ILLINOIS INSTITUTE OF TECHNOLOGY

Reenacting Transactions to Compute their Provenance

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

IIT DB Group Technical Report IIT/CS-DB-2014-02

2014-09

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

LIMITED DISTRIBUTION NOTICE: The research presented in this report may be submitted as a whole or in parts for publication and will probably be copyrighted if accepted for publication. It has been issued as a Technical Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IIT-DB prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. payment of royalties).

Reenacting Transactions to Compute their Provenance

Bahareh Arab
Illinois Institute of Technology
barab@iit.edu

Dieter Gawlick
Oracle Corporation
dieter.gawlick@oracle.com

Vasudha Krishnaswamy
Oracle Corporation
vasudha.krishnaswamy@oracle.com

Venkatesh Radhakrishnan
Oracle Corporation
venkatesh.radhakrishnan@oracle.com

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

ABSTRACT

Database provenance is essential for auditing, data debugging, understanding transformations, and many additional use cases. While these applications do benefit from state-of-the-art provenance tracking for queries, most use cases also require provenance for transactional updates. We present the first provenance model for concurrent database transactions. Our model extends the well-known semiring provenance framework with version annotations and update operations. Based on this model, we present the first solution for computing the provenance of database transactions. Our approach can retroactively trace transaction provenance as long as an audit log and time travel functionality are available (both are supported by most DBMS) and without storing any additional information. For a given transaction, our approach constructs a *reenactment query* that simulates the effect of the transaction. This query is guaranteed to produce the updated versions of tables produced by the transaction and has the same provenance as the original transaction, i.e., it is annotation-equivalent. Using time travel and by adopting well-known techniques for computing the provenance of queries, we can use reenactment to retroactively compute the provenance of transactions. Currently, we support two widely applied concurrency control mechanisms: snapshot isolation and read committed snapshot isolation. We have implemented a prototype on-top of a commercial database system and our experiments confirm that 1) the runtime and storage overhead required to support time-travel and the audit log is tolerable and 2) by applying novel optimizations we can efficiently compute the provenance of large transactions over large data sets.

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 errors in input data, understanding complex data transformations, and as a supporting technology for data 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. These techniques have been pioneered by systems such as Perm [18], DBNotes [7], Orchestra [25], and others to compute the provenance of queries. While provenance for queries is important, many use cases (e.g., auditing) would benefit from provenance for update operations. In relational databases, updates are executed as part of transactions and DBMS apply concurrency control techniques to guarantee ACID properties for transactions. Provenance tracking for database updates needs to take into account the transactional semantics and idiosyncrasies of concurrency control mechanisms to correctly describe the origin of tuple versions.

While there are existing solutions for computing the provenance of update operations [25, 35, 8], no solution exist for tracking the provenance of transactions. Developing a provenance model for transactional updates is challenging, because such a model needs to correctly represent the complex interdependencies between tuple versions that are the result of concurrent transactions. Provenance can easily outgrow the data it is describing and computing provenance can result in significant overhead. Ideally, it should be possible to compute the provenance of any past transaction without having to eagerly materialize provenance information during transaction execution. This would have the advantage of avoiding the runtime and storage overhead of provenance computation for every transaction executed by the system.

In this work we present a provenance model for transactional updates using the *snapshot isolation* (*SI*) and *read committed snapshot isolation* (*RC-SI*) concurrency control protocols (these protocols are applied by many commercial and open-source systems including PostgreSQL, Oracle, and MSSQL). Our model is an extension of the semiring annotation framework and inherits the advantages of this model. Specifically, it generalizes several extensions of the relational model including set-semantics, bag-semantics, and various less informative types of provenance. Based on this provenance model we introduce the novel concept of *reenactment queries*. Reenactment queries are queries that simulate the effect of an update or transaction. Importantly, reenactment queries are annotation equivalent to the operation they are simulating, i.e., they have the same provenance. Reenactment is the main enabler of our approach for retroactively

Technical Report
.

Bus

	company	number	price	fromCity	toCity
b_1	Whitedog	13	210	New York	Chicago
b_2	Whitedog	15	140	Seattle	Chicago
b_3	Picobus	2	65	Chicago	Schaumburg

Schedule

	company	bnum	departureTime
s_1	Whitedog	13	08:15
s_2	Whitedog	15	16:00
s_3	Picobus	2	10:30
s_4	Picobus	2	12:30

Figure 1: Database Instance For Running Example

Bus after Transaction 1

	company	number	price	fromCity	toCity
b'_1	Whitedog	13	210	New York	Chicago
b'_2	Whitedog	15	140	Seattle	Chicago
b'_3	Picobus	2	65	Chicago	Schaumburg
b'_4	Greyhound	5	96	Chicago	New York

Schedule after Transaction 2

	company	bnum	departureTime
s'_1	Whitedog	15	16:00
s'_2	Picobus	2	10:30
s'_3	Greyhound	5	20:15

Figure 3: Instance After Executing Example Transactions

computing the provenance of transactions. We demonstrate how our provenance model can be encoded as standard relations and how to implement provenance computation for transactions by translating reenactment queries into SQL queries using time travel to access past database states. Such queries can be executed over any DBMS that supports time travel (e.g., Oracle [30] and MSSQL [28]). Time travel essentially requires a transaction time history [22, 33] for each relation in the database. For systems that do not support this feature, there exist standard approaches for keeping a transaction time history by creating a history table and using triggers to backup old versions of updated rows to the history table.

EXAMPLE 1 (RUNNING EXAMPLE). *Consider the example bus schedule database shown in Figure 1. The database has two relations* `Bus` *and* `Schedule` *which store bus routes of traveling companies and the scheduled departure times for each route. Two transactions were executed (Figure 2) over this database. Transaction T_1 inserts a new bus route into relation* `Bus`*. Transaction T_2 deletes all scheduled departures for the* Whitebus *route* 13*, then inserts a new departure time (*20:15*) for all* Greyhound *bus routes, and then updates the departure times for all* Picobus 2 *departures to* 10:30*. The instances for relations* `Bus` *and* `Schedule` *after the execution of these transactions are shown in Figure 3. The provenance of the tuples in the updated instances should encode their whole derivation history - from which data was that tuple derived and by which operations. For example, the new departure time for the* Greyhound 5 *route (tuple s'_3) was inserted by transaction T_2 based on a the corresponding bus route tuple (b'_4) inserted by T_1. The* `UPDATE` *of transaction T_2 has modified tuples s_3 and s_4 by updating their departure time to* 10:30*. The provenance of the updated tuples (s'_2) should model from which original tuples these updates tuples were derived from (s_3 and s_4) and which operation did create these tuples (the update of transaction T_2). When tracing the provenance of concurrent transactions, it is critical to take the concurrency control protocol into account. The provenance depends on which tuples were visible to an operation of a transaction at a certain point in time.*

The main contributions of this work are:

- We introduce the **multi-version provenance model**, a provenance model for database transactions that extends the well-established semiring annotation framework. This model gives full account of the provenance of tuple versions produced by concurrent transactional updates. In particular, we extend the semiring annotation model with update operations and versioning annotations that keep track of how a tuple was updated. Furthermore, we study SI and RC-SI as concurrency control protocols for multi-version annotated databases. Provenance polynomials extended with version annotations are the most general instance of this model. Thus, similar to how $\mathcal{K}$-relations generalize extensions of the relational model and several provenance models, our model generalizes these extensions under transactional updates.

- We introduce **reenactment queries**, queries that reenact the modifications of an update, transaction, or complete history. Reenactment queries are annotation-equivalent to the operation they are reenacting, i.e., they produce the same result (updated relations) and have the same provenance. Thus, reenactment queries can be used to track the provenance of updates and transactions.

- We present an **relational encoding** of the multi-version annotation model and demonstrate how reenactment queries can be implemented as standard relational queries using **time travel** to access past database versions and an **audit log** to construct such reenactment queries. Since most database systems support these features, we can apply this approach to retroactively compute transaction provenance over multiple database backends without having to explicitly store any provenance information. One important advantage of this approach is that rewrite based annotation-propagation techniques that were originally introduced to compute the provenance of queries can easily be extended for reenactment queries and, thus, can be lifted to compute the provenance of transactions.

- We present an implementation within our **GProM** system running as a middleware on-top of commercial DBMS X. GProM extends SQL with constructs for computing the provenance of queries and transactions that are compiled into SQL queries expressed in the dialect of the backend DBMS. These provenance language constructs are seamlessly integrated within the SQL language, e.g., provenance requests can be used as subqueries.

- We discuss several optimization techniques that make the relational encoding of reenactment queries digestible by standard optimizers. Specifically, we introduce heuristic rewrites that simplify the translated reenactment queries, discuss alternatives for encoding reen-

Transaction 1	Transaction 2	Time
`INSERT INTO Bus (company, number, price, fromCity, toCity) VALUES (Greyhound,5,96,Chicago,New York);`		10
`COMMIT;`		11
	`DELETE FROM Schedule WHERE company='Whitedog' and bnum=13 ;`	12
	`INSERT INTO Schedule (company, bnum, departureTime) SELECT company, number,'20:15'FROM Bus WHERE company='Greyhound';`	13
	`UPDATE Schedule SET departureTime='10:30' WHERE company='Picobus'and bnum=2;`	14
	`COMMIT;`	15

Figure 2: Transactions Example

actment queries as standard relational queries, and present techniques for filtering unrelated information from the provenance early on.

- Our extensive experimental evaluation demonstrates that 1) reenactment is efficient and scales to large databases, complex transactions, and large number of updates; 2) the storage and runtime overhead paid to support time travel and audit logging is tolerable.

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

2. RELATED WORK

Provenance of relational queries has been studied extensively in the recent years. Buneman et al. [9] where the first to distinguish Why-provenance (which tuples were used to compute a result) and Where-provenance (which inputs is a value in the result copied from). This work has addressed this problem for a hierarchical data model. See [12] for a relational version. Cui et al. [13] introduced the Lineage provenance model and presented an implementation based on tracing queries that iteratively trace the provenance of an output though a relational query. The seminal paper from Green et al. [20] introduced the K-relational model, an extension of the relation model with annotations from a commutative semiring and has shown how such annotations propagate through positive relational algebra ($\mathcal{RA}^+$) queries. In this model, the semiring of provenance polynomials is the most general form of semiring annotations. Provenance polynomials generalize the relational datamodel (set and bag semantics), several extensions (e.g., incomplete databases and trust), and several less informative provenance models including Lineage, Why-provenance, and minimal Why-provenance. See [24] for an overview of this model and its extensions beyond positive relational algebra (e.g., set difference [14, 2, 19] and aggregation [3, 32]). Kostylev et al. [27] study 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. [29] discuss factorization of provenance polynomials and Amsterdamer et al. [1] rewrite queries into equivalent queries with minimal provenance. It has been proven that provenance polynomials can be extracted from the PI-CS [18] and Provenance Games [26] models. Our approach extends the semiring annotation framework with update operations and transactional semantics. The idea of annotating parts of a provenance polynomial with functional symbols was, to the best of our knowledge, first applied in the context of the Orchestra system to record applications of schema mappings [23]. The version annotations in our model were inspired by this idea.

Systems such as DBNotes [7], Orchestra [25], and Perm [18] 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 transactions, but instead reconstruct provenance when requested. Zhang et al. [36] demonstrated that an audit log and time travel functionality is sufficient for computing the provenance of past queries. This approach only requires minor modifications to the standard query rewrite rules for provenance computation using annotation propagation to trace a past query's provenance. 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) and reconstructing provenance by replaying such operations has also been applied in the DistTape system [37] in the context of distributed datalog processing and the Ariadne system [17] in the context of stream processing.

The provenance of updates has been studied in related work [25, 8, 35, 5], but none of these approaches addresses

the complications that arise when updates are run as parts of concurrent transactions.[1] Buneman et al. [8] have studied a copy-based provenance type for the nested update language and nested relational calculus. Vansummeren et al. [35] define provenance for SQL DML statements. This approach modifies the updates to store provenance. Our approach differs in that we reconstruct provenance on demand instead of computing and storing provenance for all operations. Furthermore, we are the first to compute provenance for transactions.

In terms of defining the semantics of concurrent transactions, we rely on standard concurrency control protocols that are widely applied in commercial and open-source database systems. In particular, we focus on the snapshot isolation (SI) [6] and read committed snapshot isolation (RC-SI) multi-versioning concurrency control protocols corresponding to isolation levels `SERIALIZE` and `READ COMMITTED` in systems such as Oracle and PostgreSQL.[2]

3. BACKGROUND

3.1 Snapshot Isolation

Snapshot isolation (SI) [21, 6] is a widely applied multi-versioning concurrency control protocol. Under SI each transaction T sees a private snapshot of the database containing changes of transactions that have committed before T started and T's own changes. Using SI, reads do never block concurrent reads and writes, because each transaction sees a consistent version of the database as of its start (and, as mentioned before, its own changes). To support such snapshots, old versions of tuples cannot be removed 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 this transaction will see (its snapshot). Whether a certain version of a tuple is visible to a transaction can be checked by comparing its timestamp (if it is a committed tuple version) with the transaction's timestamp.

Concurrent writes are allowed under SI. However, if several concurrent transactions have written the same data item, only one of them will be allowed to commit. There are several ways how to enforce this constraint. One option is to check at commit time for transaction T whether any concurrent transaction has an overlapping write set (has modified a data item written by transaction T). If this is the case, all but one of the transactions writing data item d have to be aborted. Under the *First Committer Wins* (*FCW*) rule, the transaction writing data item d which first tries to commit would be allowed to commit and all other transactions writing data item d would be aborted. Under the *First Updater Wins* (*FUW*) rule, the first transaction updating d is allowed to commit. No matter which of the two rules is applied, checking the rule requires maintaining of a write set for each transaction in the system. Implementations of SI (e.g., Oracle or PostgreSQL) do not apply these checks directly, but instead use write locks that are held until transaction commit. A transaction T waiting for a write lock is aborted if the transaction T' holding the write lock commits (and is allowed to continue if T' aborts).

SI does not guarantee serializability.[3]. However, there exist serializable extensions of SI that have been recently implemented in database engines [31, 34, 11]. Under serializable snapshot isolation (SSI), transactions are aborted if their execution may lead to concurrency anomalies. Since SI and SSI only differ in the fact that SSI aborts some transactions that would be allowed to commit under SI, our techniques for computing the provenance of SI transactions can be readily applied to databases which use SSI.

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

Figure 4: Example Semiring

3.2 Read Committed Snapshot Isolation

Database systems implementing SI as a concurrency control protocol typically use a variant of SI with statement-level snapshots as a substitute for the standard `READ COMMITTED` isolation level. We refer to this protocol as *read committed snapshot isolation* (*RC-SI*). Under RC-SI each statement of a transaction sees changes of transaction that committed before the statement was executed. In contrast to SI, transactions waiting for a write lock are allowed to resume their operation once the lock is released no matter whether the transaction holding this lock committed or aborted. The exact details of the implementation differ from system to system. For instance, Oracle restarts an update that had to wait for a write lock to guarantee that the updates sees a consistant snapshot of the database. In contrast, in PostgreSQL the update is resumed after the lock it is waiting for is released. In this paper we assume the RC-SI semantics implemented by Oracle.

3.3 The Semiring-Annotation Framework

Green et al. have introduced the semiring annotation in their seminal paper [20]. In this framework [24] relations are annotated with elements from a commutative semiring $\mathcal{K}$. Such relations are called $\mathcal{K}$-relations. Formally, a $\mathcal{K}$-relation R is a (complete) function that maps tuples to elements from $\mathcal{K}$ with the convention that tuples mapped to the $0_\mathcal{K}$, 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. Green et al. have demonstrated that $\mathcal{K}$-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 of a join was produced from two input tuples annotated with k and k', then the result tuple t would be annotated with $k \times_\mathcal{K} k'$.

The semiring of provenance polynomials($\mathbb{N}[X]$) is the most

[1]Note that the "transactions" studied by Archer et al. [5] are sequences of update operations and not concurrent database transactions.

[2]Newer versions of PostgreSQL implement a serializable variant of snapshot isolation called serializable snapshot isolation (SSI). See Section 3.1.

