PROVENANCE FOR TRANSACTIONAL UPDATES

BY

BAHAREH SADAT ARAB (A20304723)

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved ______________________________
Advisor

Chicago, Illinois
May 2019

© Copyright by

BAHAREH SADAT ARAB (A20304723)

May 2019

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vi

LIST OF FIGURES . . . xi

LIST OF SYMBOLS . . . xii

ABSTRACT . . . xvi

CHAPTER

1. INTRODUCTION . . . 1

1.1. Motivational Example . . . 2
1.2. A Provenance Model For Queries, Updates, and Transactions 5
1.3. Capturing Provenance With Reenactment . . . 8
1.4. Historical What-if Queries . . . 10
1.5. Applications . . . 10
1.6. Contributions . . . 11

2. BACKGROUND . . . 13

2.1. Relational Data Model . . . 13
2.2. Relational Algebra . . . 14
2.3. Updates and History . . . 16
2.4. The Semiring-Annotation Framework . . . 18
2.5. Concurrency Control Protocol . . . 21
2.6. Summary . . . 26

3. RELATED WORK . . . 28

3.1. Provenance Models . . . 28
3.2. Provenance Systems . . . 30
3.3. Provenance for Updates and Past Operations . . . 31
3.4. What-if Queries . . . 32
3.5. Summary . . . 33

4. THE MV-SEMIRING MODEL . . . 34

4.1. MV-semirings . . . 34
4.2. Version Annotations . . . 37
4.3. MV-semiring Annotation Domain . . . 39
4.4. Normal Form and Admissible Instances . . . 42
4.5. Properties of MV-semirings . . . 43
4.6. Queries . . . 44

4.7. Update Operations . . . 47
4.8. Transactions and Histories . . . 50
4.9. Summary . . . 64

5. REENACTMENT . . . 74

5.1. Reenactment Queries . . . 75
5.2. Update Reenactment . . . 75
5.3. SI Reenactment . . . 77
5.4. RC-SI Reenactment . . . 80
5.5. Relational Reenactment using Time Travel and Audit Logging 85
5.6. Summary . . . 90

6. IMPLEMENTATION . . . 93

6.1. System Overview . . . 93
6.2. Support for Updates And Transactions . . . 95
6.3. Heuristic and Cost-based Optimization . . . 97
6.4. Database Independence . . . 98
6.5. Summary . . . 99

7. OPTIMIZATIONS . . . 100

7.1. Reducing MV to Standard Relational Semantics . . . 100
7.2. Prefiltering Provenance . . . 101
7.3. Reducing Relation Accesses . . . 103
7.4. Summary . . . 106

8. HISTORICAL WHAT-IF QUERIES . . . 107

8.1. Motivational Example . . . 107
8.2. Historical What-if Queries . . . 114
8.3. Overview of Our Approach . . . 115
8.4. Incremental Maintenance and Reenactment . . . 118
8.5. Data Slicing . . . 120
8.6. Program Slicing . . . 123
8.7. Symbolic Execution . . . 127
8.8. MILP Compilation . . . 134
8.9. Summary . . . 135

9. APPLICATIONS . . . 139

9.1. Provenance-aware Versioned Dataworkspaces . . . 139
9.2. Post-mortem Debugging of Transactions . . . 141

9.3. Summary 144

10. EXPERIMENTS 146

10.1. Setup and Workload 146
10.2. Performance of Provenance Capture 147
10.3. Querying Provenance 150
10.4. Overhead and Eager Provenance Capture 151
10.5. Answering Historical What-if Queries 152
10.6. Summary 159

11. CONCLUSIONS AND FUTURE PLANS 167

APPENDIX 168

A. PROOFS 169

B. EXAMPLES 206
B.1. First Provenance Capture Example 207
B.2. Second Provenance Capture Example 213

BIBLIOGRAPHY 223

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1.1 Example audit log for a transactional history 3

1.2 Database before execution of T_5 and T_6 4

1.3 Database after execution of T_5 4

1.4 Database after execution of T_6 5

1.5 Relational encoding of the provenance and intermediate results for relation `Account` with respect to Transaction T_5. 5

1.6 Reenactment 8

2.1 Relational algebra operators (set semantic) 15

2.2 Example Algebra Tree 16

2.3 Example Query Result 17

2.4 Grammar defining the syntax of expressions $\mathbf{e}$ and conditional expressions $\boldsymbol{\phi}$ 17

2.5 Equivalence expression $\mathbf{e}$ rules 18

2.6 Semirings and Their Corresponding Models 20

2.7 Relation R and the result of query Q 21

2.8 Equivalence relations for $\mathcal{K}^{\nu}$ 22

2.9 Example Transactional History 23

2.10 Running example database instance 27

2.11 New and modified tuples after execution of the example history under RC-SI (version 26). 27

3.1 Comparison of the functionality provided by provenance systems . 31

4.1 Example audit log for a SI transactional history 38

4.2 Database before execution of T_5 and T_6 38

4.3 Database after execution of T_5 39

4.4 Database after execution of T_6 39

4.5 Example audit log for a RC-SI transactional history 40

4.6 Running example database instance 65

4.7 New and modified tuples after execution of the example history under RC-SI (version 26). 65

4.8 SI historic database definition 66

4.9 RC-SI historic database definition 67

4.10 Bob's Transaction 68

4.11 Bind Parameters for Transactions T_1 and T_2 68

4.12 Execution Order of Transactions T_1 and T_2 68

4.13 History corresponding to T_1 and T_2 69

4.14 Database before execution of T_1 and T_2 ($\nu = 3$) 69

4.15 Database after execution of T_1 ($\nu = 10$) 70

4.16 Database after execution of T_2 ($\nu = 13$) 70

4.17 Database visible to T_2 at its commit ($\nu = 13$) 70

4.18 Provenance restricted to Transaction T_2 71

4.19 Relational encoding of $\mathbb{N}[X]^{\nu}$-relation `account` with annotations restricted to Transaction T_2 ($account[T_2]$) 71

4.20 Evaluating the update of Transaction T_2 72

4.21 Alice's transaction 72

4.22 Database states before and after the execution of Alice's transaction 73

5.1 Structure of the reenactment query for SI 79

5.2 Definition of auxiliary RC-SI reenactment operators 81

5.3 Structure of the reenactment query for RC-SI 83

5.4 Schema and instance of the relational encoding of $R[T]$ 85

5.5 Structural rewrite rules for translating $\mathcal{K}^{\nu}$-semantics reenactment queries into standard relational semantics (bag) 91

5.6 Annotation attributes rules for translating $\mathcal{K}^{\nu}$-semantics reenactment queries into standard relational semantics (bag) 92

6.1 GProM architecture 95

8.1 Running example database instance 109

8.2 Transactional history implementing the new ShippingFee policy and a hypothetical change the policy (update $u_1{}'$ replaces u_1) 109

8.3 Database state after executing the original history 110

8.4 Q_{total} result 110

8.5 Database state based on the historical what-if query 111

8.6 Q_{total} result 111

8.7 The Naive Method 113

8.8 The Proposed Method 113

8.9 $\Delta(Q, H(D), H[\mathcal{M}](D))$ 116

8.10 Running example for evaluating updates over VC-Tables. 137

8.11 Compilation rules for translating constraints into an MILP (the remaining comparison operators are omitted since they can be expressed using negation) 138

9.1 Screenshot of the Debugger GUI 142

9.2 Example transaction 143

10.1 Relation size 161

10.2 History size 161

10.3 Optimization methods 161

10.4 Affected tuples 161

10.5 Index vs. no index 161

10.6 Inserts and deletes 161

10.7 Provenance for TPC-C 162

10.8 Isolation Levels 162

10.9 Aggregation 162

10.10 Query Provenance 162

10.11 Query Vers. Ann. 162

10.12 Updates/Transaction 162

10.13 Dependent Updates 163
10.14 Provenance Retrieval 163
10.15 Runtime Overhead 163
10.16 Naive vs. Mahif 163
10.17 Naive vs. Mahif 163
10.18 Naive vs. Mahif 163
10.19 Breakdown Naive 164
10.20 Breakdown Mahif 164
10.21 Optimization 164
10.22 Dependent Updates 164
10.23 Affected Data 164
10.24 Relation size 164
10.25 Relation size 165
10.26 Relation size 165
10.27 Aggregate Query 165
10.28 #Attributes 165
10.29 #Conditions 165
10.30 Eager vs. reenactment 166
A.1 Cases of how tuple versions with a fixed identifier *id* can occur in $R[\nu_{n+1}]$ and $R[T, \nu_{n+1}]$ 200
B.1 Example Instance For Running Example 208
B.2 Example Audit Log 209
B.3 Updated Example Instances 224
B.4 Example Transaction and Translated Updates 225
B.5 Provenance for the Running Example Transaction 226
B.6 Provenance Attribute Names 226

B.7 Relational encoding of the provenance and intermediate results for relation employee 226

LIST OF SYMBOLS

Symbol	Definition
ν	a time (version)
id	a tuple identifier
T	a transaction
$End(T)$	transaction T's commit time
u	an update operation
$\nu(u)$	the point in time when u was executed
H	a history
R_ν	snapshot of relation R at time ν
$\mathcal{U}_i$	boolean attribute recording whether the version annotation of the ith update of a transaction is present in an annotation
$\mathcal{A}$	a version annotation
$U/I/D/C$	Version annotation (update U, insert I, delete D, or commit C)
$X^{id}_{T,\nu}(k)$	Version annotation denoting that an operation of type X that was executed at time $\nu-1$ by transaction T affected a previous version of a tuple with identifier id and previous provenance k
$\mathbb{I}$	domain of tuple identifiers
$\mathbb{V}$	domain of version identifiers
$\mathbb{T}$	domain of transaction identifiers
$\mathbb{A}$	set of all version annotations
P	finite symbolic expression adhering to the grammar from Eq. (2), page 5
$\mathcal{K}$	a semiring

$\mathcal{K}^{\nu}$	the MV version of semiring $\mathcal{K}$
$+_{\mathcal{K}}$	addition operation of semiring $\mathcal{K}$; represents alternative use of inputs (e.g., union)
$\cdot_{\mathcal{K}}$	multiplication operation of semiring $\mathcal{K}$; denotes conjunctive use of inputs (e.g., join)
$\mathbb{N}[X]$	provenance polynomials semiring
$\mathbb{N}[X]^{\nu}$	MV-semiring corresponding to the provenance polynomials semiring
$[k]_{\sim}$	the equivalence class of a symbolic MV-semiring expression k
$\equiv_{\mathcal{K}}$ /$\sqsubseteq_{\mathcal{K}}$	denotes query equivalence/containment over $\mathcal{K}$-relations
$n(k)$	number of summands in a normalized MV-semiring element k (a sum of subexpressions which do not contain addition)
$k[i]$	the ith summand in a normalized MV-semiring element k
$\mathcal{U}[\theta, A, T, \nu](R)$	update operator: updates tuples in R that fulfill θ using the projection expressions in A
$\mathcal{I}[Q, T, \nu](R)$	insert operator: inserts the result of query Q into R
$\mathcal{D}[\theta, T, \nu](R)$	delete operator that removes all tuples that fulfill θ
$\mathcal{C}[T, \nu](R)$	commit operator
$\textsc{com}[T, \nu](k)$	Function that wraps input k into a commit annotation if the outermost version annotation of k corresponds to an update of transaction T
$f_{id}(T, \nu, t, k)$	Skolem function used to create unique tuple identifiers
$R[\nu]$	version of relation R at time ν
$R[T, \nu]$	version of relation R as seen by transaction T at time ν
$R[T]$	relation R restricted to provenance of transaction T
validAt	function that returns 1 if part of an annotation of a tuple is valid at a given point in time and 0 otherwise

