning
Updates and Transactions	
ν	a time (version)
id	a tuple identifier
T	a transaction
$End(T)$	transaction T's commit time
u	an update operation
$\nu(u)$	the point in time when u was executed
H	a history
R_ν	snapshot of relation R at time ν
Semirings and MV-semirings	
$\mathcal{A}$	A version annotation
$I/D/U$	either one of I, D, or U
$\{t_1 \rightarrow k_1, \ldots\}$	denotes an annotated relation where tuple t_i is annotated with k_i
$\mathcal{K}$	a semiring
$\mathcal{K}^\nu$	the MV version of semiring $\mathcal{K}$
$\equiv_\mathcal{K}$ /$\sqsubseteq_\mathcal{K}$	denotes query equivalence/containment over $\mathcal{K}$-relations (see [18])
$n(k)$	number of summands in a normalized MV-semiring element k (a sum of subexpressions)
$k[i]$	the ith summand in a normalized MV-semiring element k
χ	An assignment of variables to elements of a semiring
$[k]_\sim$	the congruence class of a symbolic MV-semiring expression k
$k \equiv_\sim k'$	k and k' belong to the same congruence class
MV-updates and Historic Databases	
$\mathcal{U}[\theta, A, T, \nu](R)$	updates tuples in R that fulfill θ using the projection expressions in A
$\mathcal{I}[Q, T, \nu](R)$	Inserts the result of query Q into R
$\mathcal{D}[\theta, T, \nu](R)$	Delete operator: removes all tuples that fulfill θ
$R[\nu]$	version of relation R at time ν
$R[T, \nu]$	version of relation R as seen by transaction T at time ν
$R[T]$	relation R restricted to provenance of transaction T (see Section 5.5)
VALIDAT	functions that returns 1 if part of an annotation of a tuple is valid at a given point in time
UPDATED	predicate that checks whether a transaction has overwritten (updated or deleted the annotated tuple) an annotation
Reenactment	
$\alpha_{X,T,\nu}(R)$	annotation operator the wraps every summand in an the annotation of a tuple in a version annotation $X_{T,\nu}$ where $X \in \{I, U, D, C\}$
$\mathbb{R}(X)$	denotes the reenactment query for operation/transaction/history X
Homomorphisms	
UNV	operator that maps a $\mathcal{K}^\nu$-relation to $\mathcal{K}$-relation by applying homomorphism h_U to each annotation
h_U	homomorphism $\mathcal{K}^\nu$ to $\mathcal{K}$ that maps an $\mathcal{K}^\nu$ element to an element of the embedded semiring $\mathcal{K}$ by evaluating the expression k (interpreting version annotations as functions $\mathcal{K} \rightarrow \mathcal{K}$).
h^ν	a homomorphism ${\mathcal{K}_1}^\nu \rightarrow {\mathcal{K}_2}^\nu$ created by lifting homomorphism $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$
Query Languages	
$\mathcal{RA}^+$	positive relational algebra
$\mathcal{RA}^{+/\alpha}$	positive relational algebra including the annotation operator α
Relational Encoding	
REL(R)	relational encoding of an $\mathcal{K}^\nu$-relation R
REW(q)	rewrite of a query q with $\mathcal{K}^\nu$-relational semantics into a standard relational semantics query returning the relational encoding of the MV-relational output produced by q
$\mathcal{P}(Q)$	annotation attributes used to store part of an annotation by the standard relational encoding of MV-semiring annotated relation
$P(R)$	renaming of the attributes of relation R for storing a variable x in an $\mathcal{K}^\nu$ provenance polynomial.
$\mathcal{P}(R)$	$P(R)$ and additional attributes Xid, $versionAttr$, Id
$\mathcal{U}_i$	boolean attribute recording whether the version annotation of the ith update of a transaction is present in an annotation

NULL(A)	a singleton relation containing a list of null values and $false$ constants with the same arity as A. $false$ is used for $\mathcal{U}$ attributes and null values are used for all other attributes.
$ID_{\mathcal{P}}(A)$	renames attributes in list A to guarantee that every attribute name in A is unique.
$ID_x(A)$	appends a suffix _x to every attribute name in A
Misc	
$\triangleright$	List concatenation
SCH(R)	Schema of relation R

B Proofs

Theorem 5.1 *Let $h : \mathcal{K} \to \mathcal{K}'$ be a semiring homomorphism, then h commutes with any Q in the above algebra if h is applied to Q. Let I be a $\mathcal{K}$ database instance. Then, $h(Q)(h(I)) = h(Q(I))$*

Proof We only need to show that the theorem holds for the new operator $Q = \{t \to k\}$. The result of this operator is a relation R with $R(t') = 0$ for $t' \neq t$ and $R(t) = k$. Applying the homomorphism to R we get a singleton relation $R(t) = h(k)$. Applying the homomorphism to Q we get $h(Q) = \{t \to h(k)\}$. Evaluating this query we get the singleton relation $R(t) = h(k)$ as well.

Theorem 5.2 *h_U is a surjective semiring homomorphism.*

Proof Note that h_U evaluates the symbolic expression of an representative k of a congruence class $[k]_\sim$. To prove that h_U is well-defined we have to show that $k \equiv_\sim k' \Rightarrow h_U(k) = h_U(k')$, i.e., all representative of a congruence class are mapped to the same element of K. If $k \equiv_\sim k'$ then there has to exist at least one sequence of applications of the equivalences that define the congruence relation of $\mathcal{K}^\nu$ (Figure 4) such that applying this sequence to k we get k'. We prove the implication by induction over these equivalences. Most of these equivalences follow directly from the construction of h_U and the definition of MV equivalences:

Laws of commutative semirings

$$\begin{aligned} h_U(k + 0_{\mathcal{K}}) &= h_U(k) +_{\mathcal{K}} h_U(0_{\mathcal{K}}) \\ &= h_U(k) +_{\mathcal{K}} 0_{\mathcal{K}} = h_U(k) \\ h_U(k \times 1_{\mathcal{K}}) &= h_U(k) \times_{\mathcal{K}} h_U(1_{\mathcal{K}}) \\ &= h_U(k) \times_{\mathcal{K}} 1_{\mathcal{K}} = h_U(k) \end{aligned}$$

$$\begin{aligned} h_U(k + k') &= h_U(k) +_{\mathcal{K}} h_U(k') \\ &= h_U(k') +_{\mathcal{K}} h_U(k) = h_U(k' + k) \\ h_U(k \times k') &= h_U(k) \times_{\mathcal{K}} h_U(k') \\ &= h_U(k') \times_{\mathcal{K}} h_U(k) = h_U(k' \times k) \end{aligned}$$