[3]Note that isolation level `SERIALIZABLE` corresponds to SI for Oracle and older versions of PostgreSQL.

general form of semiring annotation. The elements of this semiring are polynomials over a set of variables X which represent base tuples in the database. Usually, the assumption is that every tuple in a database instance is annotated by a unique variable $x \in X$. The provenance polynomial semiring 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 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 $\mathcal{K}$. Figure 4 shows some example semirings and the extensions of the relational model encoded by these semirings. For example, the semiring $\mathbb{B}$ consisting of the elements *true* and *false* using $\vee$ as addition $\wedge$ as multiplication corresponds to standard relational set semantics. The semiring $\mathbb{N}$ of the set of natural numbers with standard arithmetical operators 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 were used to derive t. The semiring over the powerset of tuples in an database instance (represented as variables X) using set union for both addition and multiplication corresponds to Lineage [12].[4]

Example 2. *Consider the $\mathbb{N}[X]$-relation R shown below and the result of evaluating the query $Q = \Pi_A(R) \times \Pi_B(R)$ over this relation. For instance, the provenance polynomial for the first result tuple records that this tuple has been produced by joining x_1 with itself ($x_1 \times x_1$) and by joining x_1 with x_2 ($x_1 \times x_2$). By mapping x_1 and x_2 to true and interpreting $+$ as $\vee$ and $\times$ as $\wedge$ we get an $\mathbb{B}$-annotation true indicating that this tuple is in the result under set semantics. By mapping x_1 and x_2 to $1 \in \mathbb{N}$ and evaluating the resulting expression we get $1 \times 1 + 1 \times 1 = 2$, the multiplicity of the tuple if Q would have been evaluated under bag semantics. Finally, by mapping x_1 to $\{x_1\}$ and x_2 to $\{x_2\}$ and interpreting the expression using the lineage semiring we get $\{x_1, x_2\}$, the Lineage of the first result tuple.*

R

	A	*B*
x_1	*1*	*1*
x_2	*2*	*1*

Result

	A	*B*
$x_1 \times x_1 + x_1 \times x_2$	*1*	*1*
$x_2 \times x_1 + x_2 \times x_2$	*2*	*1*

Provenance polynomials are considered the most general form of annotation in the semiring framework, because any valuation $\nu : X \to K$ of variables in X to elements from a semiring $\mathcal{K}$ can be lifted to a semiring homomorphism $Eval_\nu : \mathbb{N}[X] \to \mathcal{K}$. Semiring homomorphisms commute with queries. This means that any type of semiring annotation can be computed from the provenance polynomial of a query result. For instance, given a query result relation with $\mathbb{N}[X]$ annotations we can compute the Lineage of the query results or bag semantics multiplicities as illustrated in the example above.

4. MULTI-VERSION PROVENANCE MODEL

We extend $\mathcal{K}$-relations with multi-version annotations (functions from $K \to K$) that enable us to annotate parts of the provenance to indicate how it was derived by an update operation. We then define update operations and a snapshot isolation compatible transactional semantics for these multiversion $\mathcal{K}$-relations.

Definition 1 (Multi-version Semirings). *Let $\mathcal{K} = (K, +_K, \times_K, 0_K, 1_K)$ be a commutative semiring. The set $\mathbb{A}_\mathcal{K}$ of version annotations for $\mathcal{K}$ is a set of abstract function symbols defined as follows. Let T be a transaction and ν be a version. Then $\mathbb{A}_\mathcal{K}$ includes the following functions:*

$$ {}^T_I\tau_\nu, {}^T_U\tau_\nu, {}^T_D\tau_\nu $$

The multiversion semiring (MV-semiring) $\mathcal{K}^\nu$ for a semiring $\mathcal{K} = (K, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K})$ is the structure

$$ (K^\nu, +_\mathcal{K}, \times_\mathcal{K}, 0_\mathcal{K}, 1_\mathcal{K}, \mathbb{A}_\mathcal{K}) $$

of expressions over K and version annotations from $\mathbb{A}_\mathcal{K}$ of the following form. If $k \in K$ then k is in K^ν. If k and k' are expressions in K^ν and $\tau \in \mathbb{A}_\mathcal{K}$ is a version annotation, then the following expressions are also elements of K^ν:

$$ k +_K k' \mid k \times_K k' \mid \tau(k) $$

The version annotations we have introduced represent the application of update operations. For example, if a tuple is annotated with ${}^T_U\tau_\nu(k)$, then this tuple was produced by an update (U) executed by transaction T at database version ν and was derived from a tuple annotated with k. Similarly, ${}_I\tau$ and ${}_D\tau$ represent insert and delete operations.

Definition 2 ($\mathcal{K}^\nu$-relation). *Let $\mathbb{D}$ be a universal domain of values and $\mathcal{K}$ a semiring. An n-nary $\mathcal{K}^\nu$-relation R is a function: $\mathbb{D}^n \to K^\nu$ that maps each tuple $t \in \mathbb{D}^n$ to an annotation from K^ν. A $\mathcal{K}^\nu$-database is a set of $\mathcal{K}^\nu$-relations. Similar to $\mathcal{K}$-relations we require that the support of a $\mathcal{K}^\nu$-relation (tuples mapped to elements other than $0_\mathcal{K}$) is finite.*[5]

The unversioning operator Unv transforms a $\mathcal{K}^\nu$-relation into a corresponding $\mathcal{K}$-relation by interpreting the version annotations as functions $K \to K$. Insert and update annotations are mapped to identity functions on elements from K. The deletion annotation maps all elements k to the $0_\mathcal{K}$, the 0-element of $\mathcal{K}$.

Definition 3 (Unversioning). *Let R be a $\mathcal{K}^\nu$-relation. The unversioning operation Unv(R): $\mathcal{K}^\nu$-relation $\to$ $\mathcal{K}$-relation maps R onto a corresponding $\mathcal{K}$-relation by interpreting the abstract versioning functions as follows.*

$$ {}^T_{I/U}\tau_\nu(k) = k $$
$$ {}^T_D\tau_\nu(k) = 0_\mathcal{K} $$

If the unversion operator is applied to a $\mathcal{K}^\nu$-database D, then the result is defined as applying the operator to each relation in D.

Example 3. *Reconsider our running example. Assume that the original state of the database was produced by a transaction T_0 that inserted all data simultaneously at time 1. Figure 7 shows how, under this assumption, the initial state of the relations before the executions of transactions T_1 and T_2 would be represented as an $\mathbb{N}[X]^\nu$-database (the*

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

[5]That means relations are finite.

multiversion version of provenance polynomials). In this $\mathbb{N}[X]^\nu$-database all tuples are annotated with a single variable enclosed in a version annotation representing the fact that the tuple got inserted by transaction T at time 1. To derive the bag semantics version of these relations we would replace each variable with an element from $\mathbb{N}$ and then apply the unversioning operator to get back an $\mathbb{N}$-relation (the semiring encoding bag semantics relations). In the running example, each tuple appears exactly once. Thus, each variable would be replace by 1 and, after unversioning, each tuple is annotated with 1.

4.1 Queries

The positive relational algebra over $\mathcal{K}^\nu$-relations is defined in the same ways as for $\mathcal{K}$-relations. For sake of completeness, we repeat these definitions 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 an condition θ and tuple t, $\theta(t)$ denotes a functions that returns $1_\mathcal{K}$ if $t \models \theta$ and $0_\mathcal{K}$ otherwise. We add one additional operator $\{t \to k\}$ that creates a singleton relation containing a the tuple t annotated with k. Note that this a generalization of the empty relation operator introduced in the original work on $\mathcal{K}$-relations [24].

Definition 4 ($\mathcal{RA}^+$ on $\mathcal{K}^\nu$-relations). *Let R, S denote relations and t, u denote tuples, and k be an element from K^ν. We use $t.A$ to denote the projection of tuple t on expressions A and $t[R]$ to denote the projection of tuple t on the attributes from relation R. The operators of the positive relational algebra on $\mathcal{K}^\nu$-relations are defined as follows:*

$$\Pi_A(R)(t) = \sum_{u:u.A=t} R(u)$$
$$\sigma_\theta(R)(t) = R(t) \times \theta(t)$$
$$(R \bowtie S)(t) = R(t[R]) \times S(t[S])$$
$$(R \cup S)(t) = R(t) + S(t)$$
$$\{t' \to k\}(t) = \begin{cases} k & \text{if } t = t' \\ 0_\mathcal{K} & \text{else} \end{cases}$$

We use $\emptyset$ as a short cut for the empty $\mathcal{K}^\nu$-relation, i.e., $\emptyset(t) = 0_\mathcal{K}$ for all t.

An important property of our model is that queries can be executed over the multi-version annotation model and the result of the query can be interpreted as a standard $\mathcal{K}$-relation. That is the unversioning operator commutes with queries. The fundamental property of the semiring framework still hold for $\mathcal{K}^\nu$-relations. That is the $\mathcal{K}^\nu$ version of provenance polynomials generalizes all other $\mathcal{K}^\nu$ semirings and translating from provenance polynomials to another semiring $\mathcal{K}$ using an assignment of elements for $\mathcal{K}$ to each variable in $\mathbb{N}[X]$-relation is a semiring homomorphism and commutes with queries. Practically, this means that from the $\mathbb{N}[X]^\nu$-provenance of query results one can derive the query results in any other $\mathcal{K}^\nu$-semiring.

Theorem 1 (Unv Commutes With Queries). *Let D be a $\mathcal{K}^\nu$-database and $Q(D)$ a query.*

$$Q(\textsc{Unv}(D)) = \textsc{Unv}(Q(D))$$

Proof. Trivial induction over the structure of $\mathcal{K}^\nu$ expressions and algebra operators using the fact that Unv maps version annotations to identify functions and functions that return 0 for all inputs. □

4.2 Update Operations

We now define an update language for $\mathcal{K}^\nu$-relations. The annotation of a tuple t in a $\mathcal{K}^\nu$-relation records the different ways of how tuple t has been produced. For example, consider that a tuple t was inserted into a relation R by transaction T at version ν. Afterwards, transaction T executed an update operation at time ν' that modified the values of a tuple t' to match t. Then in the updated version of R, t would be annotated with ${}^T_I\tau_\nu(1_\mathcal{K}) + {}^T_U\tau_{\nu'}(k')$ where k' denotes the original annotation of t'.

In the following it will be helpful to think of an element $k \in \mathcal{K}^\nu$ as a sum $k_1 + \ldots + k_n$ and introduce notations for accessing particular elements from such a sum. We use $n(k)$ to denote the number summands in this representation of an $\mathcal{K}^\nu$ element k and $k[i]$ to denote the ith element in the sum representation of k.

We introduce three update operations for our model. For each update operation we consider it to be executed at a time ν as part of a transaction T. Each update operations takes as input a $\mathcal{K}^\nu$-relation R and returns the updated version of this $\mathcal{K}^\nu$-relation.

An insertion $\mathcal{I}[Q, T, \nu](R)$ inserts the result of query Q into relation R. Note that this operation can be used to express both SQL style `INSERT INTO R VALUE (...)` (by using the singleton operator in Q) and `INSERT INTO R (SELECT ...)` statements. Newly inserted tuples are wrapped in 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 version annotation. Recall that the interpretation of a delete annotation maps all inputs to 0_K. Thus, deleted tuples are removed when R is mapped to the corresponding K-relation.

Definition 5 (Update Operations). *Let R be an $\mathcal{K}^\nu$-relation. We define three types of update operations (insert, delete, update) and a commit operation. We use $\nu(u)$ to denote the version at which an update was executed.*

$$\mathcal{U}[\theta, A, T, \nu](R)(t) = R(t) \times \neg\theta(t) + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u))} {}^T_U\tau_{\nu+1}(R(u)[i]) \times \theta(u)$$
$$\mathcal{I}[Q, T, \nu](R) = R(t) + {}^T_I\tau_{\nu+1}(Q(D)(t))$$
$$\mathcal{D}[\theta, T, \nu](R) = R(t) \times \neg\theta(t) + \sum_{i=0}^{n(R(t))} {}^T_D\tau_{\nu+1}(R(t)[i]) \times \theta(t)$$

Example 4. *Consider an update operation running over a relation from our running example at time 2 as part of an transaction T'. Assume that the initial state of relations is as shown in Figure 7.*

```
UPDATE Schedule SET departureTime='8:00'
WHERE company='Whitedog' AND bnum=13;
```

For brevity we use c, b, and d to represent the company, bnum, and departureTime attributes, respectively. This update operation can be expressed in our model as:

$$\mathcal{U}[c =' Whitedog' \wedge b = 13, d \rightarrow \textit{ 8:00 }, T', 2](Schedule)$$

The first tuple in relation `Schedule` *matches condition of the update and, thus, would be updated. The annotation of resulting tuple t =(Whitedog,13,8:00) contains multiple summands. The first summand $R(t) \times \neg\theta(t)$ would evaluate to 0, because $R(t)$ in the input is 0 (i.e., t was not present in the input). The second part of the annotation of t is a sum over all tuples u that would get updated to t by the update operation. These are all tuples $(Whitebus, 13, X)$ for any X. Except for the first tuple in the input relation, all these tuples are annotated with 0 in the input. Thus, $R(u)[0] = 0$ for all summands except for $X = 8:15$ which is annotated with ${}_{I}^{T_0}\tau_1(s_1)$ in the input. This summand would be wrapped in a version annotation. Ignoring summands that evaluate to zero the final annotation for t is ${}_{U}^{T'}\tau_2({}_{I}^{T_0}\tau_1(s_1))$. From this annotation we can determine that t was produced by updating a tuple that was inserted by transaction T_0. The annotation for a tuple which was not updated is the same as in the input. For example, the annotation for the second tuple t' from the input is ${}_{I}^{T_0}\tau_1(s_2) \times 1 + \sum_{\emptyset} = {}_{I}^{T_0}\tau_0(s_2)$, because tuple t' does not fulfill the update's condition and there exist no tuple u such that result of applying the update to u would be t' (the update sets departure time to 8:00 and the departure time of t' is 16:00).*

Having defined the semantics of updates, we now demonstrate that $\mathbb{N}[X]^{\nu}$ the multi-version semiring is the most general $\mathcal{K}^{\nu}$-semiring and, thus, the best choice to represent update provenance. Furthermore, we need to check that $\mathbb{N}^{\nu}$ and $\mathbb{B}^{\nu}$ correspond to standard relation updates under set and bag semantics. That is, our model is a strict generalization of set and bag semantics.

PROPOSITION 4.1 (FUNDAMENTAL PROPERTY). *Let ν be a valuation from variables X to elements of a MV-semiring $\mathcal{K}^{\nu}$. Then $Eval_{\nu}$ and* UNV *commute with updates.*

4.3 Transactions and Histories

We now build upon the update operations to define semantics for transactional histories for $\mathcal{K}^{\nu}$-databases. We consider two concurrency control protocols: SI and RC-SI. For simplicity we will limit the discussion to histories that start from an empty database. However, the results naturally extend to histories that are applied to an existing $\mathcal{K}^{\nu}$-database (with an associated history). One important property of such histories is that they completely determine what we refer to as a *historic database*. A historic database $D(H, CC)$ for a history H and concurrency control protocol CC encodes the versions of D seen at each point in time by the transactions in the history H. Here it will be useful to model transactions as a sequence of relational update operations instead of read and write operations on data items. Note that we do not consider transaction failures and partially executed transactions. Theoretically, aborts could be included in the model, but to simplify the discussion, we will refrain from modelling aborted transactions and partial transactions in histories. This is not a real drawback, because the mechanisms introduced in this paper are used to retroactively reconstruct provenance and, thus,only apply to finished transactions.

DEFINITION 6 (HISTORY). *A transaction $T = \{u_1, \ldots, u_n, c\}$ is a sequence of update operations followed by a commit operation (c). We require that the order relation $<_{\nu}$: $\{(u_i, u_j) \mid \nu(u_i) < \nu(u_j)\}$ is a total order for the statements of T. Transactions implicitly start with their first update operation. 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 so that no two operations in H were executed at the same time ($\forall u, u' \in H : \nu(u) = \nu(u') \rightarrow u = u'$).*

Note that in our model updates are explicitly part of a transaction and we record at which point in time (ν) an update has been executed. This will allow us to determine the state of the database seen by each update of a transaction. Recall that we assume that the initial state of the database contains only $\emptyset$ relations.

EXAMPLE 5. *Reconsider the running example from the introduction. Again, we abbreviate attributes as c, b, d for relation* `Schedule` *and c, n, p, f, t for relation* `Bus`*. The history for this example is:*

$$T_1 = \{\mathcal{I}[\{(Greyhound, 5, 96, Chicago, NY) \rightarrow b_4\}, T_1, 10](Bus)\}$$
$$T_2 = \{\mathcal{D}[c =' Whitedog' \wedge b = 13, T_2, 12](Schedule),$$
$$\mathcal{I}[\Pi_{c,n,'20:15'}(\sigma_{c='Greyhound'}(Bus)), T_2, 13](Schedule),$$
$$\mathcal{U}[c =' Picobus' \wedge b = 2, d \rightarrow \textit{ 10:30 }, T_2, 14](Schedule)\}$$

4.4 Historic Database

If we fix a concurrency control protocol CC, then a history H uniquely defines a multi-versioned $\mathcal{K}^{\nu}$-database D that models the annotated database seen within the context of each transaction $T \in H$ at a certain version ν. We call the collection of states of D for all T and ν pairs a *historic database*. The exact content of such a history database depends on the concurrency control protocol. In the following we define historic databases for SI and RC-SI.

4.4.1 Snapshot Isolation

We first define the historic database for the snapshot isolation protocol based on the update operations defined for our model.