UPDATED	predicate that checks whether a transaction has overwritten an annotation (updated or deleted the annotated tuple)
$R_{ext}[T, \nu]$	version of relation R that is visible to an update of transaction T executed at time ν under RC-SI
VALIDEX	returns 0 if the tuple version has been replaced with a new updated version and 1 otherwise
VALIDIN	returns 1 for a tuple version in $R[T, \nu]$ if it was created by previous updates of T and 0 otherwise
$\alpha_{X,T,\nu}(R)$	annotation operator that wraps every summand in an the annotation of a tuple in a version annotation $X_{T,\nu}$ where $X \in \{I, U, D, C\}$
$\mathbb{R}(X)$	reenactment query for operation/ transaction/history X; for transactions and histories this is a set of queries - one per relation R in the database schema
$\mathbb{R}^R(T)$	Denotes the reenactment query for transaction T that returns the updated version of relation R
LAST(T, R, ν)	denotes the last update executed before ν in Transaction T that updated relation R
$\mu(R_1, R_2)(t)$	version merge operator: merges two versions R_1 and R_2 of a relation R such that the output includes 1) each tuple version that exists in both R_1 and R_2 once and 2) the newer version of each tuple which exists as different versions in both inputs
$isMax(R, k)$	returns 0 when relation R has a newer version of the tuple version encoded as annotation k
$isStrictMax$	a strict version of $isMax$ function that also returns 0 when the tuple version k exists in R
$idOf(k)$	returns the identifier of a tuple version k, i.e., the identifier encoded in the outermost version annotation of k, e.g., $idOf(U^{id_1}_{T,\nu}(x)) = id_1$
$versionOf$	returns the version encoded in the annotation k, e.g., $versionOf(U^{id_1}_{T,\nu}(x)) = \nu$
h	a semiring homomorphism

h_U	homomorphism from $\mathcal{K}^\nu$ to $\mathcal{K}$ that maps an $\mathcal{K}^\nu$ element to an element of the embedded semiring $\mathcal{K}$ by evaluating the expression k (interpreting version annotations as functions $\mathcal{K} \rightarrow \mathcal{K}$)
Q	a query
$\mathcal{RA}^+$	positive relational algebra

ABSTRACT

Database provenance explains how results are derived by queries. However, many use cases such as auditing and debugging of transactions require understanding of how the current state of a database was derived by a transactional history. We introduce an approach for capturing the provenance of transactions. Our approach does not just work for serializable transactions but also non-serializable transaction such as read committed snapshot isolation (RC-SI). The main drivers of our approach are a provenance model for queries, updates, and transactions and reenactment, a novel technique for retroactively capturing the provenance of tuple versions. We introduce the MV-semirings provenance model for updates and transactions as an extension of the existing semiring provenance model for queries. Our reenactment technique exploits the time travel and audit logging capabilities of modern DBMS to replay parts of a transactional history using queries. Importantly, our technique requires no changes to the transactional workload or underlying DBMS and results in only moderate runtime overhead for transactions. Furthermore, we discuss how our MV-semirings model and reenactment approach can be used to serve a wide variety of applications and use cases including answering of historical what-if queries which determine the effect of hypothetical changes to past operations of a business, post-mortem debugging of transactions, and Provenance-aware Versioned Dataworkspaces (PVDs). We have implemented our approach on top of a commercial DBMS and our experiments confirm that by applying novel optimizations we can efficiently capture provenance for complex transactions over large data sets.

CHAPTER 1
INTRODUCTION

Provenance, information about the creation process and origin of data, is critical for many applications including auditing, debugging data by tracing erroneous results back to erroneous inputs, and understanding complex transformations. It is also useful as a supporting technology for integration and probabilistic databases. How to model and capture the provenance of database queries is relatively well understood. Most approaches model provenance as annotations on data [41, 42, 21, 40] and propagate annotations to compute the annotation (provenance) of a query result. That is, the annotation of a tuple t in the result of a query records which input tuples are in tuple t's provenance and how these inputs were combined to derive tuple t. Annotation propagation techniques have been pioneered by systems such as Perm [40], DBNotes [14], Orchestra [51], Orchestra [51], Propolis [32], LogicBlox [50], Qplain [29], Smoke [63], ProvSQL [65], and others. However, many use cases require the user to understand how data was derived by updates executed as part of concurrent transactions which is not supported by current approaches [51, 11, 19, 69, 74]. For instance, tracing a query result tuple back to its provenance in the query input is not sufficient for auditing, because this type of provenance does not explain how the query inputs were created (i.e., inserted or updated by past transactions). It would be useful for a transaction developer to be able to drill into the execution of a past transaction to determine the cause of an erroneous outcome when debugging a transactional history or a query result. In this case, it is not sufficient to debug a transaction's code in isolation since the error may be caused by, e.g., concurrency anomalies which occur under non-serializable isolation levels. We need a model which capture the provenance of update operations and concurrent transactions in addition

to queries. The model must explains which input tuples were used and how these inputs were modified by update operations and concurrent transactions to derive a tuple.

Given the lack of support for transactional provenance, users resort to the *audit logging* and *time travel* functionality natively supported by many DBMS (e.g., Oracle, DB2, SQLServer) for their auditing and debugging needs. Time travel enables access to the transaction time history of relations, i.e., the user can query past committed versions of the database. An audit log records which SQL statements were executed by which user at which time and as part of which transaction. While these features can unearth facts about past operations and database states, there are certain limitations. For example, these features can not be used to track dependencies based on read operations, e.g., how the tuples created by an `INSERT INTO SELECT ...` depend on the tuples accessed by the `SELECT` query. Also, they can not expose which statements of a transaction affected a tuple which is important for debugging transaction execution. Our approach overcomes these limitations.

1.1 Motivational Example

Figure 1.2 shows a relational database which has a relation Account. This relation has three attributes `cust` (the name of customer),`typ` (type of account) and `bal` (the balance of the account). The schema of this relation is `Account(cust,typ,bal)`. Each row represents a tuple. This relation contains three tuples `(Alice, Checking,400 )`, `(Alice, Savings, 1000 )`, and `(Peter, Savings, 4990)`. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations. A transaction is a logical, atomic unit of work that contains one or more of these operations. Database systems ensure consistency under concurrent execution of transactions by using concurrency control protocols. For the following example, we use snapshot

isolation (SI) as a concurrency control protocol. SI guarantees that all reads in a transaction will see a consistent snapshot of the database, and the transaction will commit if none of it's updates conflict with any other concurrent updates executed since that snapshot.

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

Figure 1.1. Example audit log for a transactional history

Example 1. *Figure 1.2 shows an example database storing information about banking accounts and overdrafts. Suppose Bob executed the Transactions T_5 shown in Figure 1.1 under SI. Bob implemented a policy of depositing \$100 bonus to all savings accounts and an depositing additional \$300 bonus to all savings accounts with a balance higher than \$5000. The database instance after the execution of Transaction T_5 is shown in Figure 1.3. Attribute values affected by an update are highlighted in red. Meanwhile, Alice did withdraw money (\$1500) from her checking account which triggered Transaction T_6. This transaction inserts an overdraft record into the relation* `Overdraft(cust,bal)` *since the total balance of Alice's accounts is negative after the withdrawal. The states of the* `Account` *and* `Overdraft` *relations after the execution of both transactions are shown in Figure 1.4. After a while, Alice receives an overdraft notice. She checks her account and is surprised to see that the total balance of her accounts is positive and, thus, she should not have received the \$100 overdraft. In this example, the unexpected result is caused by a concurrency anomaly*

Account

cust	typ	bal
Alice	Checking	400
Alice	Savings	1000
Peter	Savings	4990

Overdraft

cust	bal

Figure 1.2. Database before execution of T_5 and T_6

Account

cust	typ	bal
Alice	Checking	400
Alice	Savings	1100
Peter	Savings	5390

Figure 1.3. Database after execution of T_5

called write-skew [13] which can occur under snapshot isolation. Under snapshot isolation each Transaction T executes over a private snapshot which contains changes made by transactions that executed and committed before T's start. Hence, Transactions T_5 and T_6 did not see each others changes. Transaction T_6 sees the previous balance of $1000 instead of $1100 for Alice's savings account. After the withdrawal of $1500 from her checking account with balance of $400, her checking account balance would be $-1100. Therefore, it computes a total balance of $1000+(-1100) = -100 < 0$ *for her accounts.*

Auditing or debugging errors such as the one illustrated in the example above is virtually impossible without access to past database states and operations. For the above example, an audit log would provide information as shown in Figure 1.1 while time travel gives a user access to the database states as shown in Figure B.1. However, these database states are not very helpful in determining the cause of the overdraft, because Alice's total account balance is non-negative after the execution

Account

cust	typ	bal
Alice	Checking	-1100
Alice	Savings	1100
Peter	Savings	5390

Overdraft

cust	bal
Alice	-100

Figure 1.4. Database after execution of T_6

Account			Provenance for the First Update			Provenance for the Second Update			u_1	u_2
cust	typ	bal	P(cust,u_1)	P(typ,u_1)	P(bal,u_1)	P(cust,u_2)	P(typ,u_2)	P(bal,u_2)	$\mathcal{U}_1$	$\mathcal{U}_2$
Alice	Savings	1100	Alice	Savings	1000	Alice	Savings	1100	T	F
Peter	Savings	5390	Peter	Savings	4990	Peter	Savings	5090	T	T

Figure 1.5. Relational encoding of the provenance and intermediate results for relation `Account` with respect to Transaction T_5.

of both transactions. Technically, once the error is detected, a user with a deep understanding of snapshot isolation may be able to recognize that this particular interleaving of operations can lead to a write-skew. However, even for a power user it would be challenging to determine the cause for such errors if several other transactions were run concurrently with the transactions involved in the error. Thus, this example motivates the need for capturing the provenance of tuples that are updated by concurrent transactions. The provenance of a tuple should record how it was derived from previous tuple versions and by which operations. We now give a brief introduction of our provenance model for transactions and then demonstrate how it can be used to understand unexpected results.

1.2 A Provenance Model For Queries, Updates, and Transactions

Our provenance model called *Multi-version semirings* (*MV-semirings*) records provenance as annotations on tuples. While there are existing solutions for computing

the provenance of updates [51, 69, 19], these approaches do not support transactions and are not integrated with provenance for queries. The annotation of a tuple t in our model is a symbolic expression that encodes **1)** which tuples where used to derive t (variables, e.g., x_1, x_2, ... represent tuples) **2)** how these tuples have been combined (addition and multiplication in this symbolic expression represent alternative and joint use of inputs) so it can also capture provenance of queries over provenance of updates and transactions **3)** which DML operations executed by which transactions at which time did create the annotated tuple version.

Note that we do not consider provenance dependencies at the application side in this work. For instance, consider an application that runs a query, stores the result in a client-side variable, and then uses the variable in an update statement. Detecting such dependencies requires tracking provenance of procedural programming languages which is beyond the scope of this work. Typically, a user would like to be able to drill down into a part of a history instead of tracing the origin of a tuple through the whole history of the database. Our model supports this type of drill down by replacing subexpressions in an annotation with fresh variables to prune parts of the history from a tuple's annotation. The technical details of our model and its relationship to the semiring provenance framework will be covered in Chapter 4.

Tracking Read and Write Dependencies of Tuples. One way to investigate the example error is to determine which tuple versions were used to derive the erroneous overdraft tuple. This would unveil that it was computed based on Alice's savings account balance before the bonus was added by Transaction T_6. Note that this is a read dependency. The second account tuple version from Figure 1.2 was read by the `INSERT INTO Overdraft SELECT ...` statement which was executed by Transaction T_6. Time travel can expose write dependencies caused by updates if a tuple can be identified across versions (e.g., the DBMS uses immutable tuple identifiers). However,

it cannot be used to track read dependencies.

Tracking Applications of Updates. Understanding which statements of which transactions were involved in the derivation of a tuple version is important to answer auditing questions such as "What data was affected by statements executed by a compromised user account?". Audit logs record which statements were executed and when they were executed. However, even when this information is correlated with a transaction time history using time travel, it is highly non-trivial to answer such questions since 1) only write tuple dependencies are available and 2) time travel only exposes committed database states. In our model, this information is readily available in the nesting of version annotations in the annotation of a tuple. For example, consider the second tuple in the database state shown in Figure 1.3. Based on its annotation we know that this tuple version was created by an update of Transaction T_5 which was applied to a tuple created by an insert of Transaction T_1. Furthermore, we know when these operations were executed and when these transactions did commit. By analyzing the annotation of the new overdraft tuple in running example, the version annotations show that none of the updates of Transaction T_5 (adding the account bonuses) did affect the tuple versions on which the overdraft is based on.

Exposing Intermediate States. We develop relational encoding model that can expose intermediate states of relations produced by transactions, e.g., the state of a relation after a particular operation. Furthermore, the encoding records dependencies across such states. This is useful for debugging a transaction's execution, since the user can investigate, e.g., whether and how an update modified a tuple.