$$\begin{aligned} h_U(k + (k' + k'')) &= h_U(k) +_{\mathcal{K}} h_U(k' + k'') \\ &= h_U(k) +_{\mathcal{K}} h_U(k') +_{\mathcal{K}} h_U(k'') \\ &= h_U(k + k') +_{\mathcal{K}} h_U(k) \\ &= h_U((k + k') + k'') \\ h_U(k \times (k' \times k'')) &= h_U(k) \times_{\mathcal{K}} h_U(k' \times k'') \\ &= h_U(k) \times_{\mathcal{K}} h_U(k') \times_{\mathcal{K}} h_U(k'') \\ &= h_U(k \times k') \times_{\mathcal{K}} h_U(k) \\ &= h_U((k \times k') \times k'') \end{aligned}$$

$$\begin{aligned} h_U(k \times 0_{\mathcal{K}}) &= h_U(k) \times_{\mathcal{K}} h_U(0_{\mathcal{K}}) \\ &= h_U(k) \times_{\mathcal{K}} 0_{\mathcal{K}} = h_U(0_{\mathcal{K}}) \end{aligned}$$

$$\begin{aligned} h_U(k \times (k' + k'')) &= h_U(k) \times_{\mathcal{K}} h_U(k' + k'') \\ &= h_U(k) \times_{\mathcal{K}} (h_U(k') +_{\mathcal{K}} h_U(k'')) \\ &= (h_U(k) \times_{\mathcal{K}} h_U(k')) \\ &\quad +_{\mathcal{K}} (h_U(k) \times_{\mathcal{K}} h_U(k'')) \\ &= h_U((k \times k') + (k \times k'')) \end{aligned}$$

Evaluation of expressions with operands from K

$$\begin{aligned} h_U(k + k') &= h_U(k) +_{\mathcal{K}} h_U(k') \\ &= k +_{\mathcal{K}} k' = h_U(k +_{\mathcal{K}} k') \\ h_U(k \times k') &= h_U(k) \times_{\mathcal{K}} h_U(k') \\ &= k \times_{\mathcal{K}} k' = h_U(k \times_{\mathcal{K}} k') \end{aligned}$$

Equivalences involving version annotations

$$h_U(\mathcal{A}(0_{\mathcal{K}})) = h_U(0_{\mathcal{K}}) = 0_{\mathcal{K}} = h_U(0_{\mathcal{K}})$$

For $\mathcal{A}(k + k')$ we need to distinguish two cases. Either $\mathcal{A} = D^{id}_{T,\nu}$ and we get:

$$\begin{aligned} h_U(\mathcal{A}(k + k')) &= 0_{\mathcal{K}} = 0_{\mathcal{K}} +_{\mathcal{K}} 0_{\mathcal{K}} \\ &= h_U(\mathcal{A}(k)) +_{\mathcal{K}} h_U(\mathcal{A}(k')) \\ &= h_U(\mathcal{A}(k) + \mathcal{A}(k')) \end{aligned}$$

In the second case if $\mathcal{A} \neq D^{id}_{T,\nu}$ we get:

$$\begin{aligned} h_U(\mathcal{A}(k + k')) &= h_U(k + k') = h_U(k) +_{\mathcal{K}} h_U(k') \\ &= h_U(\mathcal{A}(k)) +_{\mathcal{K}} h_U(\mathcal{A}(k')) \\ &= h_U(\mathcal{A}(k) + \mathcal{A}(k')) \end{aligned}$$

Thus, h_U is well-defined and for the remainder of the proof it suffices to restrict the discussion to single representatives of congruence classes.

We now prove that h_U is surjective. Consider an arbitrary element $k \in K$. By construction of K^ν, $k \in K^\nu$. We have $h_U(k) = k$ and, thus, h_U is surjective.

It remains to be shown that h_U is a semiring homomorphism. We have to show that $h_U(0_{\mathcal{K}^\nu}) = 0_\mathcal{K}$, $h_U(1_{\mathcal{K}^\nu}) = 1_\mathcal{K}$, $h_U(k + k') = h_U(k) +_\mathcal{K} h_U(k')$ and $h_U(k \times k') = h_U(k) \times_\mathcal{K} h_U(k')$. Recall that $0_{\mathcal{K}^\nu} = [0_\mathcal{K}]_\sim$ and $1_{\mathcal{K}^\nu} = [1_\mathcal{K}]_\sim$. As proven above we can choose an arbitrary representative of a congruence class when applying h_U. We get $h_U(0_{\mathcal{K}^\nu}) = h_U(0_\mathcal{K}) = 0_\mathcal{K}$ and analog $h_U(1_{\mathcal{K}^\nu}) = h_U(1_\mathcal{K}) = 1_\mathcal{K}$. Furthermore, $h_U(k + k') = h_U(k) +_\mathcal{K} h_U(k')$ and $h_U(k \times k') = h_U(k) \times_\mathcal{K} h_U(k')$ trivially hold based on the definition of h_U. Thus, h_U is a semiring homomorphism.

Theorem 5.3 *Any semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism $h^\nu : {\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$ as defined below. If h is surjective then so is h^ν.*

$$h^\nu(k) = \begin{cases} h(k) & \text{if } k \in K_1 \\ \mathcal{A}(h^\nu(k')) & \text{if } k = \mathcal{A}(k') \\ h^\nu(k_1) + h^\nu(k_2) & \text{if } k = k_1 + k_2 \\ h^\nu(k_1) \times h^\nu(k_2) & \text{if } k = k_1 \times k_2 \end{cases}$$

Proof Note that the mapping h^ν is applied to a representative of a congruence class. We need to prove that if $k \equiv_\sim k'$ then $h^\nu(k) \equiv_\sim h^\nu(k')$. Note that ${\mathcal{K}_1}^\nu$ and ${\mathcal{K}_2}^\nu$ are using the same congruence relations with the exception of evaluating expressions with operands from the embedded semiring which is $\mathcal{K}_1$ in the first case and $\mathcal{K}_2$ in the other. Since by construction h^ν preserves the structure of expressions, the implication holds as long as it is true for any expression which involves only elements from k and the $+$ and $\times$ semiring operations. For elements from $\mathcal{K}_1$ we have $k \equiv_\sim k' \Rightarrow k = k'$ because there are no equivalences in the congruence relation that equate elements from $\mathcal{K}_1$. Thus, $h^\nu(k) = h(k) = h(k') = h^\nu(k')$ and we get $h^\nu(k) \equiv_\sim h^\nu(k')$.