DEFINITION 7 (SI HISTORIC DATABASE). *Let H be a history over a database D. The historic database $D(H, SI)$ of D based on H is a set of historic relations. An n-nary historic relation R^{ν} is a function $\mathbb{D}^n \times \mathbb{T} \times \mathbb{V} \rightarrow \mathcal{K}^{\nu}$. We use $R[T, \nu]$ to denote the restriction of R^{ν} generated by fixing the last two parameters to T and ν and apply the same notation also for databases. Furthermore, we use $H(D)$ as a shortcut for $D[End(T)]$ where T is the transaction from H that committed last. R^{ν} is defined in Figure 5.*

Figure 5a shows the recursive definition of $R[T, \nu]$. Per convention we define $R[T, \nu] = \emptyset$ if $\nu < Start(T)$. For $\nu = Start(T)$, relation R contains all changes of transactions that committed before $Start(T)$ as required by snapshot isolation. We use $R[\nu]$ to denote this version of R and will discuss its definition below. If T does contain an update

(a) Historic Relation

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

(b) Committed Tuple Versions at ν

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

(c) Valid Tuple Versions from Transaction T at ν

$$\textsc{validAt}(T,t,k,\nu) = \begin{cases} 1_K & \text{if } k = {}^{T}_{I/U/D}\tau_{\nu'}(k') \wedge (\neg\exists T \neq T' : End(T') \leq \nu \wedge \textsc{updated}(T',t,k)) \\ 0_K & \text{else} \end{cases}$$

(d) Tuple Versions Updated By Transaction T

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

Figure 5: SI Historic Database Definition

(a) Historic Relation

$$R[T,\nu] = \begin{cases} \emptyset & \text{if } \nu < Start(T) \\ R[T,\nu-1] & \text{if } Start(T) \leq \nu < End(T) \wedge \neg\exists u \in T : \nu(u) = \nu - 1 \\ u(R_T[\nu-1]) & \text{if } \exists u \in T : \nu(u) = \nu - 1 \\ R_T[\nu] & \text{if } \nu = End(T) \wedge \neg\exists u \in T : \nu(u) = \nu - 1 \end{cases}$$

(b) Committed Tuple Versions and Updates of T

$$R_T[\nu](t) = \sum_{T' \in H \wedge (End(T') \leq \nu \vee T' = T)} \sum_{i=0}^{n(R[T',\nu](t))} R[T',\nu](t)[i] \times \textsc{validAt}(T,T',t,R[T',\nu](t)[i],\nu)$$

(c) Valid Tuple Versions from T' at ν with Respect to T

$$\textsc{validAt}(T,T',t,k,\nu) = \begin{cases} 1_K & \text{if } k = {}^{T'}_{I/U}\tau_{\nu'}(k') \wedge (\neg\exists T'' : End(T'') \leq \nu \wedge \textsc{updated}(T'',t,k,\nu) \wedge T' \neq T'') \\ & \quad \wedge \neg\textsc{updated}(T,t,k,\nu) \\ 1_K & \text{if } k = {}^{T}_{I/U}\tau_{\nu'}(k') \wedge \neg\textsc{updated}(T,t,k,\nu) \\ 0_K & \text{else} \end{cases}$$

(d) Tuple Versions Updated by Transaction T before ν

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

Figure 6: RC-SI Historic Database Definition

that was executed at version $\nu - 1$ then $R[T, \nu]$ is the result of applying the update to $R[T, \nu - 1]$. In case transaction T did not execute an update at $\nu - 1$, $R[T, \nu]$ is the same as $R[T, \nu - 1]$.

Relation $R[\nu]$, the version of R^ν that contains all committed changes of transactions executed before T is define as follow. Each tuple is annotated with the sum of all annotations on t produced by transaction that committed before or at ν. However, some summands in such annotations may not be valid at version ν anymore, because they have been overwritten by other transactions. We use $\textsc{validAt}(T, t, k, \nu)$ to denote a function that returns 1 if annotation k on tuple t is valid at version ν within transaction T and 0 otherwise. A summand k is valid within an annotation for tuple t at transaction T and version ν if 1) it has not been overwritten by another transaction that committed before ν and 2) was produced by T before ν (is wrapped in a version annotation corresponding to an update of transaction T). For the first condition we define a predicate $\textsc{updated}(T, t, k)$ which is true if transaction T has overwritten summand (tuple version) k on tuple t.

A transaction T has overwritten a summand k in an annotation of a tuple t if there exist an update u (update or delete) within the transaction that has updated tuple t into tuple t'. Thus, there has to exist i and j so that a summand $R[T, \nu(u)][i] = k$ is in the annotation on t before the update and after the update the annotation on tuple t' contains a summand $R[T, \nu(u) + 1][j] = {}^{T}_{I/U}\tau_{\nu(u)+1}(k)$.

Example 6. *Figure 7 shows the annotations on the example database before and after the execution of transactions T_1 and T_2. Consider the first statement in transaction T_2 is a delete executed at version 12. According to the definition of a historic database the state of relation schedule after the deletion ($Schedule[T_2, 13]$) is the result of applying the deletion to $Schedule[T_2, 12]$. Since $Start(T) = 2$ this version of schedule contains all updates of transaction committed before version 12. Thus, $Schedule[T_2, 12]$ is the version shown on the top of Figure 7. Now consider the first tuple $t =$ (Whitedog,13,8:15) in $Schedule[T_2, 12]$ which is annotated with ${}^{T_0}_{I}\tau_1(s_1)$ indicating that this tuple was produced by an insertion of transaction T_0. After applying the deletion the annotation on this tuple is:*

$$ {}^{T_0}_{I}\tau_1(s_1) \times 0 + {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) \times 1 = {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) $$

The original annotation is multiplied by 0 and an additional summand has been added modelling that this tuple was deleted by transaction T_2. For illustrative purposes we will keep the summand wrapped in a deletion annotation even though it evaluates to 0 if this annotation in interpreted.

The next update from transaction T_2 is an insert. When evaluating the insertion query

$$ \Pi_{c,n,'20:15'}(\sigma_{c='Greyhound'}(Bus)) $$

over $Bus[T_2, 13]$ (which is equal to the version shown on the top of Figure 7) the annotation on t is 0. Thus, the insertion adds an additional summand ${}^{T_2}_{I}\tau_{14}(0)$ to the annotation of t resulting in:

$$ {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) + {}^{T_2}_{I}\tau_{14}(0) = {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) $$

The last operation in transaction T_2 is an update. Since the condition of the update evaluates to false for t and there exist no tuple u so that the result of applying the update to would be t (this update sets the departure time to 10:30), the resulting annotation is:

$$ \begin{aligned} & Schedule[T_2, 14] \times \neg\theta(t) \\ & + \sum_{u:u.A=t} \sum_{i=0}^{n(Schedule[T_2,14](u))} Schedule[T_2, 14](u)[i] \times \theta(u) \\ = & {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) \times 1 + \sum_{\emptyset} \\ = & {}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1)) \end{aligned} $$

The second tuple in the schedule relation has not changed by any transaction so its annotation is same as the input.

The third tuple and fourth tuple do not fulfill the condition of the deletion and are annotated with 0 in the result of the insertion query. Thus, if we remove summands that evaluate to 0 the annotations on these tuples before execution of the final update of transaction T_2 are the same as before this transaction started. Let us use t_3 and t_4 to denote these tuples. Tuple t_3 fulfills the condition of the update and, thus, is annotated with:

$$ \begin{aligned} & {}^{T_0}_{I}\tau_1(s_3) \times 0 \\ & + \sum_{u:u.A=t_3} \sum_{i=0}^{n(Schedule[T_2,14](u))} Schedule[T_2, 14](u)[i] \times \theta(u) \end{aligned} $$

The sum ranges over all tuples u which after applying the update are equal to t_3. These are all $u : (Picobus, 2, X)$ for any valid time X. Any such u fulfills the condition of the update ($c =' Picobus' \wedge b = 2$), i.e., $\theta(u) = 1$. With the exception of t_3 and t_4 all such tuples are annotated with 0 in the input. Thus, effectively the sum ranges only over the input annotations for t_3 and t_4 which each are annotated with one summand. The resulting annotation is:

$$ \begin{aligned} & {}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_3)) \times 1 + {}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_4)) \times 1 \\ = & {}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_3)) + {}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_4)) \end{aligned} $$

As discussed above tuple t_4 fulfills the condition of the update, but there exist no u such that $u.A = t_4$. Thus, the resulting annotation for t_4 is:

$$ {}^{T_0}_{I}\tau_0(s_3) \times 0 + \sum_{\emptyset} = 0 $$

4.4.2 Read Committed Snapshot Isolation

Under read committed snapshot isolation (RC-SI) [6] each update u within a transaction sees changes made by previous updates of the same transaction and changes of transactions that have committed before u was executed. Recall that under RC-SI each update of a transaction sees changes of transactions which committed before the update was executed. Thus, we have to adapt the definitions of $R[T, \nu]$ to include changes of concurrent transactions. We define $R_T[\nu]$ which denotes the version of relation R seen by transaction T at version ν and thus version includes both past changes of T and annotations created by transactions that committed before ν. Consider the range of the outer sum in the definition of $R_T[\nu]$. This sum ranges over all transactions that have committed before ν and T itself.

[6] Recall that this corresponds to isolation level `READ COMMITTED` in databases such as Oracle or Postgres

Bus

	company	number	price	fromCity	toCity
${}^{T_0}_{I}\tau_1(b_1)$	Whitedog	13	210	New York	Chicago
${}^{T_0}_{I}\tau_1(b_2)$	Whitedog	15	140	Seattle	Chicago
${}^{T_0}_{I}\tau_1(b_3)$	Picobus	2	65	Chicago	Schaumburg

Schedule

	company	bnum	departureTime
${}^{T_0}_{I}\tau_1(s_1)$	Whitedog	13	08:15
${}^{T_0}_{I}\tau_1(s_2)$	Whitedog	15	16:00
${}^{T_0}_{I}\tau_1(s_3)$	Picobus	2	10:30
${}^{T_0}_{I}\tau_1(s_4)$	Picobus	2	12:30

Bus after Transaction 1

	company	number	price	fromCity	toCity
${}^{T_0}_{I}\tau_1(b_1)$	Whitedog	13	210	New York	Chicago
${}^{T_0}_{I}\tau_1(b_2)$	Whitedog	15	140	Seattle	Chicago
${}^{T_0}_{I}\tau_1(b_3)$	Picobus	2	65	Chicago	Schaumburg
${}^{T_1}_{I}\tau_{11}(b_4)$	Greyhound	5	96	Chicago	New York

Schedule after Transaction 2

	company	bnum	departureTime
${}^{T_2}_{D}\tau_{13}({}^{T_0}_{I}\tau_1(s_1))$	Whitedog	13	8:15
${}^{T_0}_{I}\tau_1(s_2)$	Whitedog	15	16:00
${}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_3)) + {}^{T_2}_{U}\tau_{15}({}^{T_0}_{I}\tau_1(s_4))$	Picobus	2	10:30
${}^{T_2}_{I}\tau_{14}({}^{T_1}_{I}\tau_{11}(b_4))$	Greyhound	5	20:15

Figure 7: Historical Database for the Running Example Transactions ($H = \{T_1, T_2\}$)

Again, we define a relation VALIDAT which determines whether a summand in an annotation is valid at time ν. However, now this validity is specific to a transaction T. A summand is valid if it was produced by T itself (second row in the case distinction). In addition, it is also valid if it was produced by transaction T' and no other transaction that has committed or is equal to T has overwritten this summand. The definition of UPDATED is the same as under SI.

DEFINITION 8 (RC-SI HISTORIC DATABASE). *Let H be a history over a database D. The history database $D(H, RC-SI)$ of D based on H is a set of historic relations. R^ν for R in D is defined in Figure 6.*

Similar to how *Eval* and UNV commute with queries and updates, the same also holds for transactions.

PROPOSITION 4.2 (*Eval* COMMUTES WITH HISTORIES). *Let ν be a valuation from variables X to elements of a MV-semiring $\mathcal{K}^\nu$. Then $Eval_\nu$ and* UNV *commute with histories.*

This theorem shows that we can evaluate histories using the $\mathbb{N}[X]^\nu$ MV-semiring and derive the result of applying a history in any MV-semiring from $H(D)$ evaluated in $\mathbb{N}[X]^\nu$. However, this does not necessarily imply that our approach correctly generalizes SI and RC-SI for set and bag semantics. It remains to show that $H(D)$ if evaluated under $\mathbb{B}^\nu$ and $\mathbb{N}^\nu$, respectively, corresponds to standard set and bag semantics evaluation of SI and RC-SI histories.

THEOREM 2 (SI AND SI-RC CORRECTNESS). *Let $D_\mathbb{B}$ and $D_\mathbb{N}$ be the result of evaluating a history H under standard bag semantics and set semantics using SI, receptively. Then $D_\mathbb{B} = H(D)$ if evaluated in $\mathbb{B}$ and $D_\mathbb{N} = H(D)$ if evaluated in $\mathbb{B}$. The same holds for RC-SI.*

PROOF. We have already shown that updates executed under $\mathcal{K}^\nu$ semantics correspond to set and bag semantics if $\mathcal{K}^\nu = \mathbb{B}$ and $\mathcal{K}^\nu = \mathbb{N}$, respectively. Thus, it remains to show that our definition of $H(D)$ obeys the visibility rules of SI and RC-SI, respectively.

SI: Under SI each update of a transaction T sees changes by transaction that committed before T started and updates of previous statements of T. For a sequence of statements within T the correctness follows from the correctness for single updates. It remains to show that $R[\nu]$ is correct. $R[\nu]$ has to contains all changes to relation R of transactions that committed before ν. Obviously, this set contains all tuple versions created by transactions that have committed before ν and that have not been overwritten by preceding transactions which also committed before ν. We prove that this set corresponds to $R[\nu]$ by induction over the number of transaction that have committed before ν.

Induction Start: Per convention the initial state of the database is empty and, thus, the conjecture trivially holds. If only a single transaction T' committed before ν then $R[\nu]$ is the result of a single sequence update operations (the operations of transaction T') and the correctness follows from

the correctness for simple updates.

Induction Step: Assume that the conjecture holds for i transaction that committed before ν. We need to show that the same applied if $i+1$ transaction have committed before ν. WLOG assume that the new transaction T_{i+1} was the last to commit. Given that T_{i+1} committed last, all summands created by this transaction should be in $R[\nu]$. This is the case, because $End(T_{i+1}) \leq \nu$ and, thus, T_{i+1} is considered in the outer sum of $R[\nu]$. However, we need to show that $\textsc{validAt}(T_{i+1}, t, k, \nu)$ evaluates to 1 for any t and k. For any summand created by T_{i+1} we know from the definition of $R[\nu]$ that this summand was not visible to any other transaction that committed before ν. This follows from the fact that T_{i+1} was the last transaction to commit. For any other transaction T_j that committed before T_j, $R[Start(T)]$ the outer sum does not range over T_{i+1}, because $End(T_{i+1}) > Start(T)$. Furthermore, we need to show that any summand created by a transaction T_j that was overwritten by T_{i+1} is not present in $R[\nu]$. Let k be such an summand in an annotation for a tuple t created by transaction T_j with $End(T_j) < Start(T)$. Note that we do not need to consider case $End(T_j) > Start(T)$, because annotations created by such a transaction are not visible to T_{i+1}. WLOG assume that transaction T_{i+1} did overwrite k using an update operation executed at $\nu' < \nu$. No matter what type of update is applied (update or delete), the result will include a summand wrapping k into a version annotation (this follows from the definition of update operations). Hence, in $\textsc{validAt}(T_j, t, k, \nu)$ the not exists condition evaluates to false for $T' = T_{i+1}$, because $End(T_{i+1}) < \nu$ and $\textsc{updated}(T_{i+1}, t, k)$ is true. This proves the induction step.
RC-SI: The proof for RC-SI is similar in nature with the exception that we needs to range over operations of transactions and the visibility rules are slightly different. □

4.4.3 $\mathcal{K}^{\nu}$ Database Size

We have introduced a representation system for the provenance of transactional histories and proven several important properties. However, so far we have not discussed how the size (combined size of non zero annotations) of a historic database is related to the size of the history. Here we are only interested in parts of an annotation that does not evaluate to 0.

Inspecting the definition of update operations, it should be immediately clear that with the exception of the insert statement, all updates do not generate additional summands in the annotation. Even though an update $\mathcal{U}[\theta, A, T, \nu](R)$ may combine summands from more than one input tuple, this does not increase the number of summands, because these summands will be effectively removed from these input tuples (will evaluate to 0 in the result of the update). If we limit insertion queries to singleton relations, then each insertion adds at most one summand to one annotation in the database. The total number of summands in the database is then bound by the number of insert statements in the history. Update and delete statements wrap a subset of the existing summands in version annotations and, thus, may increase the size of the annotation by at most the current number of summands in all annotations. We can deduce that the total annotation size in the final version of a historic database produced under this restriction of inserts has to be less than the square of the number of updates. Compared to the maximum number of tuples created by set and bag semantics for the same history the size may be increased by a factor equal to the number of updates. This is not surprising, because each summand (corresponding to at least one tuple under bag semantics) may be wrapped in up to number of update version annotations.