Example 2. *Recall that Figure 1.5 shows the provenance of the* `Account` *relation w.r.t. Transaction T_5. The provenance annotation of each tuple is encoded in additional attributes that are added to the schema. A* "provenance" *attribute $P(attr, u)$ stores the value of attribute* attr *for the version of a tuple in the provenance seen by*

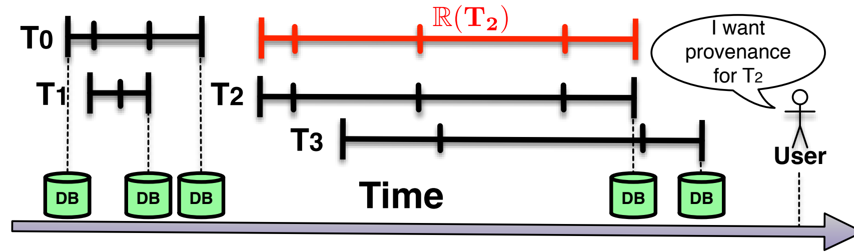


Figure 1.6. Reenactment

the update statement u. We use u_1 *and* u_2 *to denote the updates of Transaction* T_5*. For instance, attributes* $P(bal, u_1)$ *and* $P(bal, u_2)$ *store the balance of a tuple before the execution of update* u_1 *respective* u_2*. The boolean attribute* $\mathcal{U}_i$ *stores whether update* u_i *affected a tuple which can be determined by our provenance model. Suppose manager Tom wants to know which accounts received the $300 bonus implemented by the update* u_2 *and what was the previous balance of these accounts before the bonus. This question can be answered by the SQL query shown below where* Prov *denotes the encoding from Figure 1.5.*

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

Understanding Errors Caused by Concurrency Anomalies. By virtue of exposing intermediate states of relations created by transactions and tracking how a tuple depends on operations and other tuples, our model can be used to detect and analyze errors caused by concurrency anomalies such as the *write-skew* [13] in the running example. Errors caused by anomalies are common, but hard to debug since they may only occur for a particular interleaving of transactions.

1.3 Capturing Provenance With Reenactment

We have developed a provenance capture mechanism that produces a relational encoding of our provenance model for a provenance request submitted by a user and have implemented this mechanism in our provenance database middleware called *GProM* (available at https://github.com/IITDBGroup/gprom). We use *reen-*

actment, a novel technique for replaying a transactional history (or parts thereof) using queries instrumented to capture provenance. Reenactment retroactively captures the provenance of tuple versions produced by a transactional history. Figure 1.6 illustrates how reenactment is applied to retroactively compute provenance for updates and transactions based on replay with provenance capture. Consider the database states induced by a history of concurrently executed transactions. With our approach, a user can request the provenance of any transaction executed in the past, e.g., Transaction T_2 in the example. Using reenactment, a temporal query is generated that simulates the transaction's operations within the context of the transactional history and this query is instrumented for provenance capture. This so-called reenactment query is guaranteed to return the same results (updated versions of the relations modified by the transaction) as the original transaction. In the result of the reenactment query, each tuple is annotated with its complete derivation history: 1) from which previous tuple versions was it derived and 2) which updates of the transaction affected it. Notably, our approach does not require any eager materialization of provenance during transaction execution. Hence, we avoid paying the runtime and storage overhead of provenance capture for every transaction executed by the system. Reenactment solely relies on the information provided by audit logs and time travel and is expressible in SQL. As we demonstrate in Chapter 10.4, the overhead of activating these features is quite manageable (less than 20% for the workloads we considered) and significantly less than the overhead of capturing provenance eagerly during transaction execution. Many users that would be interested in transaction provenance already make extensive use of the auditing and time travel features of current DBMS. Our approach does not result in any additional overhead for these users. Another advantage of our approach is that it requires no modifications of the underlying DBMS and transactional workload.

1.4 Historical What-if Queries

We introduce *historical what-if query*, a new type of predictive query that determines the effect of a hypothetical change to past operations of a business (transaction executions). For example, consider Figure 1.1, we can answer a historical what-if query like *"what would be the state of the* `Account` *relation if they applied additional $100 bonus instead of $300 to all savings accounts with a balance higher than $5000?"*. This type of queries give user more insight about their past policies and help them to improve their business operations in future. We develop efficient techniques for answering historical what-if queries which use reenactment to determine how a modified history affects the current database state. We also propose novel optimizations including for answering such queries like program and data slicing techniques that determine which updates and data can be excluded from reenactment without affecting the result.

1.5 Applications

As an extension of this research, we introduce set of applications which use the MV-semiring model and reenactment. We can apply our approach and extend it in other applications such as Provenance-aware Versioned Dataworkspaces (PVDs), and post-mortem debugging of transactions. PVDs is a sandboxed environment in which users can apply or undo changes to their data and workflows easily. Another interesting application is post-mortem debugging of concurrent transactions. It is useful for developers as databases do not provide tools for debugging concurrent execution of transactions and it is hard to detect concurrency anomalies without using such tools.

1.6 Contributions

The main contributions of this work are:

- We introduce *multi-version semirings* (MV-semirings), a provenance model for database queries, updates, and transactions. In our model, tuples are annotated with symbolic expressions that model dependencies among tuples and which operations affected a tuple. We show how to reduce provenance to transactions of interest. We use a relational encoding of our model to be able to query provenance.
- We introduce *reenactment*, a technique for replaying a transactional history using queries. Importantly, the reenactment query for a transaction T is equivalent to T within the context of a history under MV-semiring semantics, i.e., it returns the same database state and has the same provenance. We reduce reenactment queries with MV-semiring semantics to queries in standard SQL that return a relational encoding of provenance.
- We develop optimizations for reenacting SI and RC-SI transactions including alternative ways of encoding reenactment as SQL queries and filtering unrelated information from the provenance early on.
- We define historical what-if query, anew type of predictive query. We discuss how our reenactment approach is useful for answering historical what-if queries which compute the effect of hypothetical changes to past update operations and transactions.
- We present optimization methods such as programming and data slicing methods which can be applied for answering historical what-if queries.
- We present set of applications as an extension of our work using the MV-semiring model and reenactment such as Provenance-aware Versioned Data-

workspaces, and post-mortem debugging of transactions.

- Our experiments demonstrate that 1) provenance capture based on reenactment is efficient and scales to large databases, complex transactions, and large number of updates; 2) the storage and runtime overhead incurred for running transactions when time travel and audit logging is activated is tolerable and significantly smaller than the overhead of eagerly capturing and materializing provenance during transaction execution ; and 3) our reenactment can be used for answering historical what-if queries efficiently.

The remainder of this thesis is organized as follows. We review necessary background in Chapter 2 and related work in Chapter 3. We introduce our provenance model in Chapter 4. We cover reenactment our in Chapter 5, discuss implementation in Chapter 6 and optimizations in Chapter 7. We present our solution for answering historical what-if queries in Chapter 8, and discuss additional applications for the MV-semiring model and reenactment in Chapter 9. We present experimental results in Chapter 10. We demonstrate conclusion and future plan in Chapter 11. Proofs are available in Appendix A and comprehensive provenance capture example are presented in Appendix B.

CHAPTER 2
BACKGROUND

I introduce necessary database definitions, notations, and semiring annotated data as we extend this provenance model for update operations and concurrent transactions. Our model captures provenance for concurrent transactions that were executed under two common concurrency control protocols. We also discuss these concurrency control protocols in this chapter.

2.1 Relational Data Model

The relational data model is the primary data model of most commercial and open-source database systems. This model was first proposed by Edgar F. Codd. [27] in 1969. In the relational model, data is represented as relations, i.e., sets of tuples with a common schema. Each tuple of a relation is composed of a list of attribute values. The *schema* of a relation defines what attributes each tuple in the relation has. Formally, we distinguish between the schema (structure) and instance (data) of a relation as defined below. The schema of a relation includes its attributes, and optionally the types of the attributes and constraints on the relation such as primary and foreign key constraints. The schema of a relation refers to its logical design, while an instance of the relation refers to its contents at a point in time.

Definition 1. *A database schema* $\mathbf{D} = \{\mathbf{R_1}, \ldots, \mathbf{R_n}\}$ *is a set of relation schemas* $\mathbf{R_1}$ *to* $\mathbf{R_n}$*. A relation schema* $\mathbf{R}(\mathbf{A_1}, \ldots, \mathbf{A_n})$ *consists of a name (R) and a list of attribute names* A_1 *to* A_n*. The arity of a relation schema is the number of attributes in the schema.*

Definition 2. *Let* $\mathcal{U}$ *be a universal domain of values. An instance* R *of a relation schema* $\mathbf{R}$ *(sometimes also called a relation) is a subset of* $\mathcal{U}^n$*, i.e., a set of tuples*

with the same arity as the schema and values from $\mathcal{U}$. An instance D of a database schema $\mathbf{D}$ is a set of relation instances - one for each relation schema in $\mathbf{D}$. We use $\textsc{Sch}(R)$ *as an alternative to $\mathbf{R}$ to denote the schema of a relational instance R.*

This definition of relation instances is often called *set semantics*, because each relation is a set of tuples. Implementations of the relational model which use the SQL query language (essentially all implementations of relational databases) use a slightly different model called *bag semantics* where a relation may contain multiple duplicates of the same tuple. Formally, this can, e.g., be achieved by modelling a relation as a function from $\mathcal{U}^n \to \mathbb{N}$ that associates each tuple with a multiplicity (the number of times it occurs in the relation) and maps tuples that do not occur in the relation to 0.

Definition 3. *Let $\mathcal{U}$ be a universal domain of values. An instance R of a relation schema $\mathbf{R}$ under bag semantics is a function $\mathcal{U}^n \to \mathbb{N}$ with finite support $\mid \{t \mid R(t) \neq 0\} \mid$. We use t^m to denote that tuple t occurs with multiplicity m, i.e., $R(t) = m$.*

2.2 Relational Algebra

Relational algebra is a procedural query language. As the name suggests it is an algebra of relations. The relational algebra provides a set of operations that take one or more relations as input and return a relation as an output. There are well-known methods for translating between relational algebra and SQL, the query language used by most database systems. Thus, we can study provenance computation and optimization for relational algebra and the results are guaranteed to translate to the corresponding features in SQL.

Definition 4. *Relational algebra is an algebra of relations. The operators of the variant of relational algebra we use in this work are defined in Figure 2.1.*

Operator	Definition
σ	$\sigma_\theta(R) = \{t \mid t \in R \wedge t \models \theta\}$
Π	$\Pi_A(R) = \{t.A \mid t \in R\}$
$\cup$	$R \cup S = \{t \mid t \in R \vee t \in S\}$
$\cap$	$R \cap S = \{t \mid t \in R \wedge t \in S\}$
$-$	$R - S = \{t \mid t \in R \wedge t \notin S\}$
$\times$	$R \times S = \{(t,s) \mid t \in R \wedge s \in S\}$
$\bowtie$	$R \bowtie_\theta S \equiv \sigma_\theta(R \times S)$

Figure 2.1. Relational algebra operators (set semantic)

Since the output of an operator in relational algebra is again a relation, it allows us to express complex queries by combining multiple operators. We now discuss the definitions of the relational algebra operators for set semantic shown in Figure 2.1. Here we use the set semantics. Selection $\sigma_\theta(R)$ returns all tuples from relation R which satisfy the condition θ. Projection $\Pi_A(R)$ projects all input tuples on a list of projection expressions. Here, A denotes a list of expressions with potential renaming (denoted by $e \rightarrow a$) and $t.A$ denotes applying these expressions to a tuple t. Union $R \cup S$ returns the set union of tuples from relations R and S. Intersection $R \cap S$ returns the tuples which are both in relation R and S. Difference $R - S$ returns the tuples in relation R which are not in S. Crossproduct $R \times S$ returns all possible combinations of two tuples - one from relation R and one from relation S. $R \bowtie_\theta S$ returns all combinations of tuples from R and S that match the condition θ. A join $R \bowtie_\theta S$ can be equivalently expressed as $\sigma_\theta(R \times S)$. Example 3 shows a relational algebra expression and its representation as an operator tree (graph).