Since the symbolic expressions of ${\mathcal{K}_1}^\nu$ and ${\mathcal{K}_2}^\nu$ are generated by the same grammar with the exception that $k \in \mathcal{K}_2$ instead of $k \in \mathcal{K}_1$, h^ν is obviously a mapping from ${\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$. For the same reason, surjectivity of h implies surjectivity of h^ν. For an expression ${k_2}^\nu$ in ${\mathcal{K}_2}^\nu$ let k_1 to k_n be the elements from $\mathcal{K}_2$ that occur in this expression. Given that h is surjective we can find l_1 to l_n in $\mathcal{K}_1$ such that $h(l_i) = k_i$. Now we construct an element ${k_1}^\nu$ with the same structure as ${k_2}^\nu$, but with l_i instead of k_i for $i \in \{1, \ldots, n\}$. From the constructions and definition of h^ν follows that $h^\nu({k_1}^\nu) = {k_2}^\nu$. It remains to be shown that h^ν is a homomorphism.

$$h^\nu(k_1 + k_2) = h^\nu(k_1) + h^\nu(k_2) \qquad \text{(by construction)}$$
$$h^\nu(k_1 \times k_2) = h^\nu(k_1) \times h^\nu(k_2) \qquad \text{(by construction)}$$

$$h^\nu(0) = 0 \quad (h(0) = 0 \text{ and } k \equiv_\sim k' \Rightarrow h^\nu(k) \equiv_\sim h^\nu(k'))$$
$$h^\nu(1) = 1 \quad (h(1) = 1 \text{ and } k \equiv_\sim k' \Rightarrow h^\nu(k) \equiv_\sim h^\nu(k'))$$

Theorem 5.4 *Let h^ν be a lifted homomorphism as defined in Theorem 5.3. h^ν commutes with updates.*

Proof We have to show for each update operation that $h^\nu(u(R)) = u(h^\nu(R))$. Recall that any lifted homomorphism is history preserving, i.e., it keeps the structure of expressions intact.

Update:

$$\begin{aligned} & h^\nu\Big(\mathcal{U}[\theta, A, T, \nu](R)(t)\Big) \\ =& h^\nu\Big(R(t) \times (\neg\theta)(t) \\ & \quad + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u))} U_{T,\nu+1}^{id(R(u)[i])}(R(u)[i]) \times \theta(u)\Big) \end{aligned}$$

Since h^ν is a homomorphism it commutes with semiring operations and we get:

$$\begin{aligned} =& h^\nu\Big(R\Big)(t) \times (\neg\theta)(t) \\ & \quad + \sum_{u:u.A=t} h^\nu\Big(\sum_{i=0}^{n(R(u))} U_{T,\nu+1}^{id(R(u)[i])}(R(u)[i])\Big) \times \theta(u) \end{aligned}$$

Note that application of a lifted homomorphism such as h^ν to a normalized annotation does not change the structure of summands in this annotation, i.e., h^ν can be pushed into this sum.

$$\begin{aligned} =& h^\nu\Big(R\Big)(t) \times (\neg\theta)(t) \\ + & \sum_{u:u.A=t} \sum_{i=0}^{n(h^\nu\left(R\right)(u))} U_{T,\nu+1}^{h^\nu(id(h^\nu\left(R\right)(u)[i]))}(h^\nu\Big(R\Big)(u)[i]) \times \theta(u) \\ =& \mathcal{U}[\theta, A, T, \nu](h^\nu\Big(R\Big))(t) \end{aligned}$$

Inserts:

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

Recall that based on the construction of h^ν it follows that $h^\nu(I^{id_{new}}_{T,\nu+1}(k)) = I^{id_{new}}_{T,\nu+1}(h^\nu(k)))$. Furthermore, since h^ν is a homomorphism it commutes with queries. Thus,

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

<u>Deletes</u>:

$$\begin{aligned}
& h^\nu\Big(\mathcal{D}[\theta, T, \nu](R)(t)\Big) \\
=& h^\nu\Big(R(t) \times (\neg\theta)(t) \\
& + \sum_{i=0}^{n(R(t))} D^{id(R(t)[i])}_{T,\nu+1}(R(t)[i]) \times \theta(t)\Big) \\
=& h^\nu\Big(R\Big)(t) \times (\neg\theta)(t) \\
& + \sum_{i=0}^{n(h^\nu\left(R\right)(t))} D^{h^\nu(id(h^\nu\left(R\right)(t)[i]))}_{T,\nu+1}(\Big(R\Big)(t)[i]) \times \theta(t) \\
=& \mathcal{D}[\theta, T, \nu](h^\nu\Big(R\Big))(t)
\end{aligned}$$

<u>Commits</u>:

$$\begin{aligned}
& h^\nu\Big(\mathcal{C}[T, \nu](R)(t)\Big) \\
=& h^\nu\Big(\sum_{i=0}^{n(R(t))} \textsc{com}[T, \nu](R(t)[i])\Big) \\
=& \sum_{i=0}^{n(h^\nu\left(R\right)(t))} h^\nu\Big(\textsc{com}[T, \nu](R(t)[i])\Big)
\end{aligned}$$

where

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

so if $R(t) = I/U/D^{id}_{T,\nu'}(k')$ then

$$\begin{aligned}
& h^\nu(\textsc{com}[T, \nu](R(t))) \\
=& h^\nu(C^{id}_{T,\nu+1}(R(t))) \\
=& C^{h^\nu(id(h^\nu\left(R\right)(t)[i]))}_{T,\nu+1}(h^\nu\Big(R\Big)(t)[i]) \\
=& \textsc{com}[T, \nu](h^\nu(R)(t))
\end{aligned}$$

otherwise we get

$$\begin{aligned}
& h^\nu(\textsc{com}[T, \nu](R(t))) \\
=& h^\nu(R(t)) \\
=& h^\nu(R)(t) \\
=& \textsc{com}[T, \nu](h^\nu(R)(t)))
\end{aligned}$$

In summary $h^\nu(\textsc{com}[T, \nu](R(t))) = \textsc{com}[T, \nu](h^\nu(R)(t))$. Thus,

$$\begin{aligned}
& \sum_{i=0}^{n(h^\nu\left(R\right)(t))} h^\nu\Big(\textsc{com}[T, \nu](R(t)[i])\Big) \\
=& \sum_{i=0}^{n(h^\nu\left(R\right)(t))} \textsc{com}[T, \nu](h^\nu(R)(t)[i]) \\
=& \mathcal{C}[T, \nu](h^\nu\Big(R\Big))(t)
\end{aligned}$$

Theorem 5.5 *Let h^ν be a lifted homomorphism (Theorem 5.3). h^ν commutes with histories.*

Proof As was demonstrated before, h^ν commutes with updates and, thus also sequences of updates. Thus, for single transactions the theorem holds. Specifically, for any update u in a transaction T executed at ν we have $u(h^\nu(R[T, \nu]) = h^\nu(u(R[T, \nu]))$

It remains to be shown that h^ν commutes with the computation of $R[\nu]$ over the results of past transactions, i.e., applying h^ν to the result of this computation is the same as applying it to every input $R[T, \nu]$ of the computation. By iteratively pushing the homomorphism through all transactions involved in a history the result follows.