This result is important because it suggests that it may be possible to efficiently compute such a representation. In fact, we will demonstrate in Section 6 how an relational encoding of an MV-database and reenactment queries can be used to efficiently implement this approach.

If we lift the restriction on inserts then there exists no bound on the size of the resulting historic database, because an insert query may arbitrarily increase the size of the annotated database. For example, consider a query of the form $R \times \ldots \times R$. Such a query will produce a number of summands that is exponential in the number of crossproducts in the query. However, note that also under bag semantics the number of result tuples of this query will also be exponential in the number of crossproducts.

Theorem 3 (Historic Database Size). *Let H be a history where insert statements only use the constant relation operator in their query Q. Let the size of an historic database be defined as the total number of non zero elements in annotations in the latest version of this database. Let $\#U(H)$ denote the number of updates in history H. The size of $D(H, CC)$ is $O(\#U(H)) \times \#U(H)))$.*

Proof. Since we require that every history starts from a empty database, the initial database has size 0. New summands can only be added by insert operations. Each insert adds exactly one summand wrapped in a version annotation. Thus, for a history H with $n = \#U(H)$ updates, there may exist no more than n summands in all annotations combined. An update operation wraps input annotations in version annotations. However, the total number of summands stays constant, because any summand that is added to an annotation of a tuple t would be removed from its original tuple t'. In the worst case an update affects all summands and, thus, increase the total database size by the total number of summands in its input. Since the total number of summands in the database at any point of the history is bound by n, it follows that each update can not possible increase the size by more than n. Thus, the total database size is bound by $n \times n$. □

4.5 Partial Provenance

For databases with large histories the provenance annotation of a tuple stores its complete derivation history since the origin of the database. This amount of information can be overwhelming to a user and expensive to compute. Thus, it is important to provide a mechanism for limiting provenance information to one update operation, transaction, or a set of transactions. In the $\mathcal{K}^{\nu}$ model this can be achieved in a natural way 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 fresh variables.

Example 7. *Assume that a user is only interested the provenance of transaction T_2 from the running example. The historic database for the example history also encodes the changes applied by transaction T_1. Partial provenance for T_2 can be derived by 1) replacing subexpressions within version annotations for updates that were executed before T_2*

started with fresh variables, 2) remove summands created by updates that were not visible to updates within T_2, and 3) remove summands that were not affected by any update in T_2. In the resulting instance, only tuples affected by T_2 will be part of this partial provenance (i.e., annotated with non-zero annotations). For example, consider the annotation ${}^{T_2}_{I}\tau_{13}({}^{T_1}_{I}\tau_{10}(b_4))$ on tuple (Greyhound,5,20:15) in Schedule$[T_2, End(T_2)]$. To limit this annotation to provenance related to T_2 we would replace ${}^{T_1}_{I}\tau_{10}(b_4)$ with a fresh variable, say x.

Definition 9 (Partial Provenance). *Let T be a transaction in a history H. The partial provenance $D[T]$ of T is a $\mathcal{K}^\nu$-database derived from $D[T, End(T)]$ by applying the following substitution rules to each summand k in an annotation in $D[T, End(T)]$. Let x_{new} denote a fresh variable that does not occur in any annotation in $D[T, End(T)]$.*

$$k \rightsquigarrow \begin{cases} {}^{T}_{I/U/D}\tau_\nu(sub(k')) & \textit{if } k = {}^{T}_{I/U/D}\tau_\nu(k') \\ 0 & \textit{else} \end{cases}$$

$$sub(k) = \begin{cases} k' + k'' & \textit{if } k = k' + k'' \\ k' \times k'' & \textit{if } k = k' \times k'' \\ k & \textit{if } k = 0 \vee k = 1 \\ {}^{T}_{I/U/D}\tau_\nu(sub(k')) & \textit{if } k = {}^{T}_{I/U/D}\tau_\nu(k') \\ x_{new} & \textit{if } k = {}^{T'}_{I/U/D}\tau_\nu(k') \wedge T' \neq T \end{cases}$$

Partial provenance for sets of updates and transactions can be defined in a similar fashion. For reasons of space we omit these definitions here. Partial provenance preserves the beneficial properties of the $\mathcal{K}^\nu$-model as long as in an evaluation the new variables introduced by the substitution are replaced with the result of evaluating the original subexpression.

5. REENACTMENT QUERIES

It is well-known that updates can be expressed as relational algebra expressions that are evaluated over the version of a relation before the update and return the updated relation. For example an update

```
UPDATE R SET a = a + 5 WHERE b = 16;
```

can be expressed as

$$\Pi_{A+5\to A,B}(\sigma_{b=16}(R)) \cup \sigma_{\neg(b=16)}(R)$$

Based on the same idea, we now present a technique called *reenactment queries*, that enables us to reconstruct the provenance of an update u or whole transaction T in a history by executing a reenactment query $Q(u)$ respective $Q(T)$ which is annotation equivalent to u respective T. To be precise $u \equiv_{\mathbb{N}[X]^\nu} Q(u)$, the original update and its reenactment produce the same annotated relation, i.e., have the same result and provenance. As Green demonstrated $Q \equiv_{\mathbb{N}[X]} Q' \Rightarrow Q \equiv_{\mathbb{N}} Q'$ (this result translates to MV-semirings). Thus, if we disregard provenance, then reenactment queries have the important property that they produce the same updated relation as the original update.

We need to extend our query algebra with an annotation operator that adds version annotations, because the operators of $\mathcal{RA}^+$ are not capable of introducing new version annotations which is required for reenacting updates.

Definition 10 (Version Annotation Operator). *The version annotation operator ${}^{T}_{I/U/D}\mathcal{A}_\nu(R)$ 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 ${}^{T}_{I/U/D}\tau_\nu$.*

$${}^{T}_{I/U/D}\mathcal{A}_\nu(R)(t) = \sum_{i=0}^{n(R(t))} {}^{T}_{I/U/D}\tau_\nu(R(t)[i])$$

5.1 Update Reenactment

We first define reenactment for a single update operation u. Such a reenactment query is executed over the historic database seen by u's transaction at the time of the update u ($R[T, \nu(u)]$).

Definition 11 (Update Reenactment). *Let H be a history over a database D and u and update operation in H. The reenactment query $Q(u)$ of u is defined as follows:*

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

An update modifies a relation by applying the expression from A to all tuples that match the update condition θ. All other tuples are kept as is. Thus, we can compute the result of an update as the union between these two sets. An insert statement adds the result of a query to the affected relation. It can be reenacted as the union of the relation before the update and the result of the insertion query. A deletion removes tuples matching the deletion condition (wraps them in deletion annotations). Thus, it can also be expressed as a union between the original tuples that do not match the condition and the deleted versions of tuples that match the condition.

Example 8. *Consider the insert and update operations (let us refer to these operations as u_1 and u_2) from transaction T_2 of the running example shown in Example 5. The reenactment query $Q(u_1)$ for the insert u_1 is:*

$$Schedule[T_2, 13]$$
$$\cup {}^{T_2}_{I}\mathcal{A}_{14}(\Pi_{c,b,'20:15'}(\sigma_{c='Greyhound'}(Bus[T_2,13])))$$

The reenactment query $Q(u_2)$ is:

$${}^{T_2}_{U}\mathcal{A}_{15}(\Pi_{c,b,10:30\to d}(\sigma_{c='Picobus' \wedge b=2}(Schedule[T_2,14])))$$
$$\cup \sigma_{\neg\theta}(Schedule[T_2,14])$$

Theorem 4 (Annotation Equivalence). *Let u be an update statement and $Q(u)$ its reenactment query. Then u and $Q(u)$ are annotation equivalent:*

$$u \equiv_{\mathbb{N}[X]^\nu} Q(u)$$

Proof. Proven by substitution of the definitions of the update operations and query and annotation operators. As an example we show the proof for an update $u = \mathcal{U}[\theta, A, T, \nu](R)$. The reenactment query Qu for u is:

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

We have to show that $u(t) = Qu(t)$ for any tuple $t \in R$. Let $Q' = \Pi_A(\sigma_\theta(R[T,\nu]))$

$$R[T,\nu] \rightsquigarrow \begin{cases} Q^R(\textsc{Last}(R,T,\nu)) & \text{if } \exists u \in T: u(R) \wedge \nu(u) < \nu \\ R[Start(T)] & \text{else} \end{cases}$$

Figure 8: SI Transaction Reenactment Substitution Rules

Applying the definitions of $\mathcal{RA}^+$ we get:

$$Qu(t) = \sum_{i=0}^{n(Q'(u))} {}^T_U\tau_\nu(Q'(u)) + (R(t) \times \neg\theta(t))$$

$$= R(t) \times \neg\theta(t) + \sum_{i=0}^{n(Q'(t))} {}^T_U\tau_\nu(Q'(t)[i])$$

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

$$= R(t) \times \neg\theta(t) + \sum_{i=0}^{n(Q'(t))} {}^T_U\tau_\nu((\sum_{u:u.A=t} R(u) \times \theta(u))[i])$$

Since the annotation operator applies the version annotation to each summand in an annotation we can pull out the inner sum:

$$= R(t) \times \neg\theta(t) + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u)\times\theta(u))} {}^T_U\tau_\nu((R(u) \times \theta(u))[i])$$

Applying the distributivity laws for semirings, we get:

$$= R(t) \times \neg\theta(t) + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u))} {}^T_U\tau_\nu(R(u)[i] \times \theta(u))$$

Since ${}^T_U\tau_\nu$ is interpreted as identify we can pull out the multiplication $\theta(u)$ to get:

$$= R(t) \times \neg\theta(t) + \sum_{u:u.A=t} \sum_{i=0}^{n(R(u))} {}^T_U\tau_\nu(R(u)[i]) \times \theta(u)$$

$$= \mathcal{U}[\theta, A, T, \nu](R)(t)$$

This proves the theorem for updates. The proof for inserts and deletes in analog. □

Based on this theorem, a reenactment query can be used to simulate the effect of any update expressible in our model.

Collary 5.1. *Eval commutes with updates and query processing and the same holds for unversioning. Thus, for any MV-semiring $\mathcal{K}^\nu$ we get:*

$$u \equiv_{\mathcal{K}^\nu} Q(u)$$

$$\textsc{Unv}(u) \equiv_{\mathcal{K}} \textsc{Unv}(Q(u))$$

5.2 Transaction Reenactment

To reenact a transaction we merge the reenactment queries for the updates of the transaction in a way the respects the visibility rules enforced by the concurrency control protocol. Interestingly, the resulting query only accesses committed tuple versions (there is no access to $R[T,\nu]$ if $\nu \neq End(T)$). This will become important in the next section, where we demonstrate how to retroactively compute the provenance of transactions based on reenactment queries, because we want to run such provenance computations over a regular database with time travel support and all system we are aware of only support accessing committed tuple versions.

5.2.1 SI Reenactment

Under snapshot isolation, each update u_i of a transactions sees the version of the database at transaction start plus local modifications of updates u_j of transaction T with $j < i$. Thus, effectively, each update u_i updating a table R is evaluated over the annotated database produced by the most recent update u_j with $j < i$ that updated R. Since we have already proven that $u \equiv_{\mathbb{N}[X]^\nu} Q(u)$, each reference to a relation $R[T,\nu]$ produced by update u_j can be replaced with $Q(u_j)$. Applying this substitution recursively results in a single query $Q^R(T)$ per relation R effected by transaction T. Each such query only references versions of R valid when the transaction started ($R[Start(T)]$). Technically, the reenactment of a transaction is a set of queries. However, abusing terminology we will refer to this set of queries as the reenactment query of the transaction.

Definition 12 (Transaction SI Reenactment). *Let $T = u_1, \ldots u_n, c$ be a transaction in a SI history H. We use $R(T)$ to denote all relations effected by transaction T (updated by at least one update in T) and $\textsc{Last}(R,T,\nu)$ to denote the last update executed before ν in T that updated table R. The reenactment query $Q(T)$ for T is defined as follows:*

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

$$Q^R(T) = Q^R(\textsc{Last}(T, R, End(T)))$$

where transaction update reenactment query $Q^R(u)$ is defined as the result of exhaustively applying the substitution rule shown in Figure 8 to $Q(u)$.

Example 9. *Consider the transaction T_2 from the running example presented in Example 5. Let us refer to the update operations of this transaction as u_1, u_2, and u_3. We use S as a shorthand for relation Schedule and B for relation Bus. We abbreviate attribute names as in previous examples. The reenactment query for T_2 is:*

$$Q^S(T_2) = Q^S(u_3)$$

$$Q^S(u_3) = {}^{T_2}_U\mathcal{A}_{15}(\Pi_{c,b,10:30\to d}(\sigma_{c='Picobus' \wedge b=2}(Q^S(u_2)))) \cup \sigma_{\neg(c='Picobus' \wedge b=2)}(Q^S(u_2))$$

$$Q^S(u_2) = Q^S(u_1) \cup {}^{T_2}_I\mathcal{A}_{14}(\Pi_{c,b,'20:15'}(\sigma_{c='Greyhound'}(B[12])))$$

$$Q^S(u_1) = {}^{T_2}_D\mathcal{A}_{13}(\sigma_{c='Whitedog' \wedge b=13}(S[12])) \cup \sigma_{\neg(c='Whitedog' \wedge b=13)}(S[12])$$

For example, in $Q^S(u_2)$ the access to relation Bus is kept, because there is no update operation in transaction T_2 that updated this relation before u_2 was executed. The same applies to the accesses to relation Schedule in $Q^S(u_1)$. In contrast $S[T_2, 14]$ in $Q^S(u_3)$ is replaced with $Q^S(u_2)$, the query for u_2, the last update before u_3 that updated relation Schedule.

$$R[\nu_b,\nu_e](t) = \sum_{T \in H \wedge End(T) \leq \nu_e} \sum_{i=0}^{n(R[T,\nu])} \text{ENDV}(t, R[T,\nu]) \times \text{VALIDAT}(T,t,k,\nu_b,\nu_e)$$

$$\text{ENDV}(t, {}^{T}_{I/U/D}\tau_{\nu}(k')) = {}^{T}_{I/U/D}\tau_{\nu}^{\nu_e}(k') \text{ with } \nu_e = max(\{\nu' \mid \text{VALIDAT}(T,t,k,\nu')\})$$

$$\text{VALIDAT}(T,t,k,\nu_b,\nu_e) = \exists \nu_b \leq \nu \leq \nu_e : \text{VALIDAT}(T,t,k,\nu)$$

Figure 9: Extended Version Annotations for RC-SI

Having defined reenactment queries for SI transactions we now proceed to prove that these reenactment queries are equivalent to the transaction they are reenacting.

THEOREM 5 (ANNOTATION EQUIVALENCE). *Let T be a SI transaction and $Q(T)$ its reenactment query. Then T and $Q(T)$ are annotation equivalent:*

$$T \equiv_{\mathbb{N}[X]^\nu} Q(T)$$

PROOF. We prove the theorem by induction over the number of updates in transaction T. For simplicity, we assume that transaction T updates a single relation R. The proof can easily be extended for transactions that update multiple relations.

Induction Start: For a transaction with a single update u_1, the theorem follows from the annotation equivalence for update operations.

Induction Step: Assume that we proven that reenactment is annotation equivalent for transactions with up to i update operations. We have to show that the same holds for any $T = u_1, \ldots, u_i, u_{i+1}, c$ and $T_i = u_1, \ldots, u_i, c$. WLOG assume $End(T) = End(T_i)$. We know that $Q(T_i) \equiv_{\mathbb{N}[X]^\nu} T_i = R[T_i, End(T_i)] = R[T_i, \nu(u_i)+1]$. 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 $R[T, End(T)] = R[T, \nu(u_{i+1})+1] = u_{i+1}(R[T, \nu(u_{i+1})])$. From the equivalences stated above we can deduce $u_{i+1}(R[T, \nu(u_{i+1})]) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}(Q^R(u_i)))$. We know that $Q(u_{i+1}) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}$ and, thus, it follows that $R[T, End(T)] \equiv_{\mathbb{N}[X]^\nu} Q^R(u_{i+1})$. Since $Q^R(u_{i+1})$ is the reenactment query for T, this concludes the proof. ☐

5.2.2 RC-SI Reenactment

Under RC-SI each update u in a transaction T sees tuple versions created by previous updates of the same transaction and tuple versions committed before $\nu(u)$.

We would like reenactment queries for RC-SI to be defined recursively without requiring to recalculate the right mix tuple versions from transaction T and from concurrent transactions after each update. To be able to fulfill this requirement we need to 1) enhance our annotation model to record the version when a summand ceased to be valid in version annotations and 2) introduce an extended version of the selection operator that allows conditions which access the start and end time of version annotations using a pseudo attributes V_b and V_e. We use this operator to filter summands from annotations based on the version annotations they are wrapped in.

We first extend version annotations and based on this extension define $R[\nu_b, \nu_e]$, a version of relation R that is annotated with all summands that were valid at some time between ν_b and ν_e.