Example 3. *The relational algebra expression shown below returns customer's account information such as name of customers (`cust`), type of their account (`typ`), and the balance of their account (`bal`) which have Overdraft. We have split the query into two parts: the first part (j) matches accounts to their related overdrafts. The second part uses selection to narrow down the result to orders that have bal less than $0 in their overdraft and uses projection to return the attributes we are interested in. The algebra tree representation of this query is shown in Figure 2.2. The result of executing this query over the instance from Figure 1.4 is shown in Figure 2.3.*

$$j = Account \bowtie_{Account.cust=Overdraft.cust} Overdraft$$

$$q_{ex} = \Pi_{Account.cust,Account.typ,Account.bal}(\sigma_{Overdaft.bal<0}(j))$$

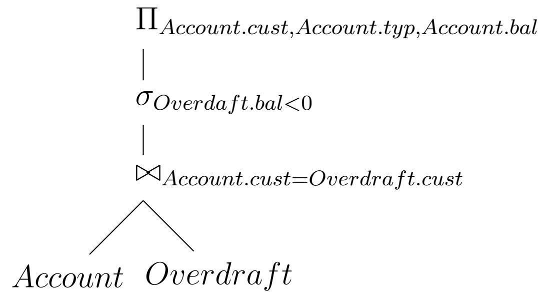


Figure 2.2. Example Algebra Tree

2.3 Updates and History

An update operation u presents one of update U (modifies one or more tuples), insert I (adds one or more new tuples), and delete D statements (removes one or more tuples) that changes a relation R. The main focus of our approach are update statements.

cust	typ	bal
Alice	Checking	-1100
Alice	Savings	1100

Figure 2.3. Example Query Result

$$\mathbf{e} := \mathbf{v}|\mathbf{c}|\mathbf{e}\{+,-,\times,\div\}\mathbf{e}|\text{if } (\boldsymbol{\phi}) \text{ then } \mathbf{e} \text{ else } \mathbf{e}$$

$$\boldsymbol{\phi} := \mathbf{e}\{=,\neq,<,\leq,>,\geq\}\mathbf{e}|\boldsymbol{\phi}\{\wedge,\vee\}\boldsymbol{\phi}|\mathbf{e} \text{ is null}|\neg\boldsymbol{\phi}|\text{true}|\text{false}$$

Figure 2.4. Grammar defining the syntax of expressions $\mathbf{e}$ and conditional expressions $\boldsymbol{\phi}$

Definition 5. $u(t) = t'$ *denotes* u *modifies the tuple* t *and creates the new tuple* t' *in the relation* R*. An update statement has a modification function Set and a condition* θ*. Set assigns an expression* e *over variables* v *and constants* c *to an attribute* A_i *of a relation* R *with* n *attributes.* $Set = (A_1 \leftarrow e_1, ..., A_n \leftarrow e_n)$*. A condition* θ *is a Boolean expression which combines atomic conditions using boolean operations. Each atomic condition is a comparison between two scalar expressions containing variables and constants. The grammar for generating Set and* θ *functions are shown in Figure 2.4.*

We write $Set(t)$ to denote the tuple produced by evaluating the expressions from Set over input tuples t. For example, for a relation $R(A, B, C)$, tuple $t = (1, 1, 1)$, and $Set = (A, A + B, 20)$ we get $Set(t) = (1, 2, 20)$. The result of applying update u to a relation R is defined as follows:

$$u(R) = \{Set(t) \mid t \in R \wedge \theta(t)\} \cup \{t \mid t \in R \wedge \neg\theta(t)\} \quad (2.1)$$

That is, the result contains updated versions of tuples t from R for which the condition of update evaluates to true (LHS of the union) and all tuples from R for which the condition evaluates to false (RHS of the union).

$$\mathbf{e} + \mathbf{e}' = \mathbf{e}' + \mathbf{e} \qquad \mathbf{e} \times \mathbf{e}' = \mathbf{e}' \times \mathbf{e} \qquad \text{(commutativity)}$$

$$\mathbf{e} + (\mathbf{e}' + \mathbf{e}'') = (\mathbf{e} + \mathbf{e}') + \mathbf{e}''$$
$$\mathbf{e} \cdot (\mathbf{e}' \cdot \mathbf{e}'') = (\mathbf{e} \cdot \mathbf{e}') \cdot \mathbf{e}'' \qquad \text{(associtivity)}$$

Figure 2.5. Equivalence expression **e** rules

Definition 6. *A transaction $T = \{u_1, \ldots, u_n, c\}$ is a sequence of update operations followed by a commit operation (c). The execution order of operations within a transaction is given by the order of these operations in the sequence. That is, if $i < j$ then u_i is executed before u_j.*

Definition 7. *A history $H = T_1, \ldots, T_n \ , \leq_H$ over a database D is a set of transactions over D with at most one update operation. Each T_i is a transaction and $\leq_H$ is a total order over the operations of H's transactions that complies with the order of operations within each transaction. We use $H(D)$ to denote the result of evaluating the history H over a database instance D.*

2.4 The Semiring-Annotation Framework

We extend semiring annotation to support capturing provenance of update operations and concurrent transactions in addition to queries. Green et al. [41, 50] have introduced the semiring annotation framework.

Definition 8. *A $\mathcal{K}$-relation R are annotated with elements from a commutative semiring $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$.*

A $\mathcal{K}$-relation R is a (complete) function that maps tuples to elements from $\mathcal{K} = (K, +_{\mathcal{K}}, \times_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ with the convention that tuples mapped to $0_{\mathcal{K}}$, the 0 element of

the semiring, are not in the relation. $\mathcal{K}$-relations generalize extensions of the relational model including bag semantics, incomplete databases, and various provenance models (e.g., Lineage).

Definition 9. *Let $\mathbb{D}$ be a universal domain of values and $\mathcal{K}$ a semiring. An n-nary $\mathcal{K}$-relation R is a function: $\mathbb{D}^n \rightarrow K$ that maps each tuple $t \in \mathbb{D}^n$ to an annotation from K. We require that R has finite support (number of tuples not mapped to 0). A $\mathcal{K}$-database is a set of $\mathcal{K}$-relations.*

A structure $\mathcal{K}$ is a commutative semiring if it fulfills the equational laws shown on the top of Figure 2.8. 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. Intuitively, the $+_{\mathcal{K}}$ and $\times_{\mathcal{K}}$ operations of the semiring correspond to alternative and conjunctive use of tuples. For instance, if an output tuple t was produced by joining input tuples annotated with k and k', then the tuple t would be annotated with $k \times_{\mathcal{K}} k'$.

Definition 10. *Provenance semirings are polynomials with integer coefficients. Positive algebra semantics for any commutative semirings factors through the provenance semantics.*

Provenance polynomials (semiring $\mathbb{N}[X]$), polynomials over a set of variables X which represent tuples in the database, are the most general form of semiring annotation. Every tuple in an instance is annotated with a unique variable $x \in X$ [41, 50]. This semiring $\mathbb{N}[X]$ has the important property that for any semiring $\mathcal{K}$ the annotation of a query result t in $\mathcal{K}$ can be derived from the provenance polynomial for t. This is done by mapping each variable $x \in X$ to an element from $\mathcal{K}$ and interpreting the abstract $+$ and $\times$ operations in $\mathbb{N}[X]$ as the corresponding operations in $\mathcal{K}$.

Definition 11. *Any valuation $\chi : X \rightarrow K$ of variables to elements from a semiring $\mathcal{K}$ can be lifted to a semiring homomorphism $Eval_{\chi} : \mathbb{N}[X] \rightarrow \mathcal{K}$.*

Semiring homomorphisms commute with queries. This means that any type of semiring annotation can be computed from the $\mathbb{N}[X]$ annotation of a query result. We extend Semiring homomorphisms to commute with update operations. We present detail in Chapter 4. The Figure 2.6 shows some semirings and the extensions of the relational model they encode.

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

Figure 2.6. Semirings and Their Corresponding Models

The semiring $\mathbb{B}$ with elements *true* and *false* using $\vee$ as addition and $\wedge$ as multiplication corresponds to set semantics (Definition 2). The semiring $\mathbb{N}$, the set of natural numbers with standard arithmetics corresponds to bag semantics (Definition 3). In the Lineage provenance model, the provenance of a result tuple t of a query is a set of tuples from the input that were used to derive t. The semiring over the powerset of tuples in an instance (represented as variables X) using set union for addition and multiplication corresponds to Lineage [25].[1]

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

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

Figure 2.7. Relation R and the result of query Q

Example 4. *Consider the* $\mathbb{N}[X]$*-relation* R *shown in Figure 2.7 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 a* $\mathbb{B}$*-annotation* true *indicating that this result tuple exists 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 under bag semantics. Finally, by mapping* x_1 *to* $\{x_1\}$ *and* x_2 *to* $\{x_2\}$*, and by interpreting the expression in the lineage semiring we get* $\{x_1, x_2\}$*, the Lineage of the first result.*

2.5 Concurrency Control Protocol

Our approach supports capturing provenance for concurrent transactions that are executed under two common concurrency control protocols such as snapshot isolation (*SI*) and read committed snapshot isolation (*RC-SI*. The term transaction refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account. It is important that either all actions of a transaction be executed and reflected completely, or none of them in case of some failure and partial effects of each incomplete transaction must be undone. This property

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

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

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

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

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

Figure 2.8. Equivalence relations for $\mathcal{K}^{\nu}$

is called *atomicity*. A transaction must preserve database *consistency*. If a transaction is run atomically in isolation starting from a consistent database, at the end of the transaction the database must be consistent again. Execution of a transaction in isolation preserves the consistency of the database. In a database system where several concurrent transactions are executing, if shared data and their updates are not controlled there is a possibility that transactions see inconsistent intermediate states created by other concurrent transactions. So, database systems must provide mechanisms to isolate transactions from the effects of other concurrent transactions. This property is called *isolation*. Further, once a transaction is successfully executed, its effects must persist in the database. Even with a system failure, the database should not ignore a transaction that successfully committed. This property is called *durability*. These properties are called the *ACID properties* that is derived from the first letter of each of these four properties [66]. Because of the above four properties, transactions are an ideal way of structuring interaction with a database. This leads us to impose a requirement on transactions themselves. When several transactions execute concurrently in the database, however, the isolation property may no longer

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

Figure 2.9. Example Transactional History

be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through a so-called concurrency control protocols. There are a variety of concurrency control protocols. We focus on transactions executed using the *snapshot isolation* (*SI*) concurrency control protocol and the *read committed* variant of this protocol (*RC-SI*). Compared to SI, RC-SI improves timeliness and performance at the cost of weaker consistency.

Example 5. *Consider the example database shown in Figure 2.10 storing information about employees and the bonuses they received. Two transactions have been executed concurrently (Figure 2.9). In this example, all software engineers got a bonus of $1000 while software architects received $2000. Suppose administrator Bob executed transaction T_7 to update the position of Mark Smith to reflect his recent promotion to architect and update his bonus accordingly (increasing it by $1000). Concurrently, user Alice executed Transaction T_8 to implement the company's new policy of giving an additional bonus of $500 to all* software engineers. *The result of executing the transactional history under different concurrency control protocols would be different that is discussed in the following examples.*

Snapshot Isolation. Under *snapshot isolation* (*SI*) [13] 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. Snapshot isolation prevents many of concurrency anomalies but it is not serializable [34]. Using SI, reads never block concurrent reads or writes, because each transaction sees a consistent database version as of its start. To support snapshots, old tuple versions cannot be deleted until all transactions that may need them have finished. Typically, this is implemented by storing multiple timestamped versions of each tuple and assigning a timestamp to every transaction when it begins that determines which version of the database it will see (its snapshot). Concurrent writes are allowed under SI. However, if several concurrent transactions write the same data item d, only one will be allowed to commit. Under the *First Committer Wins (FCW)* rule, the transaction which tries to commit first is allowed to commit. Under the *First Updater Wins (FUW)* rule, the first transaction updating d is allowed to commit. SI corresponds to isolation level `SERIALIZE` in systems such as Oracle and older versions of PostgreSQL. These implementations neither apply the *FCW* nor the *FUW* rule, but instead use write locks that are held until transaction commit. A transaction T waiting for a lock is aborted if the transaction T' holding the lock commits (and continues if T' aborts).

Example 6. *As an extension of Example 5, suppose the transactional history shown in Figure 2.9 was executed under the snapshot isolation concurrency control protocol. The Transaction T_7 and T_8 see the private snapshot of the database as they started (Figure 2.10) and they do not see any changes by the other concurrent transaction. In particular, Transaction T_8 did not see the uncommitted change of T_7 applying Mark's promotion and it updated Mark's bonus as Mark was still a software engineer. Meanwhile, Transaction T_7 tried to update Mark's bonus by adding additional $1000 bonus. Both of transactions attempted to update the same row so after Transaction T_8 committed, Transaction T_7 returned a write-conflict error based on the first-updater-*

wins version of snapshot isolation.