$$\begin{aligned}
& h^\nu\Big(R[\nu](t)\Big) \\
=& h^\nu\Big(\sum_{T \in H \wedge End(T) < \nu} \\
& \sum_{i=0}^{n(R[T,\nu](t))} R[T, \nu](t)[i] \times \textsc{validAt}(T, t, R[T, \nu](t)[i], \nu)\Big) \\
=& \sum_{T \in H \wedge End(T) < \nu} \sum_{i=0}^{n(h^\nu\left(R[T,\nu]\right)(t))} \\
& h^\nu\Big(R[T, \nu]\Big)(t)[i] \times h^\nu\Big(\textsc{validAt}(T, t, R[T, \nu](t)[i], \nu)\Big)
\end{aligned}$$

Since $h^\nu(1) = 1$ and $h^\nu(0) = 0$ we know that

$$\begin{aligned}
& h^\nu\Big(\textsc{validAt}(T, t, R[T, \nu](t)[i], \nu)\Big) \\
=& \textsc{validAt}(T, t, R[T, \nu](t)[i], \nu)
\end{aligned}$$

It remains to be shown that

$$\begin{aligned}
& \textsc{validAt}(T, t, R[T, \nu](t)[i], \nu) \\
=& \textsc{validAt}(T, t, h^\nu(R[T, \nu])(t)[i], \nu)
\end{aligned}$$

under the assumption $h^\nu(R[T,\nu](t)[i]) \neq 0$ (otherwise the value of validAt is irrelevant). Since h^ν does not affect version annotations $k = C^{id}_{T,\nu'}(k') \Rightarrow h^\nu(k) = C^{id}_{T,\nu'}(h^\nu(k'))$.

$$\begin{aligned}
&\textsc{validAt}(T,t,h^\nu(k),\nu) = 1\\
\Leftrightarrow & h^\nu(k) = C^{id}_{T,\nu'}(h^\nu(k'))\\
&\quad \wedge (\neg\exists T' \neq T : End(T') \leq \nu\\
&\qquad \wedge \textsc{updated}(T',t,h^\nu(k)))\\
\Leftrightarrow & k = C^{id}_{T,\nu'}(k')\\
&\quad \wedge (\neg\exists T' \neq T : End(T') \leq \nu\\
&\qquad \wedge \textsc{updated}(T',t,k))
\end{aligned}$$

We now have to prove that $\textsc{updated}(T',t,h^\nu(k)) \Leftrightarrow \textsc{updated}(T',t,k)$.

$$\begin{aligned}
&\textsc{updated}(T,t,h^\nu(k))\\
\Leftrightarrow & \exists u \in T, t', i, j : h^\nu(R[T,\nu(u)])(t)[i] = h^\nu(k)\\
&\quad \wedge h^\nu(R[T,\nu(u)+1])(t')[j] = h^\nu(U/D^{id}_{T,\nu(u)+1}(k))\\
\Leftrightarrow & \exists u \in T, t', i, j : R[T,\nu(u)](t)[i] = k\\
&\quad \wedge R[T,\nu(u)+1](t')[j] = U/D^{id}_{T,\nu(u)+1}(k)
\end{aligned}$$

The last equivalence follows from the fact that we have proven that h^ν commutes with the operations of one transaction above.

Theorem 6.1: *Let u be an update and $\mathbb{R}(u)$ its reenactment query. Then, $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$.*

Proof Proven by substitution of the definitions of update operations, queries, and annotation operators. We show the proof for an update $u = \mathcal{U}[\theta, A, T, \nu](R)$. 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:

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

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

$$\begin{aligned}
&= R(t) \times \neg\theta(t)\\
&+ \sum_{i=0}^{n(Q'(t))} U^{id(Q'(t)[i])}_{T,\nu+1}((\sum_{u:u.A=t} R(u) \times \theta(u))[i])
\end{aligned}$$

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

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

Note that $n(R(u) \times \theta(u)) = n(R(u))$ if $\theta(u) = 1$. If $\theta(u) = 0$ then $n(R(u) \times \theta(u)) \neq n(R(u))$, but this does not affect the result, because then each $R(u)[i] \times \theta(u) = 0$. An analog argument holds for $id(R(u) \times \theta(u))$. Applying the distributivity laws for semirings, we get:

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

Using the MV-semiring equivalence $\mathcal{A}(k \times k') = \mathcal{A}(k) \times k'$ if $k' = 1$ or $k' = 0$ we can pull out the multiplication $\theta(u)$ to get:

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

The proofs for inserts and deletes are analogous.

Theorem 6.2: *Let T be a transaction and $\mathbb{R}(T)$ its reenactment query. Then: $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$*

Proof We prove the theorem by induction over the number of updates in transaction T. For simplicity, we assume that T updates a single relation R. The proof can easily be extended for transactions that update multiple relations.
<u>Induction Start</u>: For a transaction with a single update u_1, the theorem follows from the equivalence result for updates and a simple check for the equivalence of the annotation operator for commits and commit annotations produced by T.
<u>Induction Step</u>: Assume that we have proven that reenactment is annotation equivalent 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 $T_i = u_1, \ldots, u_i, c$. In the induction start we have already proven that the commit operation of a transaction are equivalent to the commit annotation operator in its reenactment query. Thus, we ignore the existence of commit operations in the following proof. WLOG assume $End(T) = End(T_i)$. We know that $\mathbb{R}(T_i) \equiv_{\mathbb{N}[X]^\nu} T_i = R[T_i, End(T_i)] = R[T_i, \nu(u_i)+1]$. 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 historic databases 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 follows that $R[T, End(T)] \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}^R(u_{i+1})$. Since $\mathbb{R}^R(u_{i+1}) = \mathbb{R}^R(T)$, this concludes the proof.

Theorem 6.3 *For Q and Q' be two $\mathcal{RA}^+$ queries and $\mathcal{K}$ a naturally ordered semiring. Then*

$$Q \equiv_{\mathcal{K}^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}} Q'$$

Let Q and Q' be two $\mathcal{RA}^{+/\alpha}$ queries or updates and $\mathcal{K}$ a naturally ordered semiring, then

$$Q \equiv_{\mathbb{N}[X]^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}^\nu} Q'$$

Proof Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be naturally ordered semirings. It was proven in [18] that $Q \sqsubseteq_{\mathbb{N}[X]} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}} Q'$. Furthermore, $Q \sqsubseteq_{\mathcal{K}_1} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}_2} Q'$ iff there exists a surjective semiring homomorphism $\mathcal{K}_1 \rightarrow \mathcal{K}_2$. The first part of the theorem follows from the fact that UNV is a surjective semiring homomorphism (Theorem 5.2) and that the property of being naturally ordered is preserved for $\mathcal{K}^\nu$ semirings (see Lemma B.1 below).