DEFINITION 13 (EXTENDED VERSION ANNOTATIONS). *Let H be a history. Extended version annotations are version annotations enhanced with an end time. A summand that ends at version ν_e and was produced by an update of transaction T at version ν_b is denoted by ${}^{T}_{I/U/D}\tau_{\nu_b}^{\nu_e}$. If such a summand was never overwritten we set $\nu_e = \infty$. The version slice $R[\nu_b, \nu_e]$ of relation R between versions ν_b and ν_e according to a history H contains is defined in Figure 9.*

In the above definition we consider a summand k to be valid in a range of versions ν_b and ν_e if there exists a version in the interval $[\nu_b, \nu_e]$ for which k is valid. This definition is analog to $R[\nu]$ with the exception that it is defined for an interval and that each version annotation is extended with an end version (when a summand was overwritten). Note that the validity check we use here is the one we have defined for SI.

We now introduce the version selection operator, an extended version of the selection operator that is used to filter summands based on the version annotations they are wrapped in.

DEFINITION 14 (VERSION SELECTION OPERATOR). *Let θ be a condition over attributes from a relation R and pseudo attributes V_b, V_e, and X representing parameters of version annotations. Given a summand $k = {}^{T}_{U/D/I}\tau_{\nu_b}^{\nu_e}(k')$ such a condition is evaluated by replacing V_b with ν_b, V_e with ν_e, and X with T. The version filter operator using such a condition θ is defined as:*

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

For example, we could use $\gamma_{V_b<11}(R)$ to filter out parts of annotations from a relation R that were created by an update executed before 11. Note that in contrast to regular selection, a version selection operator's condition is evaluated over the individual summands in an annotation which is necessary to filter individual summands. To correctly reenact an RC-SI transaction T, we need to provide each reenactment query for an update $u \in T$ with the right mix of summands created by previous updates of transaction T and transactions that have committed before or at $\nu(u)$. We first show that $R[Start(T), End(T)-1]$ contains all summands (not created by T) that are needed to evaluate any update in T and how to filter out the input for update u_i from this version of relation R (except for tuple versions created by T itself). Then we modify reenactment queries for updates to directly operate over the result of the previous update and let the modified reenactment query for the first update modifying relation R read from $R[Start(T), End(T)-1]$.

Consider $R_T[\nu(u_i)-1]$ for an update u_i from transaction T. The summands in an annotation on a tuple t in $R_T[\nu(u_i)-1]$ can be split into two sets. Summands that

Insertions

$$R[T,\nu] \rightsquigarrow \begin{cases} \gamma_{V_b \leq \nu \wedge V_e > \nu}(Q^R(\textsc{Last}(R,T,\nu))) & \text{if } \exists u \in T : u(R) \wedge \nu(u) < \nu \\ \gamma_{V_b \leq \nu \wedge V_e > \nu}(R[Start(T), End(T)-1]) & \text{else} \end{cases} \tag{I1}$$

$$R[T,\nu] \rightsquigarrow \begin{cases} Q^R(\textsc{Last}(R,T,\nu)) & \text{if } \exists u \in T : u(R) \wedge \nu(u) < \nu \\ R[Start(T), End(T)-1] & \text{else} \end{cases} \tag{I2}$$

Deletions and Updates

$$\sigma_\theta(R[T,\nu]) \rightsquigarrow \gamma_{V_b \leq \nu \wedge V_e > \nu}(R[T,\nu]) \tag{UD1}$$

$$\sigma_{\neg\theta}(R[T,\nu]) \rightsquigarrow \gamma_{\neg\theta \vee V_b > \nu \vee V_e \leq \nu}(R[T,\nu]) \tag{UD2}$$

$$R[T,\nu] \rightsquigarrow \begin{cases} Q^R(\textsc{Last}(R,T,\nu)) & \text{if } \exists u \in T : u(R) \wedge \nu(u) < \nu \\ R[Start(T), End(T)-1] & \text{else} \end{cases} \tag{UD3}$$

Figure 10: RC-SI Transaction Reenactment Substitution Rules

have been updated previous updates of transaction T and summands created by other transactions. The second set is equal to $R[\nu(u_i)-1]$ as defined for SI historic databases. Consider a summand k in the annotation of a tuple t in $R[\nu(u_i)-1]$ that would get updated by u_i. This summand will obviously also exist in $R[Start(T), End(T)-1]$, because this version of relation R contains all summands that were valid at any time during the interval $[Start(T), End(T)-1]$.

We modify update reenactment queries so that if the first reenactment query is run over $R[Start(T), End(T)-1]$, 1) each such query Qu_i updates the correct set of summands and 2) the query propagates every summand k that is affected by an update u_j with $j > i$ or is not affected by any update in T. This is achieved as follows. For updates we replace the selection which filters out tuples that would get updated (σ_θ) with $\gamma_{\theta \wedge V_b \leq \nu \wedge V_e > \nu}$, i.e., we also check whether each summand should be visible to u_i. Furthermore, in the other branch of the union we replace $\sigma_{\neg\theta}$ with $\gamma_{\neg\theta \vee V_b > \nu \vee V_e \leq \nu}$ (this is the negation of $\theta \wedge V_b \leq \nu \wedge V_e > \nu$) to ensure that summands not visible to u_i are pushed to the reenactment queries for the following updates. For deletions we also replace σ_θ and $\sigma_{\neg\theta}$ in the same fashion. For an insertion u_i we replace accesses to each relation R_j in the query Q of the insertion with $\gamma_{V_b \leq \nu(u_i) \wedge V_e > \nu(u_i)}(R_j)$.

Definition 15 (Transaction RC-SI Reenactment). *Let $T = u_1, \ldots u_n, c$ be a transaction in a RC-SI history H. The reenactment query $Q(T)$ for T is defined as follows:*

$$Q(T) = \{Q^R(T) \mid R \in R(T)\}$$
$$Q^R(T) = \gamma_{\nu_b \leq \nu(\textsc{Last}(T,R,End(T))) \wedge \nu_e > End(T)-1}(Q^R(\textsc{Last}(T,R,End(T))))$$

The reenactment query $Q^R(u)$ is defined as the result of exhaustively applying the substitution rules shown in Figure 10. If u is an insertion, then we apply rule (I1) to the left input of the union in $Q(u)$ and (I2) to the right input of the union. For deletions and updates we exhaustively apply the following substitution rules (UD1) to (UD3) shown Figure 10 in that order.

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

Theorem 6 (Annotation Equivalence). *Let T be a RC-SI transaction and $Q(T)$ its reenactment query. Then T and $Q(T)$ are annotation equivalent:*

$$T \equiv_{\mathbb{N}[X]^\nu} Q(T)$$

Proof. Assume that transaction $T = u_1, \ldots, u_n, c$ is updating a single relation R. The proof can easily be extended for transactions updating multiple relations. We have to prove that for any potential summand k in an annotation on a tuple t, k appears in $R[T, End(T)](t)$ iff k appears in $\gamma_{\nu_e > End(xid)-1}(Q^R(u_n)(t))$. We prove this fact by induction over the number of updates in T.

Induction Start: Let $T = u_1, c$. We distinguish two cases. The first case applies if u_1 is an update or delete. In this case, the selection condition $V_b \leq \nu(u_1) \wedge V_e > \nu(u_1)$ in the modified reenactment query for u_1 returns $R[\nu(u_1)]$. The other input of the union is filtered using $V_b > \nu(u_1) \wedge V_e \leq \nu(u_1)$. This input may include additional summands k created by transactions which have committed between $\nu(u_1)+1$ and $End(T)$. However, such summands will be filtered out by final version selection on condition $\gamma_{\nu_b \leq \nu(\textsc{Last}(T,R,End(T))) \wedge \nu_e > End(T)-1}$.

If u_1 is an insert, then the version selection in the modified reenactment query for u_1 ensures that the insertion query Q is evaluated over the correct input. The right input of the union simply returns $R[Start(T), End(T)-1]$ and again the final version selection filters out summands as required.

Induction Step: Assume that any transaction of length up to i is equivalent to its reenactment query. Let $T = u_1, \ldots, u_i, u_{i+1}, c$ be a RC-SI transaction of length $i+1$. We know that the induction hypothesis holds for $T_i = u_1, \ldots, u_i, c$. We have to prove that for any summand k in an annotation of a tuple t in the input of u_{i+1}, 1) iff k was updated by u_{i+1} then it will be updated by the corresponding reenactment query and 2) iff k was not updated then it will be unmodified in the result of the reenactment query. That k is present in the input of the modified reenactment query for u_{i+1} follows from the induction hypothesis. If k was updated then k would have to fulfill the version selection condition $V_b < \nu(u_i) \wedge V_e > \nu(u_i)$ and, thus, will be updated by the modified reenactment for u_{i+1}. If k was not updated then it either 1) fulfills the condition $V_b < \nu(u_i) \wedge V_e > \nu(u_i)$, but does not fulfill the condition of the update (query in case of an insert or selection condition in case of an update or delete) or 2) does not fulfill

$V_b < \nu(u_i) \wedge V_e > \nu(u_i)$. In the first case, k will be in the result which follows from the correctness of update reenactment. In the second case, the modified reenactment query will ensure that k is in the result. In case of an update or delete, k will fulfill $V_b < \nu(u_i) \wedge V_e > \nu(u_i)$ in the version selection on $\neg\theta \vee V_b > \nu(u_{i+1}) \vee V_e \leq \nu(u_{i+1})$. In case of an insert the result of the modified reenactment for u_i is included unmodified (union). Since k was in the result of this query, it follows that k is in the result of $Q^R(u_{i+1})$ which concludes the proof. □

6. RELATIONAL REENACTMENT USING TIME TRAVEL AND AUDIT LOGS

In this section, we introduce the necessary techniques for retrieving the provenance of transactions for standard relational databases through an relational encoding of reenactment queries based on audit logging and time travel. We first define an relational encoding of $\mathcal{K}^\nu$-relations. Afterwards we discuss what type of audit logging and time travel is required by our approach - audit logs are used to determine which SQL statement was executed by which transaction and at which version. Finally, we demonstrate how reenactment queries can be translated into standard relational algebra statements that produce such the relational encoding of the partial provenance of a transaction by extending query rewrite techniques for computing the provenance of queries.

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

We extend the relational encoding of provenance polynomials introduced for the Perm [15, 18] project with additional columns that encode version annotations to be able to encode $\mathbb{N}[X]^\nu$ relations. The basic idea behind this encoding is to normalize provenance polynomials according to the query that produced them and then represent variables by actual tuple values from the inputs of the query. In particular, the provenance polynomial is normalized to a sum of products where in each product the variables are ordered according to the relation they belong too. Given the relational algebra tree for a query Q, variables in products are ordered according to the leafs of the algebra tree. We add additional attributes to be able to encode such a product and represent each summand in a provenance polynomial as a separate tuple.

Definition 16 (Normalized $\mathbb{N}[X]$). *Let $Q(R_1, \ldots, R_n)$ be a query and $Q(D)$ the $\mathbb{N}[X]$-result of evaluating Q over an instance D for relations R_1 to R_n. A provenance polynomial $Q(D)(t)$ is normalized iff it is of the form:*

$$\sum_{i=1}^{n(Q(D)(t))} \prod_{j=1}^{n} k_{ij}$$

where $k_{ij} = 1$ or $k_{ij} = x_{ij}$ for some variable x_{ij} so that $R_j(t') = x_{ij}$ for some t'. We call $Q(D)$ normalized if for every t the annotation $Q(D)(t)$ is normalized.

Based on this normal form, it is possible to define a relational encoding of the result of a query Q with $\mathbb{N}[X]$ semantics and generate a relational algebra expression Q^+ from Q that produces this relational encoding [18].

Definition 17 (Query Encoding Schema). *Let $Q(R_1, \ldots, R_n)$ be a query and $Q(D)$ the normalized $\mathbb{N}[X]$-result of evaluating Q over an instance D for relations R_1 to R_n. Let $\textsc{Null}(R)$ denote a list of null values with the same arity as relation R. We use $t \triangleright t'$ to denote the concatenation of two lists t and t' and $t(x)$ to denote the tuple annotated with variable x in the input of Q. The relational encoding $\textsc{Rel}(Q(D))$ is defined as:*

$$\textsc{Rel}(Q(D)) = \bigcup_{t:Q(D)(t)\neq 0} \bigcup_{i=1}^{n(Q(D)(t))} \{t \triangleright \hat{k_{i1}} \ldots \triangleright \hat{k_{in}}\}$$

$$\hat{k_{ij}} = \begin{cases} t(k_{ij}) & \text{if } k_{ij} = x_{ij} \\ \textsc{Null}(R_j) & \text{else} \end{cases}$$

The schema $\textsc{Rel}(Q(D))$ is defined using a renaming function P that maps a relation and attribute name to a provenance attribute name. In the following we use $P(R_i)$ to denote the list of attribute names containing $P(R_i, A)$ for each $A \in R_i$. Furthermore, let $ID_\mathcal{P}$ denote a function that takes a list of attribute names and adds unique identifiers to attribute names that occur more than once in the list.

Definition 18 (Query Encoding Schema). *The schema $\textsc{Sch}(\textsc{Rel}(Q(D)))$ for $\textsc{Rel}(Q(R_1, \ldots, R_n))$ is defined as:*

$$\textsc{Sch}(\textsc{Rel}(Q(D))) = ID_\mathcal{P}(\textsc{Sch}(Q) \triangleright P(R_1) \triangleright \ldots \triangleright P(R_n))$$

Example 10. *Consider a query $\Pi_A(R \bowtie S)$ over relations $R(A,B)$ and $S(B,C)$ with instances $R = \{(1,2) \rightarrow r_1, (1,3) \rightarrow r_2\}$ and $S = \{(2,4) \rightarrow s_1, (3,5) \rightarrow s_2\}$. This query has a single result tuple $(1) \rightarrow r_1 \times s_1 + r_2 \times s_2$. The annotation of this result tuple is already normalized, because 1) it is a sum of products and 2) in each product a variable representing a tuple from relation R is in the first position and a variable representing a tuple from relation S is in the second position. The schema for the relational encoding of provenance polynomials will be A, $P(R,A)$, $P(R,B)$, $P(S,B)$, $P(S,C)$. The full relational encoding of the result of Q is shown below. We show the corresponding part of the provenance polynomial at the left of each tuple.*

	A	P(R,A)	P(R,B)	P(S,B)	P(S,C)
$r_1 \times s_1$	1	1	2	2	4
$r_2 \times s_2$	1	1	3	3	5

Our relational encoding of the partial provenance $R[T]$ of a transaction T is based on the same principles: 1) normalize $\mathcal{K}^\nu$-expressions according to the operations that were applied to the data and 2) use additional attributes to represent a normalize $\mathcal{K}^\nu$-polynomial. We use a boolean attribute to represent a potential version annotations for a particular update operation in a transaction T. This attribute is set to true for a tuple in the encoding if this tuple represents a summand that has the version annotation. Note that a boolean attribute is sufficient, because the nesting order of version annotations from a transaction T is fixed (based on the order of update operations in the transaction). An exception are inserts where the query Q of the insert accesses a relation updated by the transaction. In this case, the elements in a normalized provenance polynomial for a query result t may carry version annotations from previous updates of the transaction. Thus, we need to add additional

Schedule

	Schedule			Schedule Provenance			u_1	u_2						u_3
	c	b	d	P(S,c)	P(S,b)	P(S,d)	$\mathcal{A}_1$	P(B,c)	P(B,n)	P(B,p)	P(B,f)	P(B,t)	$\mathcal{A}_2$	$\mathcal{A}_3$
${}^{T_2}_{D}\tau_{12}(s'_1)$	Whitedog	13	8:15	Whitedog	13	8:15	T						F	F
${}^{T_2}_{U}\tau_{14}(s'_3)$	Picobus	2	10:30	Picobus	2	10:30	F						F	T
${}^{T_2}_{U}\tau_{14}(s'_4)$	Picobus	2	10:30	Picobus	2	12:15	F						F	T
${}^{T_2}_{I}\tau_{13}(b'_4)$	Greyhound	5	20:15				F	Greyhound	5	96	Chicago	New York	T	F

Figure 11: Relational Encoding of Partial Provenance for Example Transaction T_2

$$\textsc{Sch}(R[T]) = ID_{\mathcal{P}}(\textsc{Sch}(R^n)) \qquad \textsc{Sch}(R^0) = \textsc{Sch}(R) \triangleright P(R) \qquad \textsc{Sch}(R^{i+1}) = \textsc{Sch}(R^i) \triangleright \mathcal{P}(u_{i+1})$$

$$\mathcal{P}(u_i) = \begin{cases} \mathcal{P}(R_1, i) \triangleright \ldots \mathcal{P}(R_n, i) \triangleright \mathcal{A}_i & \text{if } u_i = \mathcal{I}[Q(R_1, \ldots, R_n), T, \nu](R) \\ \mathcal{A}_i & \text{else} \end{cases} \qquad \mathcal{P}(R_j, i) = \begin{cases} P(R_j) & \text{if } R_j \neq R \\ \textsc{Sch}(R^i) & \text{else} \end{cases}$$

Figure 12: Schema of the Relational Encoding of $R[T]$

attributes to represent such version annotations. This leads to a hierarchical definition of a schema to accomodate cases such as an insert that queries R over the result of another insert querying R.