Read Committed Snapshot Isolation. Database systems implementing SI as a concurrency control protocol typically also support 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 transactions 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.

Example 7. *As an extension of Example 5, consider the transactional history shown in Figure 2.9 was executed under the read committed snapshot isolation. Figure 2.10 shows relations* `Employee` *and* `Bonus` *before the execution of these transactions. All new and updated tuples for these relations after the execution of these transactions are shown in Figure 2.11 (updated attributes are marked in red). Bob has executed a query at the end of his transaction (*T_7*) to double check the bonus amount for Mark expecting a bonus of $2000 instead of the actual result (the bonus of $2500). The unexpected $2500 bonus is produced because Transaction* T_8 *did not see the uncommitted change of* T_7 *reflecting Mark's promotion. Thus, Mark was considered to still be a software engineer and received the additional $500 bonus. After Transaction* T_8 *committed, the update statement in Transaction* T_7 *which implementing adding additional $1000 sees the committed change of* T_8 *(the new Mark's bonus of $1500) so it changes Mark's bonus to $2500. This kind of error is hard to debug, because it only materializes if*

the execution of the two transactions is interleaved in a certain way and would not occur in any serializable schedule.

2.6 Summary

In this chapter, we discussed some necessary background of our approach for computing provenance of update operations and concurrent transactions. We presented an overview of the semiring model as we extend this provenance model for update operations and transactions. One of our main contributions is computing provenance of transactions that were executed concurrently under SI or RC-SI concurrency control protocols. We compared and presented an example for these common concurrency control protocols.

Employee

ID	Name	Position	
101	Mark Smith	Software Engineer	e_1
102	Susan Sommers	Software Architect	e_2
103	David Spears	Test Assurance	e_3

Bonus

ID	EmpID	Amount	
1	101	1000	b_1
2	102	2000	b_2
3	103	500	b_3

Figure 2.10. Running example database instance

Employee

ID	Name	Position	
101	Mark Smith	Software Architect	e_1'
102	Susan Sommers	Software Architect	e_2
103	David Spears	Test Assurance	e_3

Bonus

ID	EmpID	Amount	
1	101	2500	b_1''
2	102	2000	b_2
3	103	500	b_3

Figure 2.11. New and modified tuples after execution of the example history under RC-SI (version 26).

CHAPTER 3
RELATED WORK

We propose a provenance model that supports capturing provenance of update operations and concurrent transactions in addition to queries. In this chapter, an overview of research on provenance models, provenance systems, and provenance for updates are presented. We also discuss traditional what-if queries as we introduce a new type of analytical queries, historical what-if queries.

3.1 Provenance Models

Provenance of relational queries has been studied extensively in the recent years leading to the development of several models including Why-provenance [20, 28], Where-provenance [20], Lineage [28], Why-not [23, 16, 15], How [41]. See [25, 45] for overviews. Buneman et al. [20] 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 [25] for a relational version. Green et al. [41] has addressed how data were manipulated by the query to produce the result. Chapman et al.[23] has presented why-not for provenance of missing results. Cui et al. [28] 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. See [25, 45] for overviews.

The seminal paper from Green et al. [41, 42] introduced the $\mathcal{K}$-relational model, an extension of the relational model with annotations from a commutative semiring and has shown how such annotations propagate through positive relational algebra ($\mathcal{RA}^+$) queries. The semiring of provenance polynomials is the most general form

of annotation in this model. Provenance polynomials generalize the relational datamodel (set and bag semantics), several extensions (e.g., trust), and less informative provenance models including Lineage and Why-provenance. See [50] for an overview of this model and its extensions beyond positive relational algebra (e.g., set difference [37] and aggregation [4]). Kostylev et al. [56] have studied data annotated with annotations from multiple semirings. Buneman et al. [21] 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. [62] discuss factorization of provenance polynomials and Amsterdamer et al. [3] rewrite queries into equivalent queries (under set semantics) with minimal provenance. Boolean Circuits can be used to compactly represent semiring expressions [31]. It has been proven that provenance polynomials can be extracted from the PI-CS [40] and Provenance Games [55] models. The latter also addresses negation. Negation may lead to large provenance games. This problem has been partially addressed by an extension of this model which encodes sets of tuples using constraints [64]. Provenance for missing answers is achieved by using a provenance model for queries with negation in [72].

The idea of annotating parts of a provenance polynomial with functional symbols was, to the best of my knowledge, first applied in the context of the Orchestra system to record applications of schema mappings [49]. The version annotations in my model were inspired by this idea. The major advances I made in developing my extension are 1) encode derivation under concurrent transactions and 2) model the visibility rules of the snapshot isolation and read committed snapshot isolation concurrency control protocols. The proposed model is a strict generalization of the semiring model in the sense that the semiring annotations of a tuple can be derived

from this model.

3.2 Provenance Systems

Systems such as Trio [1], DBNotes [14], Mondrian [36], MMS [68], BDMBS [33], Orchestra [51], Propolis [32], LogicBlox [50], Perm [40] encode provenance annotations as standard relations and use query rewrite techniques to propagate these annotations during query processing. ProvSQL [65] also applies query rewriting to capture the provenance of query results and it uses a separate column for their tables which contains provenance. Smoke [63] integrate the provenance capture logic into physical database operators. I implement provenance computation for transactions by propagating a relational encoding of provenance annotations. Similar to the Perm system, I refrain from eagerly computing provenance for all operations, but instead reconstruct provenance when requested. These systems mostly focus on queries or update operations without considering transactions and the concurrency control protocols that they were executed under it. Our approach captures provenance for transactional updates and it supports two commonly used concurrency control protocols such as snapshot isolation and read committed snapshot isolation.

Many provenance systems and algorithms propose solutions to reduce provenance capture runtime overhead [44, 2, 15, 16, 57, 67]. One important factor that effect the runtime of provenance capture is that whether their systems capture provenance eagerly or they compute provenance whenever they are requested in a lazy way [25]. Lazy provenance capture has a less runtime overhead during normal processing time of a system but it requires extra time for computing provenance on demand. In contrast, capturing provenance eagerly adds a higher runtime overhead while it improves capturing provenance runtime. Our approach compute provenance of transactional updates in a lazy way based on user demands and it avoids overhead of eagerly capturing provenance during each transaction execution. Figure 3.1 sum-

marizes the comparison of provenance systems and their provenance computational methods.

System	Computational Method	Query Support	Update Support	Transaction Support
Trio	eager, propagation	YES	NO	NO
DBNotes	eager, propagation	YES	NO	NO
Mondrian	eager, propagation	YES	NO	NO
MMS	lazy, propagation	YES	NO	NO
BDBMS	eager, propagation	YES	NO	NO
ORCHESTRA	eager, propagation	YES	YES	NO
Propolis	eager, propagation	YES	NO	NO
LogicBlox	eager, propagation	YES	NO	NO
Perm	lazy, propagation	YES	NO	NO
ProvSQL	eager, query rewriting	YES	NO	NO
Smoke	eager, instrument physical operator	YES	NO	NO

Figure 3.1. Comparison of the functionality provided by provenance systems

3.3 Provenance for Updates and Past Operations

Provenance for updates has been studied in related work [51, 19, 69], but none of these approaches addresses the complications that arise when updates are run as parts of concurrent transactions. Note that the "transactions" from Archer et al. [11] are sequences of updates and not concurrent transactions. Buneman et al. [19] have studied a copy-based provenance type for the nested update language and nested relational calculus. Vansummeren et al. [69] define provenance for SQL DML statements by modifying the updates to store provenance. My approach differs in that provenance is reconstructed on demand instead of computing and storing provenance for all operations. Zhang et al. [74] demonstrated that an audit log and time travel functionality is sufficient for computing the provenance of past queries. In this work,

I prove that audit logging and time travel are also sufficient for computing the provenance of transactions. This idea of using a log of operations (and changes to data) to reconstruct provenance by replaying operations has also been applied in the DistTape system [75] (distributed datalog) and the Ariadne system [39] (stream processing). Our approach simulates the updates instead of replaying them. Thus, there is no need to pay the overhead of DB update operations (caused by logging, concurrency control, and I/O of writing changes to disk) and optimizations can be applied such as reordering updates and pushing selections through updates that are not available to a DBMS if the system replays updates one at a time. Extending approaches for updates to support transactions is non-trivial, because it requires tracking provenance through multiple operations taking the visibility of tuple versions into account (some of which only exist temporarily during the execution of a transaction). I proposed the first solution to compute transactional provenance using a novel technique for query-based replay.

3.4 What-if Queries

In this section, we overview what-if analytic methods. What-if queries [5, 48] determine the effect of a hypothetical change to an input database on the results of query. What-if queries [12, 30, 76, 17] are often realized using incremental view maintenance to avoid having to reevaluate the query over the full input including the hypothetical changes. The draw back of traditional what-if queries is that it is usually ambiguous how a change to the database can be achieved primitively.

Tiresias [59] presents how-to queries which determines how to translate a change to a query result into modifications of the input data. The QFix system [70] is essentially a variation on this where the change to the output has to be achieved by a change to a query (update) workload. Changes to a query are limited to modifying existing variables and constants of that query. Similarly, our approach is based

on update operations but instead of fixing or defining an update operation based on desired result, the hypothetical modified operations is defined by the user and our approach predicts the effect of such changes on the query result. We propose a novel type of predictive queries, historical what-if queries to evaluate the effect of a hypothetical change to a past database operations and transactions. This type of analytical queries are more flexible as users can modify update operations themselves which is not limited to changing existing variables and constants (e.g. adding new variables) and investigate the effect of their changes.

3.5 Summary

We discussed and compared provenance models, and provenance systems. Most of these researches do not support update operations and none of them are designed to compute provenance of concurrent transactions. We also discussed traditional what-if queries. Our historical what-if query is a new type of analytical queries which is different from traditional what-if query and we present detail in Chapter 8.

CHAPTER 4
THE MV-SEMIRING MODEL

We now formally introduce our MV-semiring model that extends $\mathcal{K}$-relations with support for updates and transactions. The proposed model is adopted to different concurrency control protocols such as snapshot isolation and read committed snapshot isolation. We need a provenance model which is powerful enough to provide a full account of how tuple versions have been derived from other tuple versions and through updates or concurrent transactions in a history. It must support provenance tracking for queries, i.e., the provenance a query result can not just be traced back to the inputs of the query, but also reaches back into the transactional history that produced these inputs.

4.1 MV-semirings

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

It is tempting to extend such a representation of snapshots with semiring

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

Example 8. *As an example of the first problem consider how the instance of Figure 4.4 would be represented using $\mathcal{K}$-relations and a typical implementation of snapshot isolation using three additional attributes: the identifier of the transaction that created the tuple version (XID), the version when the tuple started to be valid (T_b), and the version when this tuple version is no longer valid (T_e). We show this instance below. Attribute T_b of tuple t is set to the commit time of the transaction that produced tuple t. Using this technique to store snapshot relations, the tuple versions visible to an update within a transaction T include all tuples committed before T started that were still valid when T started ($T_b \leq Start(T) < T_e$) plus all of T's own changes ($XID = T$).*

Account

	cust	**typ**	**bal**	XID	T_b	T_e
x_1	*Alice*	*Checking*	*-1100*	T_6	*15*	*UC*
x_2	*Alice*	*Savings*	*1100*	T_5	*13*	*UC*
x_3	*Peter*	*Savings*	*5390*	T_5	*13*	*UC*

Overdraft

	cust	**bal**	XID	T_b	T_e
$x_1 \cdot x_2$	*Alice*	*-100*	T_6	*15*	*UC*

The first difference of this representation to the instance annotated using our

model is that the annotations of the tuples in relation `Overdraft` *do not contain the whole provenance of such a tuple - part of its provenance is stored in a tuple from the* `Account` *relation. Thus, reconstructing the complete derivation history of a tuple requires correlating provenance across multiple tuples.*

We have lost information of how tuples have been derived, e.g., although we can infer that the two tuples annotated with x_1 and $x_1 \cdot x_2$ are somewhat related, we do not know how. All we know is that they were both produced by Transaction T_6 and started to be valid at time 15 (the time when T_6 committed). Extending the model by adding additional attributes such as tuple identifiers and identifiers for the update operation creating a tuple would solve this problem for this particular example. However, this not true in the general case. Consider a relation $R(A, B, C) : \{(1, 2, 3) \rightarrow x\}$ (here we denote a tuple t annotated with k as $t \rightarrow k$) and an insert `INSERT INTO S (SELECT A,C FROM R UNION SELECT B,C FROM R)`*. This creates the following instance $S(A, C) : \{(1, 3) \rightarrow x, (2, 3) \rightarrow x\}$. The same transaction then executes an insert* `INSERT INTO T (SELECT C FROM S WHERE f(A))` *where function f's return type is boolean. Let us assume that $f(1) = true$ and $f(2) = false$. The new tuple $t_{new} = (3)$ inserted into table T will be annotated with x. Based on this annotation it is impossible to know whether this tuple was derived from tuple $(1, 3)$ or $(2, 3)$. Additional information that we can extract from the temporal attributes of the snapshot isolation implementation is not useful for resolving this ambiguity.*

These examples illustrate the need for a provenance model that can help us track the origin of tuple versions. We have developed an extension of the semiring model that fulfills this requirement. Given a semiring $\mathcal{K}$ we construct a new semiring $\mathcal{K}^\nu$ that represents $\mathcal{K}$ with embedded history. We call structures constructed in this fashion multi-version (MV) semirings. The elements of such a semiring are symbolic expressions over elements from K, version annotations, and semiring operations where

the structure of an expression encodes the derivation history of a tuple.