The second parts holds if we can demonstrate that 1) if $\mathcal{K}$ is naturally ordered then so is $\mathcal{K}^\nu$, 2) a lifted homomorphism h^ν is surjective if h is surjective and commutes with updates, the annotation operator, and histories. In particular, since any valuation $\chi : X \rightarrow \mathcal{K}$ can be uniquely extended to a homomorphism $Eval_\chi : \mathbb{N}[X] \rightarrow \mathcal{K}$ [19], 1) and 2) imply the second part. As mentioned above 1) is proven in Lemma B.1. The lifting of homomorphisms was shown to preserve surjectivity (Theorem 5.3) and these homomorphisms commute with updates (Theorem 5.4) and histories (Theorem 5.5). The fact that h^ν commutes with the annotation operator is proven in Lemma B.2.

Lemma B.1 *Let $\mathcal{K}$ be a naturally ordered semiring, then $\mathcal{K}^\nu$ is naturally ordered.*

Proof Let $\mathcal{K}$ be a naturally ordered semiring, i.e., the natural order: $k \leq k' \Leftrightarrow \exists k'' : k + k'' = k'$ is a partial order. Recall that for a relation $\leq$ to be a partial order it has to be reflexive, antisymmetric, and transitive. Consider the natural order on $\mathcal{K}^\nu$. Reflexivity follows from $k + 0 = k$. Transitivity holds because $k_1 \leq k_2 \wedge k_2 \leq k_3 \Rightarrow \exists k_1', k_2' : k_1 + k_1' = k_2 \wedge k_2 + k_2' = k_3 \Rightarrow k_1 + k_1' + k_2' = k_3$. Now let $k_{13} = k_1' + k_2'$. We get $k_1 + k_{13} = k_3 \Rightarrow k_1 \leq k_3$. Thus, it remains to be shown that $\leq$ is antisymmetric. We prove this fact by demonstrating that there are no additive inverses in $\mathcal{K}^\nu$, i.e., the operation of adding an element k' to an element k cannot be inverted by another addition. If this property holds then $\leq$ has to be antisymmetric.

Consider two elements k and k' of $\mathcal{K}^\nu$ in normal form (a sum of elements that do not contain addition). Let $k_1, \ldots, k_n$ be the summands in k and $k_1', \ldots, k_m'$ be the summands in k'. WLOG consider $m = 1$, because if an additive inverse can be found for the sum of $k_1', \ldots, k_m'$ then there has to exist an inverse for each element k_i'. Treat every summand as an ordered tree whose leafs are elements of $\mathcal{K}$ and $\times$-operations are considered n-ary. Furthermore, order operands of such monomials as follows: 1) elements of $\mathcal{K}$ precede any elements wrapped in version annotations and are ordered based on an arbitrary extension of the natural order of $\mathcal{K}$ to a total order; 2) elements wrapped in version annotations are ordered based on some order over $\mathbb{A}$ based on the outermost version annotation; 3) two elements with the same version annotation are ordered based on the order of their children. For example, for $\mathcal{A}_1(\mathcal{A}_2(x_1) \times x_2 \times \mathcal{A}_2(x_3))$ assuming $x_1 < x_2 < x_3$ in the extension of the natural order on $\mathbb{N}[X]$ we would order the elements of the monomial as follows: $\mathcal{A}_1(x_2 \times \mathcal{A}_2(x_1) \times \mathcal{A}_2(x_3))$. We now prove that inverses for an element k' cannot exists by induction over the structure of such summands (trees).

Let $k_\nu \neq 0$ be the element we are trying to invert and $-k_\nu$ represent its inverse (if it exists).

<u>Base case</u>: Consider trees of height 1, i.e., $k_\nu = k \neq 0$ with $k \in \mathcal{K}$. If $-k_\nu = -k$ with $-k \in \mathcal{K}$ then this leads to a contradiction since $\mathcal{K}$ is naturally ordered. To see why this is true consider, $k + -k = 0$ which would imply $k \leq 0$, but also $0 + k = k$ which implies $0 \leq k$. Since we have $k \neq 0$ this yields the contradiction. Thus, if an inverse $-k_\nu$ exists it must contain at least one version annotation. However, it can be shown by induction over equivalences of the congruence relation of $\mathcal{K}^\nu$ that by adding a summand with a version annotation to an element of $\mathcal{K}$ can never yield 0 as a result.

<u>Inductive step</u>: Assume that for any tree of depth up to n we have proven that no inverse exists. Consider $k^{n+1} \neq 0$ as an element whose tree is of height $n + 1$. We have to distinguish two cases: either $k^{n+1} = \mathcal{A}(k^n)$ for some tree k^n of depth n or $k^{n+1} = \prod_{i=1}^m k_i$ where each k_i is of maximal depth n and no inverse exists for any of these k_i.

<u>Case 1 ($k^{n+1} = \mathcal{A}(k^n)$)</u>: Note that the congruence relation of $\mathcal{K}^\nu$ does not manipulate individual version annotations. Thus, $-k^{n+1}$ would have to be of the form $\mathcal{A}(-k^n)$ such that $\mathcal{A}(k^n) + \mathcal{A}(-k^n) = \mathcal{A}(k^n + -k^n) = \mathcal{A}(0) = 0$. However, this leads to a contradiction because k^n is of depth n and thus no additive inverse of k^n can exist.

<u>Case 2 ($k^{n+1} = \prod_{i=1}^m k_i$)</u>: We prove this case by induction over m. If $m = 2$ then $k^{n+1} = k_1 \times k_2$ and WLOG we have to distinguish 2 cases: 1) $k_1 = \mathcal{A}(k_1')$ and $k_2 \in \mathcal{K}$ or 3) $k_1 = \mathcal{A}(k_1')$ and $k_2 = \mathcal{A}(k_2')$. Note that $k_1, k_2 \in \mathcal{K}$ conflicts with the fact that k^{n+1} is of height

$n+1$ and, thus, we do not have to consider this case. In any case we can construct $-k^{n+1}$ as either $-k_1 \times k_2$ or $k_1 \times -k_2$. For any $k \in \mathcal{K}$ no inverse exists. Thus, we have to find the inverse of an element $k_i = \mathcal{A}(k_i')$. However, since k_i of height less than n we know that none such inverse exists. The inductive step is analog.

Lemma B.2 *Let h^ν be a lifted semiring homomorphism (as defined in Theorem 5.3). h^ν commutes with the annotation operator α.*

Proof

$$
\begin{aligned}
&h^\nu\Big(\alpha_{U/D,T,\nu}(R)(t)\Big)\\
=&h^\nu\Big(\sum_{i=0}^{n(R(t))} U/D_{T,\nu}^{id(R(t)[i])}(R(t)[i])\Big)\\
=&\sum_{i=0}^{n(h^\nu\left(R\right)(t))} U/D_{T,\nu}^{h^\nu\left(id(h^\nu\left(R\right)(t)[i])\right)}(h^\nu\Big(R\Big)(t)[i])\Big)\\
=&\alpha_{U/D,T,\nu}(h^\nu\Big(R\Big))(t)
\end{aligned}
$$

For commits recall that

$$h^\nu(\text{COM}[T,\nu](k)) = \text{COM}[T,\nu](h^\nu(k))$$

and thus $h^\nu(\alpha_{C,T,\nu}(R)(t)) = \alpha_{C,T,\nu}(h^\nu(R))(t)$.