Consider $R[T](t)[i] = k_i$, a summand in an annotation on a tuple t in the instance of relation R produced by transaction $T = (u_1, \ldots, u_n)$. This annotation will be of the form $\tau_{\ i_1}(\ldots(\tau_{\ n_i}(k'_i))\ldots)$ where each i_j is in $\{1, \ldots, n\}$ (a version annotation from one of the updates of T) and k'_i is either a simple variable x (if k'_i is an annotation of a existing tuple from R updated by the transaction) or a query result annotation $\sum_{j=1}^{n(k'_i)} \prod_{l=1}^{m_j} k_{ijl}$ (if k'_i was produced by the query of an insert within transaction T). In the second case, each k_{ijl} will again be of one of the above two forms. This insights about the structure of annotations in $R[T]$ will help us to define a normal form which in turn is used to come up with an relational encoding of $R[T]$.

Addition operations are all pulled up to the top level of an annotation in this normal form. To be able to transform annotations into that normal form, we need the following equivalence

$$\tau(k + k') = \tau(k) + \tau(k') \qquad (1)$$

for any $\mathcal{K}^\nu$-elements k and k' and version annotation τ. This equivalence follows from the interpretation of version annotations as identity functions (update and insert) and functions that map all inputs to the 0 element of the semiring (delete). Consider again an annotation $R[T](t)[i]$. In the first case, the inner element k'_i is already normalized. In the second case, we can pull out the inner sum produced by the insert query as follows:

$$\tau_{\ i_1}(\ldots(\tau_{\ n_i}(\sum_{j=1}^{n(k'_i)} \prod_{l=1}^{m_j} k_{ijl}))\ldots)$$
$$= \sum_{j=1}^{n(k'_i)} \tau_{\ i_1}(\ldots(\tau_{\ n_i}(\prod_{l=1}^{m_j} k_{ijl}))\ldots)$$

Each k_{ijl} will again be of either form (1) or (2). In the first case, the resulting expression is already normalized. In the second case, we can use distributivity of addition and multiplication to pull out additions.

Definition 19 (Normalized $\mathbb{N}[X]^\nu$). *Let T be a transaction updating relation R. An $\mathbb{N}[X]^\nu$ annotation $R[T](t)$ is normalized if it is of the form: $\sum_{i=0}^{m} k_i$ where each k_i is a mix of multiplications and version annotations resulting from applying equivalence* (1) *and distributivity of addition over multiplication. Furthermore, summands that evaluate to* 0 *and are not wrapped in a deletion annotation are removed in the normalized representation.*

By repeatedly applying equivalence (1) and distributivity of sum over multiplication any annotation $R[T](t)$ can be translated into the above normal form. Our relational encoding of $R[T]$ is based on this normal form. For simplicity, we state the definition for transactions that only update a single relation R. The extension to multiple relations is straightforward. We first define the schema of this encoding and then the actual instance.

Definition 20 (Transaction Encoding Schema). *Given the partial provenance $R[T]$ for a transaction $T = u_1, \ldots, u_n$, the schema of the relational encoding* $\textsc{Rel}(R[T])$ *of $R[T]$ is defined as follows. First let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ denote a list of boolean attributes used to represent version annotations ($\mathcal{A}_i$ represents the annotation produced by update u_i). Let P be defined as in Definition 18. The schema* $\textsc{Sch}(\textsc{Rel}(R[T]))$ *is defined in Figure 12.*

Note that in $\textsc{Sch}(R^0)$ we add provenance attributes for relation R to represent the variable x if a summand in an annotation $R[T](t)$ is of the form (1). The hierarchical definition is necessary, because for an insertion where the query Q of the insert accesses the updated relation R, the input tuples of the query from relation R may already have been annotated by previous updates and inserts of the same transaction. To be able to represent such annotations, we need to add additional version attributes and attributes of relations accessed by previous inserts.

In our implementation, we use a renaming function P that creates attribute names with a prefix `prov_` to distinguish provenance from non-provenance attributes. P includes the input relation and attribute name in its output. If necessary a sequential number if a relation is referenced more than once (corresponding to application of function $ID_{\mathcal{P}}$). For example, $P(R, A) =$`prov_R_A` if this is the first reference to $R.A$ in the provenance. As another example, consider $Q = R \bowtie R$. The provenance attribute names for the relational

$$\textsc{Rel}(R[T]) = \bigcup_{t \in R} \bigcup_{i=0}^{n(R[T](t))} t \triangleright \textsc{Rel}^n(R[T](t)[i]) \qquad \textsc{Rel}^0(k) = \begin{cases} t(x) & \text{if } k = \tau_{i_1}(\ldots(\tau_{n_i}(x))\ldots) \\ \textsc{Null}(P(R)) & \text{else} \end{cases}$$

$$\textsc{Rel}^{i+1}(k) = \textsc{Rel}^i(k) \triangleright \textsc{EncU}^{i+1}(k)$$

$$\textsc{EncU}^i(k) = \begin{cases} \mathit{True} & \text{if } k = \tau_{j_1}(\ldots(\tau_{n_j}(x))\ldots) \wedge i \in \{j_1,\ldots,n_j\} \\ \mathit{False} & \text{if } k = \tau_{j_1}(\ldots(\tau_{n_j}(x))\ldots) \wedge i \notin \{j_1,\ldots,n_j\} \\ \textsc{EncR}(R_1,k_1,i),\ldots,\textsc{EncR}(R_m,k_1,i) & \text{if } k = \tau_{j_1}(\ldots(\tau_{n_j}(k_1 \times \ldots \times k_m))\ldots) \wedge u_i \text{ is an insert} \\ \textsc{Null}(u_i) & \text{else} \end{cases}$$

$$\textsc{EncR}(R_j,k,i) = \begin{cases} t(x) & \text{if } R_j \neq R \wedge k = x \\ \textsc{Rel}^i(k) & \text{else} \end{cases}$$

Figure 13: Relational Encoding of an $R[T]$ Instance

encoding of the provenance of Q are `prov_R_A`, `prov_R_B`, `prov_R_A_1`, and `prov_R_B_1`.

The instance of the relational encoding $\textsc{Rel}(R[T])$ is created by representing each summand in a normalized annotation $R[T](t)$ as a separate tuple.

Definition 21 (Transaction Encoding Instance). *Let t be a tuple in R and $R[T](t) = \sum_1^m k_m$ be the normalized annotation of t. The annotation of t is encoded as m tuples t_1 to t_m each representing one summand in the provenance. Let $\textsc{Null}(u_i)$ denote a list of null values for each attribute from $\mathcal{P}(u_i)$. Furthermore, let $t(x)$ denote the tuple corresponding to a variable x. The relational encoding $\textsc{Rel}(R[T])$ is defined as shown in Figure 13.*

The relational encoding is constructed by generating one tuple for each summand in an normalized $\mathbb{N}[X]^\nu$-annotation of a tuple in $R[T]$. This tuple is constructed by concatenating the relational representations for parts of the annotation corresponding to the individual updates statements of the transaction. The original tuple (to be precise the single variable annotating this tuple) in case of a sequence of updates and deletes is encoded in $\textsc{Rel}^0(k)$. The part $\textsc{Rel}^i(k)$ represents the version annotation of u_i if u_i is an update or delete or the relational encoding of a query result in case u_i is an insert. Note that this encoding may encode summands that evaluate to 0 (e.g., in case of a deleted tuple).

Example 11. *Figure 11 shows the relational encoding of Schedule$[T_2]$ for transaction T_2 from the running example. We use abbreviations S and B for relations `Schedule` and `Bus`, respectively. Attribute names are abbreviated as in previous examples. This transaction contains a single insert which uses a query $\Pi_{c,s,'20:15'}(\sigma_{c='Greyhound'}(Bus))$. Thus, the provenance attribute schema contains attributes for relation `bus`. Consider the second and third tuple in this relation. These tuples represent the two summands in the annotation of tuple (Picobus,2,10:30). Both summands contain a single version annotation ${}^{T_2}_{U}\tau_{14}$. Thus, only the attribute $\mathcal{A}_3$ corresponding to this version annotation is true and all other attributes encoding version annotations are set to false. The summand encoded by the second tuple references variable s_3' which is the annotation of tuple (Picobus,2,10:30) in the partial provenance (recall that for the partial provenance subexpressions wrapped in version annotations of earlier transactions are replaced with fresh variables). The last tuple is the result of inserting the result of a query with a single tuple in (b_4') in its provenance. Thus, the version annotation*

xid	pos	version	SQL
T_1	0	10	`INSERT INTO ...`
T_1	1	11	`COMMIT`
T_2	0	12	`DELETE FROM ...`
T_2	1	13	`INSERT INTO ...`
T_2	2	14	`UPDATE Schedule ...`
T_2	3	15	`COMMIT`

Figure 14: Audit Log Example

attribute corresponding to the insertion is set to true and the values of the tuple annotated with b_4 are stored in provenance attributes representing the inputs (bus relation) of this query.

6.2 Audit Log

We assume that the underlying database system on which we want to execute provenance computations keeps an audit log that can be queried. An audit log has to provide at least the following information for each update statement executed on the database: 1) The SQL code for the update statement, 2) the version (time) when the update was executed, 3) an identifier (*xid*) for the transaction the update was part of, and 4) the position of the update within its transaction. We require that the commit operation of a transaction T is stored as a separate entry in the audit log to be able determine $End(T)$.

Given a transaction identifier T, we use the audit log to determine the operations of a transaction and the database version they have accessed during provenance computation.

Example 12. *Figure 14 shows an audit log for our running example. For instance, the update of transaction T_2 was the third statement in the transaction (*pos *is* 2*) and was executed at time* 14*.*

6.3 Time Travel

We assume a standard snapshot isolation based implementation of time travel as supported in similar fashion by multiple commercial database systems. 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 get updated by setting their end time to the current time. These tuple version modifications are only visible to the updating transaction. When a transaction commits, then new tuple versions are created for all

tuples modified by the transaction. The new tuple versions start time is set to the transaction commit time. The end time of the previous tuple version is also set to the commit time of the transaction.

Definition 22 (Snapshots and Time Slices). *A snapshot R_ν of relation R contains all committed tuple versions valid at ν. A time slice $R_{[\nu_b,\nu_e]}$ of a relation R contains all tuple versions that were valid at any time in the interval $[\nu_b, \nu_e]$. Snapshots and time slices have two additional attributes TT_b and TT_e storing the validity time interval of a tuple version.*

Example 13. *Consider the transaction T (committed at time 7) and the instance of relation R shown below. For convenience, we show the validity interval of each tuple. The snapshot R_7 of relation R at time 7 contains two tuple versions. The first version was created by the two updates of the transaction and, thus, its begin time is set to 7. The second tuple version was already present when the transaction started and was not affected by the transaction.*

R

	A	B
$[1,\infty]$	1	2
$[1,\infty]$	2	3

R₇

A	B	TT_b	TT_e
0	6	7	∞
2	3	1	∞

$$T = \mathcal{U}[A = 1, B \rightarrow 6, T, 3](R), \mathcal{U}[B = 6, A \rightarrow 0, T, 5](R)$$

6.4 Relational Implementation of Reenactment

We now discuss how reenactment queries over $\mathcal{K}^\nu$-relations can be implemented as relational algebra queries with bag semantics which produce the relational encoding of $R[T]$ for a transaction T introduced in the last section. This rewriting of queries with $\mathcal{K}^\nu$-semantics into standard bag semantics queries (expressible in SQL) significantly extends previous results for rewriting queries with $\mathcal{K}$-relational semantics into bag semantics [18, 15, 16].

The rewriting is done by recursively applying rewrite rules for single relational operators in a top-down fashion. These rewrite rules are context-free in the sense that the only information needed to rewrite an algebra operator is knowledge about which attributes from its rewritten inputs encode annotations. We apply a selection on the boolean version annotation attributes to retrieve only tuple versions that were affected by the transaction to compute the partial provenance from the translated reenactment query.

Definition 23 (SI Rewrite Rules). *Let $T = u_1, \ldots, u_n, c$ be a transaction and let τ_i denote the version annotation created by the ith update in T. The relational translation of the reenactment query $\text{Rel}(Q(T))$ restricted to partial provenance is derived from $Q(T)$ as shown below. Here* Rew *is the rewrite operator defined in Figure 15 where* Null *$(\mathcal{P}(q))$ denotes a singleton relation with null values for all annotation attributes of q except for attributes for version annotations which are set to $false$.*

$$\text{Rel}(Q(T)) = \sigma_{\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n}(\text{Rew}(Q(T)))$$

The query produced by the rewrite rules of Figure 15 creates unnecessary duplicates of annotation attributes. In praxis, we therefore share provenance attributes for updated relations in union operators where possible. Furthermore, we have ignored the renaming of provenance attributes that occur more than once ($ID_{\mathcal{P}}$).

Example 14 (Relational Reenactment). *Consider the reenactment query for transaction T_2 from our running example. The rewritten version of this query is shown in Example 9. The accesses to relations Schedule and Bus at version 12 have been rewritten into snapshots and by duplicating their attributes as initial provenance annotation. Recall that we represent variables in annotations on relations in the database as the tuples they are annotating. The part of the reenactment query corresponding to update u_1 ($Q^S(u_1)$) has been rewritten by replacing the accesses to relation Schedule with their rewritten counterpart. The version annotation operator has been replaced with a projection adding a constant true as the value for version annotation attribute $\mathcal{A}_1$. Here we have shared common annotation attributes among the two inputs of the union. For the only attribute not in the right input ($\mathcal{A}_1$) we generate* Null*(q_2) which is $\{(false)\}$. Update u_2 is an insert with a query that accesses relation Bus. The provenance of this query has provenance in relation Bus whereas the provenance of u_1 has provenance in relation Schedule. The rewritten version of $Q^S(u_2)$ applies crossproducts with constant relations to make the two inputs of the union in the reenactment query union compatible. The annotation operator has again been rewritten into a projection. Finally, the part of the reenactment query corresponding to the last update of transaction T_2 is rewritten by translating the version annotation operator into a projection and by adding a constant $false$ to each tuple from the right input to make the inputs union compatible. Note that, for sake of brevity, we have omitted the relation in annotation attributes, e.g., we write $\mathcal{P}(n)$ instead of $\mathcal{P}(B, n)$.*

Reenactment queries for RC-SI transactions apply the version selection operator and use the construct $R[\nu_b, \nu_e]$ to access a range of database versions. Recall that snapshots and time slices have two additional attributes TT_b and TT_e which encode the time during which a tuple version was valid. The version selection operator can be implemented as a simple selection over these attributes by replace pseudo attributes V_b and V_e with TT_b and TT_e, respectively (see rewrite rule for γ_θ in Figure 15. We add TT_b and TT_e to the annotation attributes of a relation to make them available to all version selections in the rewritten reenactment query. These additional attributes have to be removed in the end.

Definition 24 (RC-SI Rewrite Rules). *Let T be a RC-SI transaction. The relational translation of the reenactment query $Q(T)$ restricted to partial provenance is derived from $Q(T)$ using the rules for SI rewrite with the exception that we add the TT_b and TT_e attributes to $\mathcal{P}(R)$ and remove these attributes in the end.*

$$\text{Rel}(Q(T)) = \sigma_{\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n}(\Pi_{\text{Sch}(R), \mathcal{P}(Q(T))}(\text{Rew}(Q(T))))$$

Having defined the relational translation of reenactment queries it remains to be shown that this translation is in fact correct, i.e., for any transaction T, the query that is the result of this translation produces the relational encoding of the partial provenance of transaction T.