MV-semirings are a specific class of semirings that encode the derivation of tuples based on a history of transactional updates. For each semiring $\mathcal{K}$, there exists a corresponding semiring $\mathcal{K}^{\nu}$, e.g., $\mathbb{N}[X]^{\nu}$ is the MV-semiring corresponding to the provenance polynomials semiring $\mathbb{N}[X]$. Since $\mathbb{N}$ encodes bag semantic relations, $\mathbb{N}^{\nu}$ represents bag semantics with embedded history. Intuitively, $\mathcal{K}^{\nu}$ keeps track of what updates where applied to derive a tuple annotated with $\mathcal{K}$. Figure B.1 shows examples of $\mathbb{N}[X]^{\nu}$ annotations on the left of tuples. In these symbolic expressions variables (e.g., x_1, x_2, ...) represent identifiers of freshly inserted tuples and uninterpreted function symbols (we call *version annotations*) encode which operations (e.g., an update) were applied to the tuple. The nesting of version annotations records the sequence of operations that created a tuple version.

4.2 Version Annotations

A version annotation $X^{id}_{T,\nu}(k)$ denotes that an operation of type X (update U, insert I, delete D, or commit C) that was executed at time $\nu - 1$ by transaction T affected a previous version of a tuple with identifier id and previous provenance k. Assuming domains of tuple identifiers $\mathbb{I}$, version identifiers $\mathbb{V}$, and transaction identifiers $\mathbb{T}$, we use $\mathbb{A}$ to denote the set of all version annotations:

$$I^{id}_{T,\nu}, U^{id}_{T,\nu}, D^{id}_{T,\nu}, C^{id}_{T,\nu} \quad \text{for} \quad id \in \mathbb{I}, \nu \in \mathbb{V}, T \in \mathbb{T} \tag{4.1}$$

Example 9. *Recall Example 16, Figure 4.2 shows the example database storing information about banking accounts and overdrafts with version annotations. Suppose Transactions T_5 and T_6 shown in Figure 4.1 were executed under SI. The states of the* `Account` *and* `Overdraft` *relations with version annotations after the execution of Transaction T_5 are shown in Figure 4.3 and after the execution of both transactions*

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

Figure 4.1. Example audit log for a SI transactional history

Account

	cust	typ	bal	
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400	a_1
$C^2_{T_1,4}(I^2_{T_1,3}(x_2))$	Alice	Savings	1000	a_2
$C^3_{T_2,3}(I^3_{T_2,1}(x_3))$	Peter	Savings	4990	a_3

Overdraft

cust	bal

Figure 4.2. Database before execution of T_5 and T_6

in Figure 4.4. Consider the $\mathbb{N}[X]^\nu$*-relation* ***Account*** *in Figure 4.2. The second tuple is annotated with* $C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$*, i.e., it was created by an update of Transaction* T_5*, which updated a tuple that was inserted by* T_1*. Based on the outermost commit annotation, this tuple version is visible to transactions starting after version* 13.

Example 10. *Recall Example 5, Figure 4.6 shows the example database storing information about employees and their bonuses with version annotations. Suppose Transactions* T_7 *and* T_8 *shown in Figure 4.5 were executed under RC-SI. The states of the* ***Employee*** *and* ***Bonus*** *relations with version annotations after the execution of both transactions in Figure 4.7. Consider the* $\mathbb{N}[X]^\nu$*-relation* ***Employee*** *in Figure 4.7. The first tuple is annotated with* $C^1_{T_7,26}(U^1_{T_7,21}(C^1_{T_0,6}(I^1_{T_0,2}(x_1))))$*, it was modified by an update of Transaction* T_7*, which was inserted by* T_0 *and this tuple version is visible to*

Account

	cust	typ	bal	
$C^1_{T_1,4}(I^1_{T_1,2}(x_1))$	Alice	Checking	400	a_1
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100	a'_2
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390	a''_3

Figure 4.3. Database after execution of T_5

Account

	cust	typ	bal	
$C^1_{T_6,16}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1))))$	Alice	Checking	-1100	a'_1
$C^2_{T_5,14}(U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))))$	Alice	Savings	1100	a'_2
$C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$	Peter	Savings	5390	a''_3

Overdraft

	cust	bal	
$C^4_{T_6,16}(I^4_{T_6,15}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)) \cdot C^2_{T_1,4}(I^2_{T_1,3}(x_2)))))$	Alice	-100	o_1

Figure 4.4. Database after execution of T_6

transactions starting after version 25.

4.3 MV-semiring Annotation Domain

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

$$P := k \mid P + P \mid P \cdot P \mid \mathcal{A}(P) \tag{4.2}$$

Note that $+$ and $\cdot$ in these expressions are used to encode that a tuple depends on multiple input tuples, e.g., a query such as the one used by the insert of example Transaction T_6 or an update that modifies two tuples that are distinct in the input

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

Figure 4.5. Example audit log for a RC-SI transactional history

to be the same in the output (e.g., `UPDATE Account SET typ = 'Savings'`). For example, consider a query $\Pi_{typ}(Account)$ evaluated over the instance from Figure 4.2. The result tuple `(Savings)` is derived from the second and third tuple in the `Account` table (the two tuples with this value in attribute `typ`) and, thus, would be annotated with $C^2_{T_1,4}(I^2_{T_1,3}(x_2)) + C^3_{T_2,3}(I^3_{T_2,1}(x_3))$. We would expect certain symbolic expressions produced by the grammar above to be equivalent, e.g., expressions in the embedded semiring $\mathcal{K}$ can be evaluated using the operations of the semiring ($k_1 + k_2 = k_1 +_{\mathcal{K}} k_2$) and updating a non-existing tuple does not lead to an existing tuple ($\mathcal{A}(0_{\mathcal{K}}) = 0_{\mathcal{K}}$). This is achieved by defining domain K^ν as the set of equivalence classes (denoted by $[]_\sim$) for expressions in P based on the equivalence relations shown in Figure 2.8.

Definition 12. *Let $\mathcal{K} = (K, +_{\mathcal{K}}, \cdot_{\mathcal{K}}, 0_{\mathcal{K}}, 1_{\mathcal{K}})$ be a commutative semiring. The MV-semiring $\mathcal{K}^\nu$ for $\mathcal{K}$ is the structure*

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

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

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

Theorem 1. *Let $\mathcal{K}$ be a semiring then $\mathcal{K}^\nu$ is a semiring.*

Addition and multiplication create a symbolic expression by connecting the inputs with $+$ or $\cdot$ and then output the equivalence class for this expression. For example, $k = U^1_{T,\nu}(3 \cdot 6)$ is a valid element of the bag semantics MV-semiring $\mathbb{N}^\nu$ that shows a tuple with identifier 1 was produced by an update (U) of transaction T at version ν. This element k is in the same equivalence class as $U^1_{T,\nu}(18)$ based on the equivalence that evaluates multiplication over elements from the embedded semiring $\mathcal{K}$.The intuitive meaning of the equivalences involving version annotations are: 1) update operations never create tuples from non-existing or deleted tuples (recall that if a tuple is annotated with $0_\mathcal{K}$ in relation R this denotes that the tuple is not in the relation R) and 2) alternative use of tuples distributes over updates (e.g., updating the result of a union query returns the same result as computing the union after updating its inputs). In the following we will omit the subscript of operations and neutral elements if the semiring is clear from the context or irrelevant to the discussion. Since we typically define a single semiring structure for a given set K, we will sometimes use $\mathcal{K}$ to refer both to the semiring $\mathcal{K}$ and its set K interchangeably.

There exists a strong connection between $\mathcal{K}$ and $\mathcal{K}^\nu$ relations: By evaluating the symbolic expression that make up an $\mathcal{K}^\nu$ element interpreting version annotations as functions $K \to K$, we transform an $\mathcal{K}^\nu$ relation into a corresponding $\mathcal{K}$ relation. Conceptually, this means we are removing the embedded history from the provenance. For example, if we apply this approach to derive provenance polynomials from their $\mathcal{K}^\nu$ counterpart, the result will record from which tuples a tuple was derived (and how), but no longer encode its update history. Below we define an operator Unv that implements this mapping based on a function $h_U : \mathcal{K}^\nu \to \mathcal{K}$.

Definition 13. *Let R be a $\mathcal{K}^\nu$-relation. The unversioning operation* $\textsc{Unv}(R)$*:* $\mathcal{K}^\nu$*-relation* $\to$ $\mathcal{K}$*-relation applies the mapping* $h_U : \mathcal{K}^\nu \to \mathcal{K}$ *defined below to every tuple's annotation, i.e.,* $\textsc{Unv}(R)(t) = h_U(R(t))$.

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

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

Example 11. *Reconsider the instance of relation* `Account` *shown in Figure 4.4. This instance is annotated with* $\mathbb{N}[X]^\nu$*, the MV version of the provenance polynomial semiring. The first tuple* a'_1 *is annotated with* $C^1_{T_6,16}(U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1))))$*, i.e., it was created by an update of Transaction* T_6*, that updated a tuple inserted by* T_1 *based on another previously inserted tuple by the same transaction. Based on the outermost commit annotation we know that this tuple version is visible to transactions starting after version* 15*. The second tuple* a''_3 *is annotated with* $C^3_{T_5,14}(U^3_{T_5,13}(U^3_{T_5,11}(C^3_{T_2,3}(I^3_{T_2,1}(x_3)))))$ *, i.e., this tuple was updated by a sequence of two updates by Transaction* T_5 *(and was originally produced by an insert by Transaction* T_2*). If we apply* $\textsc{Unv}$ *to relation* `Account`*, then* a'_1 *is annotated with* x_1 *and* a''_3 *is annotated with* x_3*.*

4.4 Normal Form and Admissible Instances

In the following we make use of a normal form for $\mathcal{K}^\nu$ elements that represents

them as a sum of subexpressions which use multiplication and version annotations. This will simplify the definition of updates and transactional semantics in our model.

Definition 14. *An $\mathcal{K}^\nu$ element k is normalized if it is of the form: $\sum_{i=0}^{m} k_i$ where 1) none of the summands k_i contains addition and 2) all summands are non-zero.*

Note that any annotation k can be translated into this normal form by applying the equational laws of MV-semirings. For example, an annotation $I^3_{T,\nu_2}(U^2_{T,\nu_1}(x_1) + U^1_{T,\nu_1}(x_2))$ can be normalized based on distributivity of addition over version annotations into $I^3_{T,\nu_2}(U^2_{T,\nu_1}(x_1)) + I^3_{T,\nu_2}(U^1_{T,\nu_1}(x_2))$. It will be helpful to introduce notation for accessing particular elements from the sum of a normalized $\mathcal{K}^\nu$ element. We use $n(k)$ to denote the number of summands of a normalized $\mathcal{K}^\nu$-element k and $k[i]$ to denote the ith element in the sum (assuming some order over the summands). $\mathcal{K}^\nu$ expressions admit a (non unique) normal form representing an element $k \in K^\nu$ as a sum $\sum_{i=0}^{n} k_i$ where none of the k_i contains any addition operations. Any $\mathcal{K}^\nu$ element can be brought into this normal form by applying the equivalences from Figure 2.8. Intuitively, each summand in the normal form corresponds to a tuple under bag semantics. Thus, we will sometimes refer to a summand as a tuple version in the following. Note that some expressions produced by the grammar in Equation (4.2) can not be produced by any transactional history. For instance, $U^1_{T,11}(C^1_{T,10}(\ldots))$ is invalid, because it implies that Transaction T executed an update after its commit. A $\mathcal{K}^\nu$ instance is **admissible** if it is the result of applying a transactional history (formally defined later) to an empty input database.