For inserts consider that $id_{new} = f_{id}(T,\nu,t,k)$.

$$
\begin{aligned}
&h^\nu\Big(\alpha_{I,T,\nu}(R)(t)\Big)\\
&\qquad=h^\nu\Big(\sum_{i=0}^{n(R(t))} I_{T,\nu}^{id_{new}}(R(t)[i])\Big)\\
&\qquad=\sum_{i=0}^{n(h^\nu\left(R\right)(t))} I_{T,\nu}^{id_{new}}(h^\nu\Big(R\Big)(t)[i])\\
&\qquad=\alpha_{I,T,\nu}(h^\nu\Big(R\Big))(t)
\end{aligned}
$$

Theorem 7.1 *The* REL$(R[T])$ *operation is lossless.*

Proof We prove the theorem by induction over the number of operations in a transaction. Recall that $R[T]$ is derived from $R[T]End(T)$ by applying $filt()$.

<u>**Base Case**</u>: Consider a transaction $T = u, c$ with one operation u. We have to prove that $R[T](t)$ can be recovered from REL$(R[T])$ for any t. We treat each of the three types of update operations separately.

<u>$u = \mathcal{U}[\theta, A, T, \nu](R)$</u>: Consider $R[T](t) = \sum_{i=0}^{n(R[T])} R[T][i]$, the annotation of one tuple t in $R[T]$ and let k_i denote $R[T, End(T)][i]$. Note that that each such k_i is guaranteed to be of the form $C_{T,End(T)}^{id}(U_{T,\nu(u)}^{id}(C_{T',End(T')}^{id}(k_i')))$ with $k_i' \in \mathcal{K}$ and $T' \neq T$. This fact follows immediately from the definition of $filt()$ which removes summands that are wrapped in version annotations of other transactions and replaces subexpressions of the form $C_{T',End(T')}^{id}(k)$ with $C_{T',End(T')}^{id}(x_{id})$ if $T' \neq T$. Since $T = u, c$ every summand is bound to be structured like this.

Now consider the schema created for $R[T]$. Applying the definition shown in Figure 6 the schema is SCH$(R) \triangleright P(R) \triangleright \mathcal{U}_1 \triangleright \mathcal{U}_C$ where $P(R)$ contains attributes Xid, Id, V and a provenance renaming of the attributes of R. The attribute name of $\mathcal{U}_1$ encodes $\nu(u_1)$ and the type of the update. The attribute name of $\mathcal{U}_C$ encodes $End(T)$. According to the definition of REL$(R[T])$ every summand in the annotation of tuple t is encoded as a separate tuple $t \triangleright \text{REL}^{End(T)}(T, R, R[T](t)[i])$. Applying the definition of $\text{REL}^{End(T)}(T, R, R[T](t)[i])$, a summand $C_{T,End(T)}^{id}(U_{T,\nu(u)}^{id}(C_{T',End(T')}^{id}(x_{id})))$ would be encoded as $t \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id}) \triangleright True \triangleright True$. From T', $End(T')$ and id we can directly reconstruct $C_{T',End(T')}^{id}(x_{id})$. Based on the value $\mathcal{U}_1$ ($True$) and $\nu(u_1)$ and the type of the update (U) encoded in the name $\mathcal{U}_1$ it can be determined that the element we have constructed so far should be wrapped in $U_{T,\nu(u)}^{id}$. Finally, $End(T)$ is determined based on the name of $\mathcal{U}_C$.

<u>$u = \mathcal{I}[Q(R_1, \ldots, R_n), T, \nu](R)$</u>: Let k_i denote individual summands in $R[T](t)$ as in the previous case. Every summand k_i is of the form $C_{T,End(T)}^{id}(I_{T,\nu(u)}^{id}(k_{i_1} \times \ldots \times k_{i_n}))$ with $k_{i_j} = 1$, $k_{i_j} = C_{T',End(T')}^{id'}(x_{i_j}')$, or $k_{i_j} = x_{i_j}$. Note that n is number of (not necessarily distinct) relation mentions and constant relation operators in Q, e.g., in $Q = R \times R \times R$ we have $n = 3$. Applying the definition of SCH(REL$(R[T])$)) each k_i would be encoded using $P(R) \triangleright A_1 \triangleright \ldots \triangleright A_n$ where each $A_j = P(R_j)$ for an relation access R_j or $A_j = const$ for a constant relation operator. Recall that repeated attribute names have been disambiguated by $ID_{\mathcal{P}}$. The version annotation for u can be reconstructed as explained above for updates. The attributes of $P(R)$ are guaranteed to be null since all tuple versions in $R[T]$ have been created by u. Based on the definition of REL() if $k_{i_j} = 1$ then the attributes in $P(R_j)$ respective the *const* attribute are null else these attributes store T', $End(T')$ and id' such that $C_{T',End(T')}^{id'}(x_{i_j})$ can be reconstructed analog to the update case or store id (in case of the constant relation operator) and x_{id} can be recovered. Then k_i is reconstructed by multiplying the individual reconstructed operands and wrapping the result in $I_{T,\nu(u)}^{id}$ where id is determined using f_{id} as explained in Sec-

tion 5.2.

$\underline{u = \mathcal{D}[\theta, T, \nu](R)}$: The case for delete is analog to the case for updates.

Inductive Step: Let $T = u_1, \ldots, u_{n+1}, c$ and assume that any annotation produced by a transaction of length up to n can be recovered from its relational encoding. We now show that the same holds for T. We distinguish between three cases based on whether the last operation is an update, insert, or delete.

$\underline{u_{n+1} = \mathcal{U}[\theta, A, T, \nu](R)}$: Consider $R[T](t) = \sum_{i=0}^{n(R[T])}$ $(R[T][i]$ the annotation of one tuple t in $R[T]$ and let k_i denote $R[T, End(T)][i]$. Note that that each such k_i is guaranteed to be of the form $C^{id}_{T,End(T)}(U^{id}_{T,\nu(u)}(k'_i))$ or $C^{id}_{T,End(T)}(k'_i)$ where each k'_i is an annotation produced by a sequence of up to n updates. Thus, if we ignore the commit annotation then k'_i could have been produced by a transaction with up to n updates. Since the definition of the schema $\textsc{Sch}^{End(T)}(T, R)$ and relational encoding $\textsc{Rel}(End(T))$ is recursively defined based on the schema and relational encoding for the first n updates, we know that k'_i can be reconstructed. Specifically, $\textsc{Sch}^{End(T)}(T, R) = \textsc{Sch}(R) \triangleright ID_{\mathcal{P}}(\textsc{Sch}^{\nu(u_{n+1})}($ $T, R) \triangleright \mathcal{U}_{n+1}) \triangleright \mathcal{U}_C$. If the version annotation for u_{n+1} is present in k_i then according to the definition of $\textsc{Rel}()$ attribute $\mathcal{U}_{n+1}$ would be set to true. Based on the induction hypothesis we can reconstruct k'_i and then use the value of this attribute to determine whether $U^{id}_{T,\nu(u)}$ should be added or not.