Theorem 7 (Correctness). *Let T be a transaction updating relation R. The relational implementation of the reenactment query for T produces the relational encoding of the partial provenance for T:*

$$\text{Rel}(Q(T)) = \text{Rel}(R[T])$$

Structural Rewrite

$$
\begin{aligned}
\textsc{Rew}(R[\nu]) &= \Pi_{\textsc{Sch}(R),\textsc{Sch}(R)\rightarrow\mathcal{P}(R)}(R_\nu)\\
\textsc{Rew}(R[\nu_b,\nu_e]) &= \Pi_{\textsc{Sch}(R),\textsc{Sch}(R)\rightarrow\mathcal{P}(R)}(R_{[\nu_b,\nu_e,)]})\\
\textsc{Rew}(\sigma_\theta(q)) &= \sigma_\theta(\textsc{Rew}(q))\\
\textsc{Rew}(\gamma_\theta(q)) &= \sigma_{\theta[V_b\leftarrow TT_b, V_e\leftarrow TT_e]}(\textsc{Rew}(q))\\
\textsc{Rew}(\Pi_A(q)) &= \Pi_{A,\mathcal{P}(q)}(\textsc{Rew}(q))\\
\textsc{Rew}(q_1\cup q_2) &= (\textsc{Rew}(q_1)\times \textsc{Null}(\mathcal{P}(q_2)))\cup(\Pi_{\textsc{Sch}(q_1),\mathcal{P}(q)}(\textsc{Rew}(q_2)\times\textsc{Null}(\mathcal{P}(q_1))))\\
\textsc{Rew}(q_1\bowtie_\theta q_2) &= \Pi_{\textsc{Sch}(q_1),\textsc{Sch}(q_2),\mathcal{P}(q_1),\mathcal{P}(q_2)}(\textsc{Rew}(q_1)\bowtie_\theta\textsc{Rew}(q_2))\\
\textsc{Rew}(\mathcal{A}_i(q)) &= \Pi_{\textsc{Sch}(\textsc{Rew}(q)),true\rightarrow\mathcal{A}_i}(\textsc{Rew}(q))
\end{aligned}
$$

Annotation Attributes

$$
\begin{aligned}
\mathcal{P}(R[\nu]) &= \mathcal{P}(R)\\
\mathcal{P}(R[\nu_b,\nu_e]) &= \mathcal{P}(R), TT_b, TT_e\\
\mathcal{P}(\sigma_C(q)) &= \mathcal{P}(q)\\
\mathcal{P}(\Pi_A(q)) &= \mathcal{P}(q)\\
\mathcal{P}(q_1\cup q_2) &= \mathcal{P}(q_1)\triangleright\mathcal{P}(q_2)\\
\mathcal{P}(q_1\bowtie q_2) &= \mathcal{P}(q_1)\triangleright\mathcal{P}(q_2)\\
\mathcal{P}(\mathcal{A}_i(q)) &= \mathcal{P}(q)\triangleright\mathcal{A}_i
\end{aligned}
$$

Figure 15: Rewrite Rules for Translating $\mathcal{K}^\nu$-semantics Reenactment Queries into Standard Relational Semantics (Bag)

$$
\begin{aligned}
\textsc{Rew}(Q(T)) &= \sigma_{\mathcal{A}_1\vee\mathcal{A}_2\vee\mathcal{A}_3}(\textsc{Rew}(Q^S(u_3)))\\
\textsc{Rew}(Q^S(u_3)) &= \Pi_{c,b,10:30\rightarrow d,\mathcal{P}(c),\mathcal{P}(b),\mathcal{P}(d),\mathcal{A}_1,\mathcal{P}(c),\mathcal{P}(n),\mathcal{P}(p),\mathcal{P}(f),\mathcal{P}(t),\mathcal{A}_2,true\rightarrow\mathcal{A}_3}(\sigma_{c='Picobus'\wedge b=2}(\textsc{Rew}(Q^S(u_2))))\\
&\quad\cup\sigma_{\neg(c='Picobus'\wedge b=2)}(\textsc{Rew}(Q^S(u_2)))\times\{(false)\}\\
\textsc{Rew}(Q^S(u_2)) &= (\textsc{Rew}(Q^S(u_1))\times\{(null,null,null,null,null,false)\})\\
&\quad\cup(\Pi_{c,b,'20:15',\mathcal{P}(c),\mathcal{P}(b),\mathcal{P}(d),\mathcal{A}_1,\mathcal{P}(c),\mathcal{P}(n),\mathcal{P}(p),\mathcal{P}(f),\mathcal{P}(t),true\rightarrow\mathcal{A}_2}(\sigma_{c='Greyhound'}(\textsc{Rew}(B[12]))\\
&\qquad\times\{(null,null,null,false)\}))\\
\textsc{Rew}(Q^S(u_1)) &= \Pi_{c,b,d,\mathcal{P}(c),\mathcal{P}(b),\mathcal{P}(d),true\rightarrow\mathcal{A}_1}(\sigma_{c='Whitedog'\wedge b=13}(\textsc{Rew}(S[12])))\\
&\quad\cup\Pi_{c,b,d,\mathcal{P}(c),\mathcal{P}(b),\mathcal{P}(d),\mathcal{A}_1}(\sigma_{\neg(c='Whitedog'\wedge b=13)}(\textsc{Rew}(S[12]))\times\{(false)\})\\
\textsc{Rew}(S[12]) &= \Pi_{c,b,d,c\rightarrow\mathcal{P}(c),b\rightarrow\mathcal{P}(b),d\rightarrow\mathcal{P}(d)}(S_{12})\\
\textsc{Rew}(B[12]) &= \Pi_{c,n,p,f,t,c\rightarrow\mathcal{P}(c),n\rightarrow\mathcal{P}(n),p\rightarrow\mathcal{P}(p),f\rightarrow\mathcal{P}(f),t\rightarrow\mathcal{P}(t)}(B_{12})
\end{aligned}
$$

Figure 16: Relational Translation of the Reenactment Query for Transaction T_2 from the Running Example

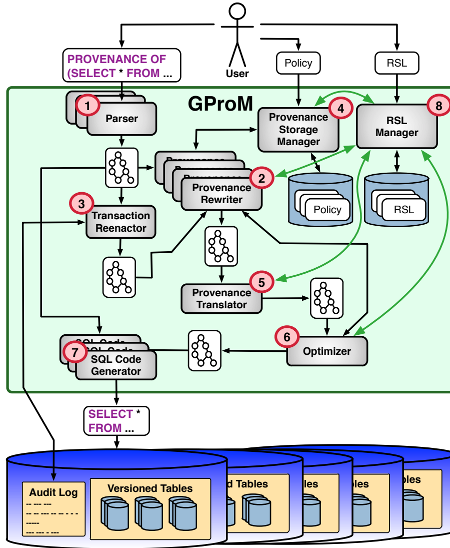

Figure 17: GProM Architecture

Proof. Note that the rewrite rules are compositional. We can prove the correctness of the rewrites Rew by an induction over the number of operators in an reenactment query. The theorem then follows from the fact that $\sigma_{\mathcal{A}_1\vee\ldots\vee\mathcal{A}_n}$ filters out tuples affected by the transaction. □

7. IMPLEMENTATION

We have implemented reenactment-based provenance computation in our **G**eneric **Pro**venance **M**iddleware (GProM) system. GProM is a middleware system that runs over standard relational database systems and adds provenance support to these systems. The system is implemented in C and uses the standard C client-library of the backend database to execute database operations. SQL statements with provenance requests are first transformed into an internal representation that is similar to relational algebra graphs (we use graphs instead of trees to explicitly encode reuse of subqueries). We refer to this model as the algebra graph model (*AGM*). If the input SQL statement contains requests for transaction provenance, then the transaction reenactor module accesses the audit log of the backend database to gather information about the involved transaction(s) and constructs a reenactment query. This reenactment query is then passed to the provenance rewriter that rewrites the reenactment query to compute its provenance. Afterwards, the optimizer of GProM simplifies and optimizes the AGM graph using a set of heuristic rules. The resulting AGM graph is then translated into executable SQL code by the *SQL Code Generator* module. For a more thorough overview of our vision for this system and its unique features (provenance storage policies, database-independence, heuristic and cost-based optimizations, ...) see [4].

Figure 17 shows an overview of the system. The user interacts with the system through an extension of the underlying database system's SQL dialect. SQL is augmented with language constructs for retrieving provenance. For instance,

```
PROVENANCE WITH TABLE R OF TRANSACTION T
```

would compute the relational encoding of the partial prov-

enance $R[T]$ of transaction T for relation R. Provenance requests are considered queries and can be used in any place of an SQL statement where a query can occur. For example, provenance requests can be nested inside queries, used in view definitions, and their result can be stored. For example:

```
INSERT INTO X (PROVENANCE OF ...)
```

8. OPTIMIZATIONS

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 a queries where the left and right input of each such union is again a union operation. Unless intermediate results are reused, this leads to exponentially many union operations in the number of updates in the transaction. We could force the backend DBMS to reuse intermediate results by materializing them, but this will lead to unnecessary overhead. A more efficient approach, that we choose in our implementation, is to use a different technique for reenacting `UPDATE` statements. Assume we have a conditional expression construct

$$\text{if } (\theta) \text{ then } e_1 \text{ else } e_2$$

available that returns e_1 if condition θ evaluates to true and e_2 else. In SQL such a construct can be implement as `CASE WHEN` θ `THEN` e_1 `ELSE` e_2 `END`. Using this construct, we can implement the relational reenactment of an update $\mathcal{U}[\theta, A, T, \nu](R)$ using a projection over relation R on conditional expressions derived from A and an additional conditional expression to determine the value of the boolean version annotation attribute for this update. Let A' denote a list of expressions that is derived from A as follows. For each attribute a not updated by A we add a to A'. For each update expressions $a \rightarrow e$ we add if (θ) then a else e to A'. For the version annotation attribute for the update, we add if (θ) then $true$ else $false$. The resulting reenactment query for an update is:

$$\Pi_{A'}(R[T, \nu])$$

The same technique can also be applied to delete statements by using a condition if (θ) then $false$ else $true$ for the version annotation attribute. This rewriting technique results in reenactment queries that reference their inputs exactly once and, thus, avoids the potential exponential blowup in query size.

Example 15. *Consider an update $\mathcal{U}[b = 10, a \rightarrow a + 5, T, \nu](R)$ over a relation $R(a, b)$ being the ith update in transaction T. The rewritten reenactment query for this update using the new conditional construct is*

$$\Pi_{if\ (b=10)\ then\ a+5\ else\ a, b, if\ (b=10)\ then\ true\ else\ false \rightarrow \mathcal{A}_i}(\text{Rew}(R[T, \nu]))$$

Theorem 8. *Let $\mathcal{U}[\theta, A, T, \nu](R)$ be an update statement. Let Q denote the rewritten update reenactment query as defined in Section 6.4 and Q' denote the rewritten reenactment query as defined above. Then*

$$Q \equiv Q'$$

Proof. Consider the an input tuple t of the union in the rewritten reenactment query Q. If t fulfills $theta$ then it will appear in the left input of the union and, thus, be updated using projection A and by adding $true$ as the value of the version annotation attribute. In Q' the conditions of the conditional constructs if (θ) then e_1 else e_2 evaluate to true returning $t.A$ and adding $true$ as the value for the version annotation attribute. If t does not fulfill $theta$ then t will only appear in the right input of the union, its values will not be updated, and $false$ will be added as the value of the version annotation attribute. The same applied for Q', because the condition of the conditional constructs will evaluate to false and the original values of t's attribute values will be returned (and the version annotation attribute will be set to $false$). □

For RC-SI transactions we can apply the same technique to encode the version selection operator. For example, $\theta \wedge V_b < 13$ can be expressed through conditional constructs if $(\theta \wedge TT_b < 13)$ then e_1 else e_2.

8.2 Prefiltering Partial Provenance

The relational encoding of reenactment queries introduced in Section 6.4 filters out tuples from the provenance of a transaction T that were not affected by xid by applying a selection to the result of the provenance computation that removes tuples where all version annotation are false, i.e., that are tuples that were not effected by any update of the transaction. This has the drawback the reenactment query is evaluated over all tuples from $R_{Start(T)}$. We now discuss two optimizations that filter out tuples that were not updated early on during reenactment.

8.2.1 Prefiltering using Update Conditions

The performance of the naive method can be improved if we can determine upfront which tuples will be affected by a transaction. Consider a transaction $T = (u_1, \ldots, u_n)$ and a tuple t valid at transaction start. Tuple t may be modified by a subset (potentially empty) of the updates of xid. If t is affected at all, then there has to exist a first update in T that modified tuple t. Let u_t denote this update. This first update will see the version of t that was valid at transaction start, because all update u_i with $i < t$ have not updated 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, the set of tuples fulfilling the condition $\theta_1 \vee \ldots \vee \theta_n$ where θ_i is the condition of the ith update operation in transaction T are exactly the tuples that where updated by T. Note that this approach is not applicable to a relation R if there exists an insert in the transaction with a query that accesses relation R. For such relations we have to fall back to the previous approach presented in Section 6.4.

Theorem 9 (Prefiltering). *Let T be an SI transaction and R a relation affected by T that is not accessed by any insertion query in T. The rewritten reenactment query $\text{Rew}(Q^R(T))$ which applies the selection $\mathcal{A}_1 \vee \ldots \vee \mathcal{A}_n$ is equivalent to the query derived from $\text{Rew}(Q^R(T))$ by applying a selection on $\theta_1 \vee \ldots \vee \theta_n$ to $R_{Start(T)}$.*

Proof. This equivalence holds as long as $\sigma_{\theta_1 \vee \ldots \vee \theta_n}(R)$ is equal to the set of tuples that were updated. Consider that there exist a tuple t that fulfils condition $\theta_{pre} = \theta_1 \vee \ldots \vee \theta_n$.

Then there has to exist at least one θ_i that is fulfilled by t. Let θ_j denote the condition from the earlest update such that $t \models \theta_j$. Then t was not updated by any u_i with $i < j$ and would be updated u_j. Thus, we have the required contradiction. Now consider that there exists a tuple t that was updated, but does not fulfill θ_{pre}. This directly leads to a contradiction, because if t does not fulfill any θ_i it cannot possibly be affected by any update from T. □

For RC-SI transactions we can apply a similar technique to filter tuples from a time slice $R_{[\nu_b,\nu_e]}$ using the condition shown below.

$$(\theta_1 \wedge TT_B \leq \nu(u_1) \wedge TT_e > \nu(u_1)) \vee \dots$$
$$\vee (\theta_n \wedge TT_B \leq \nu(u_n) \wedge TT_e > \nu(u_n))$$

8.2.2 Prefiltering Using Committed Tuple Versions

Note that the version of the database at commit of transaction T will contain all tuple versions created by the transaction. Let us require that in a snapshot each tuple version has a column xid that stores which transactions created that tuple version. 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 (for an SI transaction), we can join the result of this query with $R_{Start(T)}$ to filter out tuple version that were updated by T. Here we assume that the database system uses some unique internal identifier for tuples that do not change between versions. This assumption holds for many DBMS. Assume that the DBMS stores these identifiers in a column *tid*. We can use this *tid* attribute to join the committed tuple versions with their counterpart at transaction start. Like the technique presented in the previous subsection this approach is only applicable to relations that are not accessed by any insertion query in the transaction.

THEOREM 10 (HISTORY JOIN). *Let T be an SI transaction and R a relation affected by T. Consider reenactment query $Q^R(T)$. The query $\text{REW}(Q^R(T))$ is equivalent to the query derived from $\text{REW}(Q^R(T))$ by replacing $R_{Start(T)}$ with*

$$\Pi_R(R_{Start(T)} \bowtie_{tid=tid'} \Pi_{tid \to tid'}(\sigma_{xid=T}(R_{End(T)})))$$

PROOF. Recall that our assumption is that a *tid* attribute is not affected by any update. We know that the value of this attribute will be the same in the original version of a tuple at transaction start and at commit time. Thus, the join will return all tuples that will get updated by transaction T. □

We can adapt this technique for RC-SI transactions by using a time slice instead of a snapshot.

9. EXPERIMENTS

We have evaluated our system using a synthetic workload to test how the approach scales in the size of the database, the size of historical data, the number of updates per transactions, and the number of affected tuples per update. We also measure the storage and runtime overhead of audit logging and history maintenance. All experiments were executed on a machine with 2 x AMD Opteron(tm) Processor 4238 CPUs (12 cores in total), 128 GB RAM, and 4 x 1TB 7.2K HDs in a hardware RAID 5 configuration.

Tables	#Rows	History Size
R10K, H0	10,000	0%
R100K, H0	100,000	0%
R1000K, H0	1000,000	0%
R1000K, H10	1,000,000	10%
R1000K, H100	1,000,000	100%
R1000K, H1000	1,000,000	1,000%

Figure 18: Tables Definitions

9.1 Dataset Description

In this experiment we use a single table with five numeric columns. Values of these attributes are chosen randomly using a uniform distribution with the exception of the primary key attribute which is automatically generated. We create several versions of this table by varying the following parameters:

- **Table Size:** R is the size of the table in number of tuples.
- **History Size:** H is the size of the history in percent relative to the table size.

The different configuration we have used are shown in Figure 18. We created variants $R10K$ (10,000 rows), $R100K$ (100,000 rows) and $R1000K$ (1000,000 rows) with no significant history ($H0$), i.e., the history only contains old versions of rows modified by our workload described next. Additionally, we generated three tables with 1,000,000 rows ($R1000K$) and different history sizes: $R1000K, H10$ with 10% history (100,000 rows in the history), $R100K, H100$ has 100% history (1,000,000 rows), and $R1000K, H1000$ has 1,000% history (10,000,000 rows).

9.2 Workload Description

The first workload we are considering consists of transactions that consists solely of update statements. Updated rows are randomly chosen. We vary the following parameters:

- **Number of Updates:** U is the number of update statements in the transaction. For example, $U10$ is a transaction with 10 updates.
- **Isolation Level:** SI and RC represent the isolation levels `SERIALIZABLE` and `READ COMMITTED`.
- **Tuples Affected:** T is the number tuples affected by each update.