4.5 Properties of MV-semirings

We now discuss several properties of our model. A formal treatment including proofs is presented in Section A. An important property of provenance polynomials is that the result of a query Q in any semiring $\mathcal{K}$ can be computed from the $\mathbb{N}[X]$ result

of Q by replacing variables in polynomials with elements from $\mathcal{K}$ and evaluating the resulting expression in $\mathcal{K}$. This property was proven by Green et al. [41] by demonstrating 1) that the process described above is a semiring homomorphism, i.e., a mapping $h : \mathbb{N}[X] \to \mathcal{K}$ that agrees with semiring operations; and 2) that homomorphisms commute with queries. We demonstrate that any homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a "history-preserving" homomorphism $h^\nu : {\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$ by applying h to each $\mathcal{K}_1$ element in a ${\mathcal{K}_1}^\nu$ element k. Lifted homomorphisms also commute with updates and transactional histories. Thus, $\mathbb{N}[X]^\nu$ enjoys the same generality property among MV-semirings as $\mathbb{N}[X]$ does for semirings. We demonstrate that $Q \equiv_{\mathbb{N}[X]^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}^\nu} Q'$ for any MV-semiring $\mathcal{K}^\nu$. Thus, the equivalence between histories and reenactment queries that we prove for $\mathbb{N}[X]^\nu$ in Chapter 5 implies equivalence for any $\mathcal{K}^\nu$. In particular, reenactment works for bag semantics (semiring $\mathbb{N}^\nu$).

4.6 Queries

We extend the standard definition of positive relational algebra ($\mathcal{RA}^+$) over $\mathcal{K}$-relations [41] with an operator $\{t \to k\}$ that creates a singleton relation (a tuple t annotated with k). Let $t.A$ denote the projection of a tuple t on a list of projection expressions A and $t[R]$ to denote the projection of t on the attributes of relation R. For a condition θ and tuple t, $\theta(t)$ denotes a function that returns $1_\mathcal{K}$ if $t \models \theta$ and $0_\mathcal{K}$ otherwise.

Definition 15 (**Positive Relational Algebra over** $\mathcal{K}$-relations)**.** *Let R and S denote $\mathcal{K}$-relations, t, t' denote tuples, and $k \in K$. The operators of $\mathcal{RA}^+$ are defined as:*

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

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

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

Note that the singleton construction $\{t \to k\}$ introduced above does not affect the commutativity of semiring homomorphisms with queries. However, since this operator explicitly mentions a semiring element $k \in \mathcal{K}$, a homomorphism $h : \mathcal{K} \to \mathcal{K}'$ has to be applied to the query too to guarantee that it returns a $\mathcal{K}'$ relation (this is similar to the treatment of the constant annotation operator in [37]). Let $h(Q)$ denote the application of the homomorphism h to query Q, i.e., we replace every operator $\{t \to k\}$ in Q with $\{t \to h(k)\}$.

Theorem 2. *Let $h : \mathcal{K} \to \mathcal{K}'$ be a semiring homomorphism, then h commutes with any $\mathcal{RA}^+$ query Q if h is applied to Q. That is for any $\mathcal{K}$ database instance I. Then, we have $h(Q)(h(I)) = h(Q(I))$*

Proof. The proof presented in Chapter A. □

The mapping $h_U : \mathcal{K}^\nu \to \mathcal{K}$ used in the definition of the Unv operator introduced in Section 4.3 is a semiring homomorphism. Thus, the application of Unv commutes with queries. Practically, this means we can execute queries over relations with embedded history and then derive the corresponding relation without history or equivalently strip the history information upfront.

Theorem 3. *h_U is a surjective semiring homomorphism.*

Consider $\mathbb{N}[X]^\nu$, i.e., the MV-semiring version of the provenance polynomials semiring $\mathbb{N}[X]$. A variation of the fundamental property of the semiring framework

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

Theorem 4. *Any semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism $h^\nu : {\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$ as defined below. If h is surjective then so is h^ν.*

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

Proof. The proof presented in Chapter A. □

Example 12. *Consider a query $Q = \Pi_{cust}(Account)$ run over the instance from Figure 4.2. This query returns two single tuples c_1 =(Alice) and c_2 =(Peter) as shown below. The annotation of these tuples record that c_1 was produced from two tuples in the input of the query and how these two tuples were created (e.g., $C^1_{T_1,4}(I^1_{T_1,2}(x_1))$). c_2 was produced from a single tuple ($C^3_{T_2,3}(I^3_{T_2,1}(x_3))$) in the input of the query. To compute the answer to this query under bag semantics we first apply the* UNV *operator which returns annotation x_1+x_2 for tuple c_1 and then apply an assignment $\mathbb{N}[X] \to \mathbb{N}$. If we assume that both input tuples have multiplicity 1, then tuple c_1 will be annotated*

with $1+1=2$*, the multiplicity of this query result under bag semantics. For tuple* c_2 *after applying the* UNV *operator, it returns annotation* x_3*, then after applying an assignment* $\mathbb{N}[X] \to \mathbb{N}$*, it will be annotated with* 1 *under bag semantics.*

	cust	
$C^1_{T_1,4}(I^1_{T_1,2}(x_1)) + C^2_{T_1,4}(I^2_{T_1,3}(x_2))$	*Alice*	c_1
$C^3_{T_2,3}(I^3_{T_2,1}(x_3))$	*Peter*	c_2

4.7 Update Operations

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

Definition 16. *Let* R *be a normalized, admissible* $\mathcal{K}^{\nu}$*-relation. Let* A *be a list of projection expressions with the same arity as* R *and* id_{new} *to denote a fresh id. Let* Q *be a query over a database* D *such that for every* $\{t \to k\}$ *operation in* Q *we have*

$k \in \mathcal{K}$. *We define updates over* $\mathcal{K}^\nu$*-relations as:*

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

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

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

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

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

As a convention, if an attribute a is not listed in the list of expressions A of an update then $a \rightarrow a$ is assumed. For instance, the first update of the example transaction T_5 in Figure 4.1 would be written as $\mathcal{U}[typ = \text{'}Savings\text{'}, bal + 100 \rightarrow bal, T_5, 10](Account)$. What tuple identifiers (id_{new}) are assigned by inserts to new tuples is irrelevant as long as identifiers are sufficient for uniquely identifying tuple versions. We use a Skolem function $f_{id}(T, \nu, t, k)$ to create identifiers which takes as input the transaction T, version ν, tuple t to be annotated, and $\mathcal{K}^\nu$-element k that is wrapped in the version annotation.

Example 13. *Consider the first update operation of Transaction* T_5 *from the running example in Figure 4.1. This update runs over the version of relation* `Account` *shown in Figure 4.2. This update operation can be expressed in our model as:* $\mathcal{U}[typ = \text{'}Savings\text{'}, bal + 100 \rightarrow bal, T_5, 10](Account)$. *Tuple* a_1*: Tuple* a_1 *in the instance of relation* `Account` *is annotated with* $C_{T_1,4}^1(I_{T_1,2}^1(x_1))$ *in the input, a single element sum.*

Since, this tuple does not fulfill the update's condition, the inner sum evaluates to $U^1_{T_5,11}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))) \times (typ = \text{'}Savings\text{'})(a_1) = U^1_{T_5,11}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))) \times 0 = 0$.
Tuple a_2: It is annotated with $C^2_{T_1,4}(I^2_{T_1,3}(x_2))$. This tuple fulfills the condition $typ =$ $typ = \text{'}Savings\text{'}$ of the update and, thus, the first expression $(R(t)\times(\neg\theta)(t))$ in the annotation created by the update evaluates to: $Account(a_2)\times(\neg(typ = \text{'}Savings\text{'}))(a_2) =$ $C^2_{T_1,4}(I^2_{T_1,3}(x_2)) \times 0 = 0$. The second part of the expression sums the annotations over all tuples u such that if the update is applied to them the resulting updated tuples are equal to a'_2. Since the update sets attribute balance, these are all tuples (Alice, Savings, b') for some balance b'. However, all tuples except for a_1, a_2, and a_3 are annotated with 0 in the input (they are not part of this instance). For tuples u with $Account(u) = 0$ the inner sum evaluates to $U^2_{T_5,11}(Account(u)) \times \theta(u) = U^2_{T_5,11}(0) \times \theta(u) = 0$. Intuitively, this is the expected result, because an update is only creating new versions of existing tuples. Finally, a_2 is the only tuple which fulfills the condition of the update and is not annotated with 0 in the input. For a_2 the inner sum evaluates to $U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2))) \times(typ = \text{'}Savings\text{'})(a_2) = U^2_{T_5,11}(C^2_{T_1,4}(I^2_{T_1,3}(x_2)))$. As expected the annotation denotes that the resulting tuple was derived from tuple a_2 in the previous version of relation `Account` and was not affected by any other input tuple.

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

Theorem 5. *Let h^ν be a lifted homomorphism as defined in Theorem 4. h^ν commutes*

with updates.

Proof. The proof presented in Chapter A. □

4.8 Transactions and Histories

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

Definition 17. *Let H be a history over a database D. The version $R[\nu]$ of relation $R \in D$ at time ν and the version $R[T, \nu]$ of relation R visible within transaction $T \in H$ at time ν are defined in Figure 4.8 and Figure 4.9 for SI and RC-SI, respectively.*

Snapshot Isolation Histories. A transaction T under SI sees 1) its own updates and 2) the updates of transactions that have committed before $Start(T)$. The first condition is encoded in the definition of $R[T, \nu]$ and the second one in the definition of $R[\nu]$.

Relation Versions Visible Inside an SI Transaction. $R[T, \nu]$ contains the result of applying the latest update of T before ν to the version valid before the update. As a convention, we define $R[T, \nu] = \emptyset$ if $\nu < Start(T)$. The 1st update in a transaction sees $R[Start(T)]$, the version of R containing all changes of transactions committed before T started (2nd case in Figure 4.8a). We explain how to compute $R[\nu]$ below. Consider a transaction $T = u_1, \ldots, u_n, c$ and assume for simplicity that every update is modifying the same relation R. The 2nd update u_2 within the transaction will see the version of R produced by applying update u_1 to $R[Start(T)]$, the 3rd update u_3 will run over the version of R that is the result of applying update u_2 to the result of u_1 and so on. This is encoded by the 3rd and 5th case in Figure 4.8a. If T executed an update on R at version $\nu - 1$ then $R[T, \nu]$ is the result of applying the update to $R[T, \nu - 1]$. If transaction T committed at $\nu - 1$ then we apply a commit operation to $R[T, \nu - 1]$ (4th case). If the transaction did not execute any operation at $\nu - 1$ (including the case where $\nu > End(T) + 1$) then $R[T, \nu]$ is the same as $R[T, \nu - 1]$ (5th case).

Relation Versions Containing Committed Changes. Under SI, a transaction starting at ν will see a version of relation R that contains all changes of transactions committed before ν. Recall that we use $R[\nu]$ to denote this version of R. Figure 4.8b to 4.8d show the definition of $R[\nu]$. $R[\nu]$ can be expressed as a union (sum) over all tuple versions (annotations) created by committed past transactions as long as we make sure that we are not including the same tuple version more than once. Furthermore, we should not include annotations that correspond to tuple versions which have been replaced with newer versions or were deleted. We enforce these two conditions using a predicate VALIDAT.

Determining Valid Tuple Versions. VALIDAT(T, t, k, ν) evaluates to 1 if two conditions are met: 1) annotation k was produced by transaction T, i.e., the outermost

version annotation in k is from T; 2) the tuple version corresponding to k was not updated (predicate $\textsc{updated}(T', t, k, \nu)$) by another transaction T' that committed before ν ($End(T') < \nu$).