$\underline{u_{n+1} = \mathcal{I}[Q, T, \nu](R)}$: Every summand k_i in an annotation is either of the form 1) $C^{id}_{T,End(T)}(I^{id}_{T,\nu(u)}(k_{i_1} \times \ldots \times k_{i_m}))$ with $k_{i_j} = 1$, $k_{i_j} = C^{id'}_{T',End(T')}(k_{i_j}')$, or $k_{i_j} = x_{id''}$ (if produced by a constant relation operator) where m is the numer of relation mentions and constant relation operators in Q; or 2) $C^{id}_{T,End(T)}(k_i')$ where k'_u is produced by a sequence of n updates. The schema for the relational encoding is $\textsc{Sch}^{End(T)}(T, R) = \textsc{Sch}(R) \triangleright ID_{\mathcal{P}}(\textsc{Sch}^{\nu(u_{n+1})}(T, R) \triangleright \textsc{Sch}^{\nu(u_{n+1})}(T, X_1) \triangleright \ldots \triangleright \textsc{Sch}^{\nu(u_{n+1})}(T, X_m) \triangleright \mathcal{U}_{n+1}) \triangleright \mathcal{U}_C$. Cases 1) and 3) can be distinguished from case 2) based on whether all attributes in $\textsc{Sch}^{\nu(u_{n+1})}(T, R)$ are null or not. In case 1) k_i' is an annotation produced by less than or equal to n updates and, thus, can be reconstructed based on the induction assumption. In case 2) we can construct the monomial $k_{i_1} \times \ldots \times k_{i_m}$ in the same fashion as in the base case as long as it is possible to determine which attributes in $\textsc{Sch}^{\nu(u_{n+1})}(T, R_1) \triangleright \ldots \textsc{Sch}^{\nu(u_{n+1})}(T, R_m)$ belong to which $\textsc{Sch}^{\nu(u_{n+1})}(T, R_l)$. This is possible using the query Q of the insert which is encoded in $\mathcal{U}_{n+1}$. The tuple id in $I^{id}_{T,\nu(u)}$ is reconstructed using the deterministic scheme introduced in Section 5.2.

$\underline{u_{n+1} = \mathcal{D}[\theta, T, \nu](R)}$: The case for delete is analog to the case for updates.

Theorem 7.2 *Let T be a transaction. Then:*

$TR(\mathbb{R}^R(T)) = \textsc{Rel}(R[T])$

Proof We prove the theorem through induction over the number of operations in a transaction.
Base Case: Consider a transaction $T = u_1, c$ with one operation u and $End(T) = \nu_e$ and $\nu(u_1) = \nu_u$. We treat each of the three types of update operations separately.

$\underline{u_1 = \mathcal{U}[\theta, A, T, \nu](R)}$: The reenactment query $\mathbb{R}^R(T)$ is $\underline{\alpha_{C,T,\nu_e+1}(\alpha_{U,T,\nu_u+1}(\Pi_A(\sigma_\theta(R[\nu_u])) \cup \sigma_{\neg\theta}(R[\nu_u]))}$. This query would return $T[\nu_e + 1,]$, the version seen within transaction T at its commit. $R[T]$ is derived from this version by removing summands that are not wrapped in a commit annotation of T and replacing subexpressions of the form $C^{id}_{T',End(T')}(k')$ in the remaining summands with $C^{id}_{T',End(T')}(x_{id})$. The relational translation $TR(\mathbb{R}^R(T))$ of this reenactment query is

$$
\begin{aligned}
&TR(\mathbb{R}^R(T)) = \sigma_{\mathcal{U}_1}(\textsc{Rew}(\mathbb{R}^R(T)))\\
&\textsc{Rew}(\mathbb{R}^R(T)) = \rho_{\textsc{Sch}(\textsc{Rel}(R[T]))}(\textsc{Rew}(q) \times \rho_{\mathcal{U}_C}(\{(true)\}))\\
&\textsc{Rew}(q) = (\Pi_{A,\mathcal{P}(R),True \to \mathcal{U}_1}(\sigma_\theta(\textsc{Rew}(R[\nu_u]))))\\
&\quad \cup (\sigma_{\neg\theta}(\textsc{Rew}(R[\nu_u])) \times \rho_{\mathcal{U}_1}(\{(false)\}))\\
&\textsc{Rew}(R[\nu_u]) = \Pi_{\textsc{Sch}(R),Xid,TT_b \to V,Id,\textsc{Sch}(R) \to \mathcal{P}(R)}(R_{\nu_u})
\end{aligned}
$$

Consider $k = R[T](t)$ for an arbitrary tuple t. As shown in the proof for Theorem 7.1 each summand k_i in k will be of the form

$$C^{id}_{T,End(T)}(U^{id}_{T,\nu(u_1)}(C^{id}_{T',End(T')}(x_{id})))$$

with $T' \neq T$. In the relational encoding each such summand k_i will represented as

$$t_{k_i} = t \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id}) \triangleright True \triangleright True$$

where the two $True$ constants are for attributes $\mathcal{U}_c$ and $\mathcal{U}_1$. We have to prove that iff k_i in $R[T](t)$ then t_{k_i} is in the result of $TR(\mathbb{R}^R(T))$.

$\Rightarrow$: If k_i is a summand then there exists a tuple t' corresponding to $C^{id}_{T',End(T')}(x_{id})$ in R_{ν_u}. Thus, in $\textsc{Rew}(R[\nu_u])$ there will be a tuple $t' \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id})$. This tuple fulfills the condition θ of the update, because otherwise k_i would not occur in $R[T](t)$. Hence, it will be in the left input of the union in $\textsc{Rew}(q)$ and would fulfill the condition of the final selection $\sigma_{\mathcal{U}_1}$. After application of the projection $\Pi_{A,\ldots}$, we get

$t' \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id}) \triangleright True$. Because of the crossproduct with $\rho_{\mathcal{U}_C}(\{(true\})$) the final result tuple will be $t \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id}) \triangleright True \triangleright True = t_{k_i}$.

$\Leftarrow$: Assume that t_{k_i} is in the result of $TR(\mathbb{R}^R(T))$. Since the final operation in $TR(\mathbb{R}^R(T))$ is a selection on $\mathcal{U}_1$ and $\mathcal{U}_1$ is only true in the left branch of the union in $\textsc{Rew}(q)$ we know that this tuple is from the left input of the union. This immediately implies that there has to exist a tuple $t' \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id})$ in $\textsc{Rew}(R[T])$ which fulfills the condition of the update u_1. Thus, a summand k_i corresponding to this tuple will be in $R[T](t)$.