In our experiments each transaction was executed under isolation levels `READ COMMITTED` (RC) and `SERIALIZABLE` (SI). The default mode for experiments is SI. Unless stated otherwise, each update in a transaction will affect one row from the updated table ($T1$). The row to be updated is selected using a selection condition on the primary key column of the table.

9.3 Compared Methods

We compare different configurations for computing the provenance of transactions in the workload. Each of these configurations activates/deactivates a subset of the optimizations we have described in Section 6.

- **NoOpt (N)**: Computes the provenance of all rows in a table, even rows that were not affected by the transaction. That is we do not apply the filter condition on the version annotation attributes.

- **Prefilter (P)**: Only computes the provenance of rows updated by the transaction. These rows are filtered in a first step using a selection condition which is a disjunction of all conditions for the updates of the transaction as described in Section 8.2.1. The database system was instructed to materialize the intermediate result for each update in a transaction using temporary tables.

- **Prefilter+Merge (PM)**: Provenance is computed in the same way as for the *Prefilter* configuration. However, where possible we merge operators (particularly, projections) to reduce the number of query blocks instead of materializing each intermediate result.

- **HistJoin (H)**: This configuration also computes only the provenance of rows updated by the transaction. These rows are filtered in a first step by retrieving ROWIDs for all rows updated by the transaction from the history at commit time (database system X records which transaction has created a tuple version in a pseudo-column `VERSION_XID`) and joining the result with the version at transaction start to get the initial version of the affected rows seen by the transaction's updates. Similar to the *PM* method we also collapse query blocks in this configuration.

Each experiment was repeated 100 times and we report the average runtime.

9.4 Table Size and Updates per Transaction

In this experiment we compute the provenance of transactions varying the number of update operations per transaction ($U1$, $U10$, $U100$, and $U1000$) and the size of the database ($R10K$, $R100K$, and $R1000K$). We use the table without significant history and compute the *NoOpt* and *Prefilter+Merge* configurations. Figure 19 shows the runtime of these provenance computations. The approach scales linearly in the database size and number of updates per transaction. By reducing the amount of data to be processed by the rewritten reenactment query and by reducing the number of query blocks, the *PM* approach is up to three orders of magnitude faster then the naive *NoOpt* configuration.

9.5 History Size

Figure 20 shows the effect of history size on provenance computation performance. We have computed the provenance of a transaction with 10 updates ($U10$) over tables with 1M rows ($R1000K$) and different history sizes: $H0$, $H10$, $H100$, and $H1000$. The *NoOpt* configuration exhibits almost constant performance. The runtime is dominated by evaluating the rewritten reenactment query over 1M input rows (all the rows present in one version of the table) hiding the impact of scanning the history. Since, we have not created any indexes on the history tables, the *Prefilter+Merge* approach only has the advantage of processing less tuples in the actual provenance computation, but still has to scan most of the history to filter out the rows that were updated by the transaction.

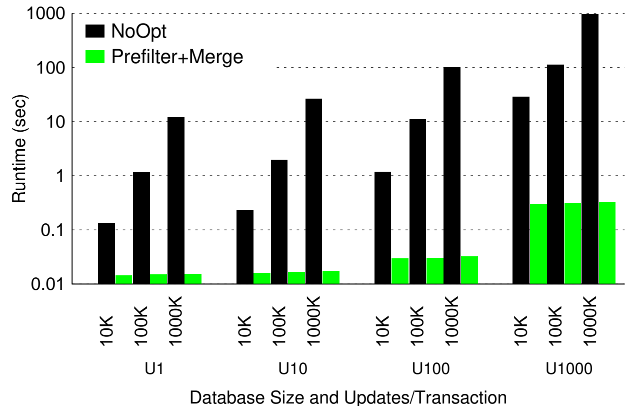

Figure 19: Table Size and Updates/Transaction (SI H0)

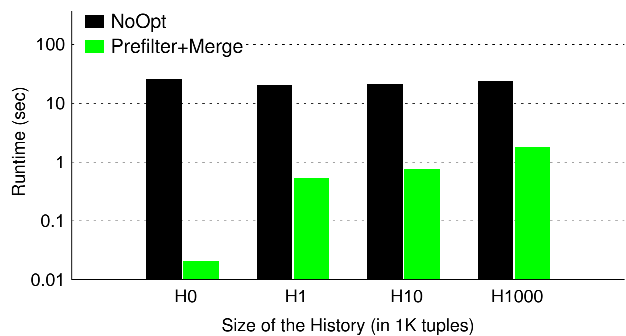

Figure 20: History Size (SI-U10-R1000K)

9.6 Isolation Levels

Figure 21 presents the result of an experiment over table $R1000K, H1000$. We have executed transactions with different number of update operations ($U1$ to $U1000$) under isolation levels SI and RC. The runtime of *NoOpt* is not affected by the choice of isolation level, because as explained in Section 6 the only difference between SI and RC reenactment is that a different version of the database is accessed and that we need to check for each row and update whether this row version is visibile to an update. This check results in more complex expressions in the conditional projections that reenact an update. However, the impact of this additional expressions is negligible compared to the cost of scanning the whole table and large parts of its history as well as producing 1M output rows. For the more efficient *Prefilter+Merge* configuration this effect is more noticeable, especially for larger number of updates per transaction. Furthermore, the condition for the initial selection that filters out rows that were updated by the transaction (compare Section 8.2.1) is more complex for RC. Note that the*NoOpt* method did not finish within the allocated time budget for 1000 updates per transaction.

9.7 Comparing Optimization Techniques

In the next experiment we compare the performance of our approach using different optimization techniques. Figure 22 shows the result for transactions with different number update operations ($U1$, $U10$, and $U100$) using table $R1000K, H1000$. In addition to *NoOpt* and *Prefilter+Merge*

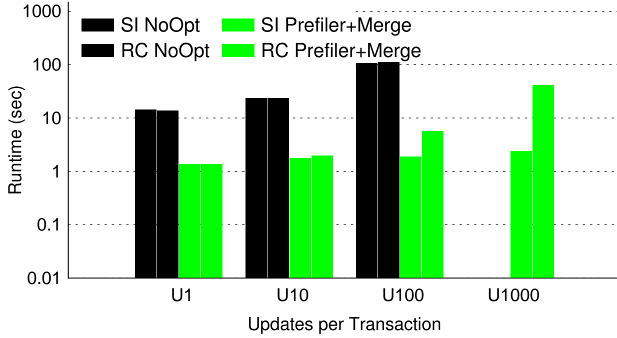

Figure 21: Isolation Levels (R1000K-H1000)

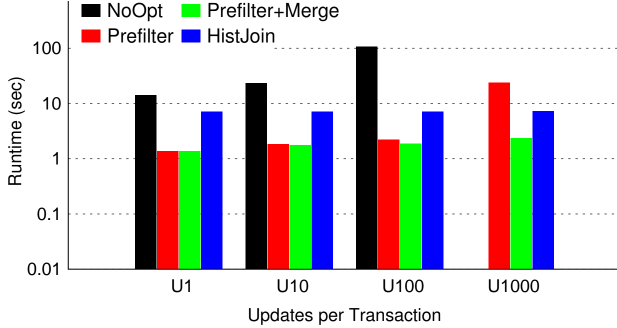

Figure 22: Optimization Methods (SI-R1000K-H1000)

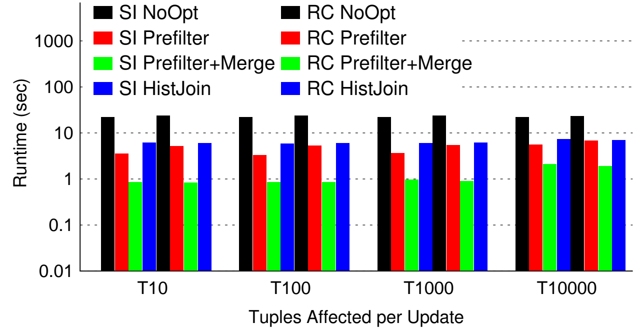

Figure 23: Affected Tuples per Update (SI-R1000K-H1000-U10)

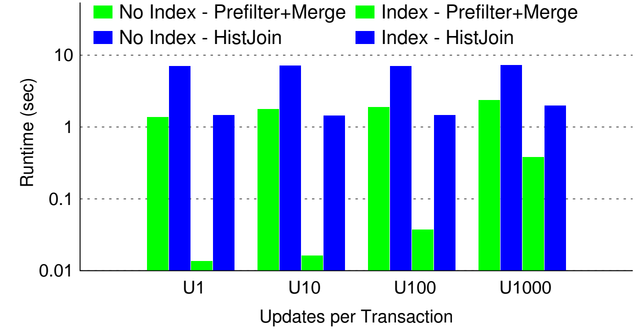

Figure 24: Index vs. No Index (SI-R1000K-H1000)

we now also consider the *HistJoin* and *Prefilter* configurations. Compared with *Prefiler*, the *Prefilter+Merge* configuration benefits from the applied query simplifications and avoids materialization. This optimization is more effective for transactions with more update, because the reenactment queries for such transactions are increasingly complex and contain more query blocks. While resulting in roughly 20% improvement for U100, it improves the runtime by a factor of roughly 10 for U1000. The cost of *Prefilter+Merge* is affected by the first selection that has to be applied to 1M rows (recall that no index was created for the history table). This condition is linear in size of the number of updates in the transaction. The runtime of *HistJoin* is almost not affected by the U parameter, because it is dominated by the join between two historical versions of the table. *Prefilter+Merge* outperforms *HistJoin* by a factor of roughly 3 to 4.

9.8 Number of Effected Rows Per Update

So far we have only considered updates that affect a single row each. Figure 23 shows the runtime of provenance computations for transactions with 10 update operations ($U10$) where each update modifies 100, 1000 or 10000 rows. We have used the table $R1000K, H1000$ in this experiment. As evident from Figure 23, the runtime is not significantly effected when increasing the number of effected rows per update; the runtime is dominated by scanning the history and filtering out updated rows (*Prefilter+Merge*) or the self-join between historic versions of the table (*HistJoin*). Increasing the T parameter by 3 orders of magnitude increase the runtime by about 150% (*Prefilter+Merge*) and 20% (*HistJoin*).

9.9 Index vs. No Index

We now study how the performance of provenance computation can be improved if indexes are created on the history tables. We have replicated the indexes defined for the regular table to its corresponding history table. Figure 24 shows a comparison of the results with and without using indexes. We have used the $R1000K-H1000$ table in this experiment and have varied the number of update operations per transaction ($U1$ to $U1000$). In this experiment we omit the *NoOpt* (this method does not benefit from indexes) and *Prefilter* (this method is consistently outperformed by *Prefilter+Merge*) configurations. Using indexes improves execution time of queries that apply *Prefilter+Merging* considerably.

9.10 Inserts and Deletes

The next experiment measures the performance of provenance computation for transactions that include inserts and deletes in addition to updates. We have used the $R1000K$ table in this experiment. Each operation in a transaction is chosen randomly from the following operations:

- **25% probability**: An update as used in the previous experiments ($T1$)
- **25% probability**: An insert that inserts one new tuple
- **25% probability**: An insert that inserts the result of a query over a different table (1 tuple inserted)
- **25% probability**: A delete that removes 1 randomly chosen tuple

Figure 25 shows the results for transactions with 20 statements each ($U20$) and varying history size ($H10$ to $H1000$).

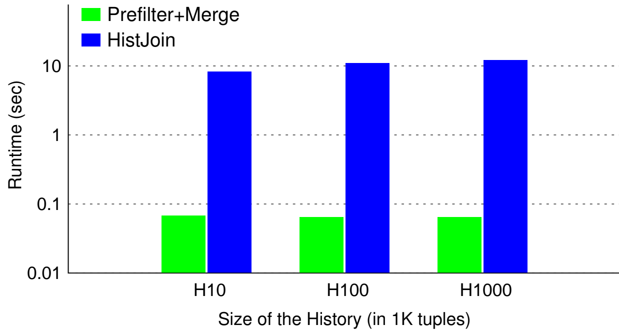

Figure 25: Transactions with Inserts, Deletes, and Updates

Affected Tup. / Update	$R1000K$	$R1000K, H1000$
T_{10}	0.001968	0.007516
T_{100}	0.002622	0.009495
T_{1000}	0.010371	0.027494
T_{10000}	0.083445	0.628319

Figure 26: Runtime Overhead (Seconds)

As evident from this Figure, the performance is comparable to the performance for only update statements.

9.11 History and Audit Logging Overhead

Our approach relies on audit logging and time travel to reconstruct provenance of past transactions using reenactment. While certainly not as heavy weight as directly computing and storing provenance for all transactions run in a database, logging SQL statements and inserting outdated tuple versions into the history table to enable time-travel does not come for free.

To measure the overhead of audit logging and history maintenance we have created two versions of the workload table - one with audit logging and history maintenance activated ($R1000K, H1000$) and one where these features where deactivated $R1000K$. We have measured the execution time of several $U10$ transactions varying the number of tuples affected by each update (parameter T).

Figure 26 shows the runtime and Figure 27 shows storage overhead incurred by these features. When creating the initial history of our workload table, the runtime is increased at least by 165%.

The storage size for the table without history is 21 byte per row. If time travel is activated, this results in an overhead of 37 bytes/row for tuples that are currently valid. On average the storage size of outdated row versions is 65 bytes/row. An size of an entry in the audit log is on average 443 bytes. Note that the audit log overhead has to be paid per executed statement independent of the number tuples affected by a statement.

9.12 Summary

Our experimental evaluation confirms the efficiency and scalability of our approach - the presented techniques easily scale to tables with millions of rows, large transactions (1000 update statements), large number of updated tuples, and large histories. Replicating the indexes defined for a table to the corresponding history table can improve performance significantly, because our reenactment approach effectively exploits available indexes. The proposed optimizations increase performance by several orders of magnitude.

	$AverageRowSize(Bytes)$
Regular Row Size	21
Overhead for Current Rows	37
Overhead for Historical Rows	65
Audit Log (per Statement)	378

Figure 27: Storage Overhead

10. CONCLUSIONS

We have presented the first solution for computing the provenance of transactions run under SI and RC-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 update operations and transactional semantics using version annotations. The resulting MV-semirings provide full account of the derivation history of tuples that were produced by concurrent transactions. We present an relational encoding of MV-semiring relations and demonstrate how reenactment queries in the MV-semiring model can be constructed using an audit log and be reduced to standard relational queries using time travel. Thus, our approach can retroactively compute the provenance of transactions using a standard DBMS and without incurring any runtime or storage overhead apart from auditing logging and maintaining historical relations for time travel. Our experimental evaluation demonstrates that our implementation in the GProM system running over a commercial DBMS can efficiently compute the provenance of transactions and scales to large databases, histories, and complex transactions.

Extending reenactment queries for additional concurrency control protocols and to more expressive provenance models (e.g., the tensor-product extension of the semiring annotation framework [3] used to model aggregation or extensions of this framework for set difference [14, 2]) are promising avenues of future work. Reenactment is an elegant mechanism that has many potential applications including determining complete sets of dependent transactions for transaction-backout and computing historic What-If scenarios by reenacting modified transactions (e.g., "What would have happened if we would have updated accounts using 10% instead of 5% interest?").

11. ACKNOWLEDGMENTS

This research is partially supported by a gift from the Oracle Corporation. We would like to thank the following contributors: Shukun Xie, Bowen Dan, Pankaj Purandare, Zefeng Lin, Ying Ni, Hao Guo, Raghav Kapoor, Jingyu Zhu, and Marcelo Lucera Silva.

12. 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. On the Limitations of Provenance for Queries with Difference. In *TaPP*, 2011.

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

[4] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for

database queries, updates, and transactions. In *TaPP*, 2014.

[5] 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.

[6] 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.

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

[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. ACM, 2013.

[11] M. Cahill. *Serializable Isolation for Snapshot Databases*. PhD thesis, The University of Sydney, 2009.

[12] 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.

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

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

[15] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.

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

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

[18] 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.

[19] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. In *ICDT*, pages 212–224, Saint Petersburg, Russia, March 2009.

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

[21] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation and serializability in sql92 and oracle7. *Oracle White Paper*, 1995.

[22] C. Jensen and R. Snodgrass. Temporal Data Management. *TKDE*, 11(1):36–44, 1999.

[23] G. Karvounarakis. *Provenance in collaborative data sharing*. PhD thesis, University of Pennsylvania, 2009.

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

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

[26] 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.

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

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

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

[30] Oracle. http://www.oracle.com/technology/products/database/oracle11g/pd data-archive-whitepaper.pdf.

[31] D. R. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[32] V. Tannen. Provenance propagation in complex queries. In *In Search of Elegance in the Theory and Practice of Computation*, pages 483–493. 2013.

[33] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal databases: theory, design, and implementation.* Benjamin-Cummings Publishing Co., Inc., 1993.

[34] C. Tilgner, B. Glavic, M. H. Böhlen, and C.-C. Kanne. Declarative Serializable Snapshot Isolation. In *ADBIS*, pages 170–184, 2011.

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

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

[37] 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.