$\underline{u_1 = \mathcal{I}[Q(R_1, \ldots, R_n), T, \nu](R)}$: The reenactment query for u_1 is $R[T, \nu_u] \cup \alpha_{I,T,\nu_u+1}(Q(D[T, \nu_u]))$. The relational rewrite for $TR(\mathbb{R}^R(T))$ is

$$
\begin{aligned}
&TR(\mathbb{R}^R(T)) = \sigma_{\mathcal{U}_1}(\textsc{Rew}(\mathbb{R}^R(T)))\\
&\textsc{Rew}(\mathbb{R}^R(T)) = q_1 \cup q_2\\
&q_1 = (\rho_{\textsc{Sch}(R),ID_1(\mathcal{P}(R))}(\textsc{Rew}(R[\nu_u]))\\
&\qquad \times \textsc{Null}(ID_2(\mathcal{P}(Q) \triangleright \mathcal{U}_1))))\\
&q_2 = (\Pi_{\textsc{Sch}(q_2),ID_1(\mathcal{P}(R)),ID_2(\mathcal{P}(Q)\triangleright\mathcal{U}_1))}(\\
&\qquad\qquad \rho_{\textsc{Sch}(Q),ID_2(\mathcal{P}(Q)\triangleright\mathcal{U}_1)}(\\
&\qquad\qquad\qquad \Pi_{\textsc{Sch}(Q),\mathcal{P}(Q),True\rightarrow\mathcal{U}_1}(\textsc{Rew}(Q))\\
&\qquad \times \textsc{Null}(ID_1(\mathcal{P}(R[\nu_u])))))\\
&\textsc{Rew}(R[\nu_u]) = \Pi_{\textsc{Sch}(R),Xid,TT_b\rightarrow V,Id,\textsc{Sch}(R)\rightarrow P(R)}(R_{\nu_u})\\
&\textsc{Rew}(R_i[\nu_u]) = \Pi_{\textsc{Sch}(R_i),Xid,TT_b\rightarrow V,Id,\textsc{Sch}(R_i)\rightarrow P(R_i)}(R_{i\nu_u})
\end{aligned}
$$

Since $\mathcal{U}_1$ is true in the right input of the union and false in the left input, because of $\textsc{Null}(ID_2(\mathcal{P}(Q) \triangleright \mathcal{U}_1)))$, any result returned by $TR(\mathbb{R}^R(T))$ originates in the right input. It remains to be shown that $\textsc{Rew}(Q)$ produces the correct result, because the additional version annotation attributes ($\mathcal{U}_1$ and $\mathcal{U}_C$), derived in the same fashion as for update, are *true*. Note that for queries the rewrite rules are the rewrite rules introduced in Perm which were shown to derive a relational encoding of provenance polynomials [17] except that a snapshot of relations is accessed and that the provenance attributes for a relation R contain additional attributes Xid, V, and Id. Since these additional attributes are not treated any different from the other provenance attributes of R in the rewrite rules, the correctness of $\textsc{Rew}(Q)$ follows from the correctness of the Perm rewrites.

$\underline{u = \mathcal{D}[\theta, T, \nu](R)}$: analog to the case for updates.

Inductive Step: Let $T = u_1, \ldots, u_{n+1}, c$ and assume that the relational rewrite for any transaction of length up to n is correct. We now show that the same holds for T. We distinguish between three cases based on whether the last operation is an update, insert, or delete. We need to show that the new parts of annotations added by u_{n+1} under $\mathcal{K}^{\nu}$-relational semantics are correctly encoded and that the correct encoding of annotations in the input is preserved in the output if these annotation occur in an annotation produced by u_{n+1}.

$\underline{u_{n+1} = \mathcal{U}[\theta, A, T, \nu](R)}$: Each summand k_i in an annotation $k = R[T](t)$ for a tuple t is either of the from

$$C^{id}_{T,End(T)}(U^{id}_{T,\nu(u_{n+1})}(k'_i)$$

where k'_i is an annotation produced by any of the previous updates of T that affected R or a summand in an annotation on a tuple in $R[Start(T)]$ (in case the annotated tuple in the input of u_{n+1} fulfills the condition the update u_{n+1} and, thus, was updated) or

$$C^{id}_{T,End(T)}(k'_i)$$

The first case is analog to the base case for updates with the only exception that the relational encoding of k'_i is more complex. However, observe that the proof of the base case does not make use of the properties of k'_i. Thus, the relational encoding of all summands that belong to the first type is correct in $TR(\textsc{Rew}(T))$. In the second case, observe that k_i would occur as a summand in an annotation on tuple t in $T' = u_1, \ldots, u_n, c$ and according to the induction hypothesis the encoding of k_i is correct in the input of u_{n+1}. Let t_{k_i} be this relational encoding. It remains to be shown that the relational translation of $\mathbb{R}^R(u_{n+1})$ propagates this encoding to its output assuming that t does not fulfill the update's condition (otherwise k_i would not occur in the annotation of t in $R[T]$). Since t does not fulfill θ, t_{k_i} would is present in the right input of the union in $\textsc{Rew}(q)$ (where q is the union in the reenactment query for u_{n+1} as in the proof of the base case for updates). Tuple t fulfills $\neg\theta$ and, thus is extended with $(false, true)$ (attributes $\mathcal{U}_{n+1}$ and $\mathcal{U}_C$). That is, as was to be proven $TR(\mathbb{R}^R(T))$ returns $t_{k_1} \triangleright false \triangleright true$.

$\underline{u = \mathcal{I}[Q(R_1, \ldots, R_n), T, \nu](R)}$: Consider a summand k_i in an annotation $k = R[T](t)$ for a tuple t. Again, we have to distinguish between two cases: summands produced by the insert and summands that are already present in the previous version of relation R before executing the insert. The correctness of the first case follows based on the proof for the base case if for $k_i = C^{id}_{T,End(T)}(I^{id}_{T,\nu(u_{n+1})}(k_{i_1} \times \ldots \times k_{i_n})$ we have that for each k_{i_j} the relational encoding of k_{i_j} in the version of relation R_j before execution of the insert is propagated correctly by the relational translations of the reenactment query for u_{n+1}. This is proven analog to the base case based on the correctness of the Perm rewrites for regular provenance polynomials and the observation that the propagation for MV-semiring elements only differs in the propagated attributes and these attributes are not affected by the rewrite rules.

The second case is trivial since the reenactment query for an insert unions the relational encoding of the previous version of relation R before the insert with the result of the insert's query. Thus, any tuple in the previous version of R is propagated by extending it with $(false, true)$ (attributes $\mathcal{U}_{n+1}$ and $\mathcal{U}_C$).

$\underline{u = \mathcal{D}[\theta, T, \nu](R)}$: The case for delete is analog to the case for updates.