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

Example 14. *Consider $Account[T_6, 11]$, the version of relation* `Account` *from our running Example 9 visible to Transaction T_6 at version 11. Since $Start(T_6) = 11$, this version is equal to $Account[11]$. We construct $Account[11]$ by combining tuple annotations created by transactions that committed before T_6 started as long as these tuple versions have not been invalidated by another already committed transaction. For instance, Transaction T_1 did create the annotation $k = C^1_{T_1,4}(I^1_{T_1,2}(x_1))$ on tuple a_1 = (Alice, Checking, 400) as shown in Figure 4.2.* $\textsc{validAt}$ *evaluates to 1 for this annotation of tuple a_1 if no transaction that committed before 11 has invalidated this version. Since there is no such transaction, we get $Account[11](a_1) = C^1_{T_1,4}(I^1_{T_1,2}(x_1))$.*

Read-Committed SI Histories. A transaction T under RC-SI sees 1) its own updates and 2) the updates of transactions that have committed before the update was run ($R_{ext}[T, \nu]$).

Relation Versions Visible Inside a RC-SI Transaction. For RC-SI, we also

apply the definition from Figure 4.8a. The only difference is the third case when an update was executed by transaction T at time $\nu - 1$ so its modifications are reflected in $R[T, \nu]$ (shown in Figure 4.9a). The update will see tuple versions are created by: 1) the transaction's own updates; 2) other concurrent transactions which committed before $\nu - 1$. We discuss how to compute this version of a relation R (denoted by $R_{ext}[T, \nu - 1]$) in the following.

Relation Version Visible to Updates. Figure 4.9b shows the definition of $R_{ext}[T, \nu]$, the version of relation R that is visible to an update of transaction T executed at time ν. This state of relation R contains all tuple versions created by committed transactions as long as they have not been overwritten by a previous update of transaction T (the first sum) and tuple versions created by previous updates of transaction T (the second sum). Here by overwritten we mean that a tuple version is no longer valid, because either it has been deleted or because it was updated and, thus, it has been replaced with a new updated version. Function validEx implements this check. It returns 1 if the tuple version has not been overwritten and 0 otherwise. This function uses a predicate updated(T, t, k, ν) which is true if transaction T has invalidated summand k in the annotation of tuple t before ν by either deleting or updating the corresponding tuple version. The second sum ranges over tuple versions $R[T, \nu]$ excluding tuple versions not created by transaction T (function validIn).

Relation Versions Containing Committed Changes. We use the same definition as for SI (Figure 4.8b and Figure 4.8c).

Example 15. *Assume T_5 and T_6 shown in Figure 4.4 were executed under RC-SI instead of SI. Consider Account$[T_6, 14]$, the version of* `Account` *visible to the insert of T_6 at time 14. The first update of Transaction T_6 did create the annotation $U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))$ on the tuple (Alice, Checking, -1100) of* `Account`*. Note that this is the version of the* `Account` *relation as seen by the insert of Transaction T_6.*

Based the our definition of tuple versions visible within a transaction T at time ν for RC-SI , we get:

$$Account_{ext}[T_6, 14](t) = \sum_{i=0}^{n(Account[14](t))} \Big(Account[14](t)[i] \times \textsc{validEx}(T_6, t, Account[14](t)[i], 14) \Big) + \sum_{i=0}^{n(Account[T_6,14](t))} \Big(Account[T_6, 14](t)[i] \times \textsc{validIn}(T_6, t, Account[T_6, 14](t)[i], 14) \Big)$$

This state of relation Account contains all tuple versions created by committed transactions as long as they have not been overwritten by a previous update of Transaction T_6 (the first sum) and tuple versions created by previous updates of transaction T_6 (the second sum). Here by overwritten we mean that a tuple version is no longer valid, because either it has been deleted or because it was updated and, thus, it has been replaced with a new updated version. Function VALIDEX *implements this check. It returns* 1 *if the tuple version has not been overwritten and* 0 *otherwise. Since* 1 *is the neutral element of multiplication and multiplication by* 0 *returns* 0 *in every semiring, this has the effect that the annotation is preserved if it is still valid (*VALIDEX *returns 1) or deleted if this tuple version is no longer valid (*VALIDEX *returns 0). Function* VALIDEX *uses a predicate* UPDATED(T, t, k, ν) *which is true if transaction T has invalidated summand k (a tuple version) in the annotation of tuple t before ν by either deleting or updating the corresponding tuple version. The second sum ranges over tuple versions in* $Account[T_6, 14]$ *excluding tuple versions not created by transaction T_6 (function* VALIDIN*).*

Consider the annotation for the first tuple of **`Account`** *at time 14:* $t_1 = (Alice, Checking, -1100)$. *The annotation of this tuple is* $k = U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))$. *This annotation records that this tuple version was created by the previous update of*

Transaction T_6. Applying the definition of VALIDIN*:*

$$\textsc{validIn}(T_6, t_1, k, 14) = \begin{cases} 1 & \textit{if}\ \exists \nu', k', id : k = X^{id}_{T_6,\nu'}(k') \wedge X \in \{U, D, I\} \\ 0 & \textit{otherwise} \end{cases}$$

This evaluates to 1 for the tuple under consideration, because ν'=12, $X = U$, and $k' = C^1_{T_1,4}(I^1_{T_1,2}(x_1))$. Similarly, applying the definition of VALIDEX *for this example we get:*

$$\textsc{validEx}(T_6, t_1, k, 14) = \begin{cases} 0 & \textit{if}\ \textsc{updated}(T_6, t_1, k, 14) \\ 1 & \textit{otherwise} \end{cases}$$

$$\textsc{updated}(T_6, t_1, k, 14) \Leftrightarrow \exists u \in T_6, t', i, j : \nu(u) < 14 \wedge \textit{Account}[T_6, \nu(u)](t_1)[i] = k$$
$$\wedge\ \textit{Account}[T_6, \nu(u) + 1](t')[j] = X^{id}_{T_6,\nu(u)+1}(k) \wedge X \in \{U, D\}$$

This evaluates to 0, because T_6 has updated that tuple. As a result, $\textit{Account}_{ext}[T_6, 14](t_1)$ contains tuple version $k = U^1_{T_6,12}(C^1_{T_1,4}(I^1_{T_1,2}(x_1)))$ since $validIn(T_6, t_1, k, 14) = 1$. Therefore, VALIDIN *returns 1 whereas* VALIDEX *returns 0 and $\textit{Account}_{ext}[T_6, 14]$ contains this version.*

Importantly, lifted homomorphisms also commute with transactional histories.

Theorem 6. *Let h^ν be a lifted homomorphism (Theorem 4). h^ν commutes with histories.*

Provenance Filtering. A tuple's annotation stores its derivation history since the origin of the database. This amount of information can be overwhelming to a user. As mentioned in Chapter 1, we can restrict provenance to tuple versions affected by a transaction. To restrict $R[T, End(T)]$, the provenance of a Transaction T for

relation R, we apply two filtering steps: 1) filter out tuples that were not affected by T. In this step, every summand is removed from the annotation of a tuple if it is not wrapped in a commit annotation of T, i.e., it is not of the form $C^{id}_{T,\nu}(k)$ 2) remove parts of the provenance that correspond to operations before the start of T. Each subexpression that is wrapped in the commit annotation of a transaction $T' \neq T$ is replaced with a variable disambiguated by tuple identifiers, i.e., every subexpression $C^{id}_{T',\nu}(k)$ is substituted with $C^{id}_{T',\nu}(x_{id})$. Assume we are interested in Transaction T and $T' \neq T$. An expression $C^{id}_{T',\nu'}(I^{id}_{T',\nu''}(I^{id_1}_{T',\nu_1}(x_1) \cdot I^{id_2}_{T',\nu_2}(x_2)))$ in the annotation of a tuple updated by T would be replaced with $C^{id}_{T',\nu'}(x_{id})$.

Definition 18. *Let T be a transaction in a history H over database D. The provenance $D[T]$ restricted to T is derived from $D[T, End(T)]$ by replacing each relation $R[T, End(T)]$ with $R[T]$ as defined below.*

$$R[T](t) = \sum_{i=0}^{n(R[T,End(T)](t))} filt(R[T, End(T)](t)[i]) \tag{4.3}$$

$$filt(k) = \begin{cases} C^{id}_{T,\nu}(h_f(k')) & \text{if } k = C^{id}_{T,\nu}(k') \\ 0 & else \end{cases} \tag{4.4}$$

$$h_f(k) = \begin{cases} k & \text{if } k \in \mathcal{K} \\ U/I/D^{id}_{T,\nu}(h_f(k')) & \text{if } k = U/I/D^{id}_{T,\nu}(k') \\ h_f(k_1) + h_f(k_2) & \text{if } k = k_1 + k_2 \\ h_f(k_1) \times h_f(k_2) & \text{if } k = k_1 \times k_2 \\ C^{id}_{T',\nu}(h_U(k')) & \text{if } k = C^{id}_{T',\nu}(k') \wedge \mathcal{K} \neq \mathbb{N}[X] \\ C^{id}_{T',\nu}(x_{id}) & \text{if } k = C^{id}_{T',\nu}(k') \wedge \mathcal{K} = \mathbb{N}[X] \end{cases} \tag{4.5}$$

[t] [t] [t] [t] [t] [t] [t] [t] [t] [t]

Debugging Example. We extend an example from our PVLDB demonstration [61] to justify the effectiveness of our approach. This example presents a post-mortem

debugger for transactions that is build on top of the reenactment functionality we have implemented in GProM. This example demonstrates the effectiveness of our model for provenance of transactions with respect to debugging, a common use case of provenance. The example shows how provenance capture using reenactment can help a developer to identify the cause of an error in a transaction execution and ultimately fix it. Here, we are not focusing on the GUI presented in our PVLDB demonstration, but rather on how our implementation of MV-semirings enables debugging.

Example 16 (Bob's Buggy Transaction)**.** *Bob is a developer at a bank that is using a database running the* snapshot isolation (SI) *concurrency control protocol. He is tasked with writing a transaction for withdrawing money from a customer's checking or savings account (a table* `account(cust,typ,bal)`*). If after the withdrawal the total balance of the checking and savings account for the customer are below 0, then an overdraft record should be inserted into a table* `overdraft(cust,bal)`*. Bob implements the transaction shown in Figure 4.10 that runs an update followed by an insert using a query that detects overdrafts. After some tests that are uneventful, Bob's solution is deployed. However, it turns out that Bob's transaction does not always report overdrafts correctly. Assume that transactions T_1 and T_2 as shown in Figure 4.11 have been executed concurrently in the order shown in Figure 4.12. Figure 4.14 shows the database state before the execution of transactions, Figure 4.15 after the execution of T_1, and Figure 4.16 after the execution of T_2. Note that the two tuples in table* `account` *were created by Transaction T_0. We omit the code for this transaction since it is not relevant to the discussion. As shown in Figure 4.16, these transactions cause an overdraft for Alice that is evident in the database state after T_2's commit (since $-20 + (-10) < 0$). However, neither T_1 nor T_2 have reported this overdraft. The cause of this problem is that SI does not guarantee serializability. In fact, it can lead to a type of concurrency anomaly called write-skew. A write-skew is the cause of the unexpected behavior of T_1 and T_2. Under SI, a transaction T runs*

over a private snapshot of the database that contains changes made by transactions committed before T started. Thus, T_1 and T_2 do not see each others changes. Both transactions compute the total balance using an outdated balance for the other account. For instance, T_2 sees the previous balance of $50 for Alice's checking account and the condition of the overdraft check evaluates to $50 + (-10) = 40 \not< 0$.

If MV-semiring provenance is captured using reenactment, then Bob can inspect the versions of the account and overdraft tables seen by the execution of Transaction T_2 that lead to the missing overdraft tuple. Using this information he would be able to determine that the problem was caused by reading an outdated balance.

We now discuss how to derive the MV-semiring relation $account[T_2]$, the version of relation `account` produced by Transaction T_2 tracing provenance back to the snapshot of the database seen by Transaction T_2. Afterwards, we discuss its relational encoding and how it aids Bob in debugging his transactions.

MV-semiring Encoding for the Running Example. Figure 4.13 shows the part of the history corresponding to the execution of T_1 and T_2. The provenance of relation `account` restricted to Transaction T_2 is computed as follows. First we compute $account[T_2, 13]$, the version of relation `account` visible within Transaction T_2 after its commit. Then we use provenance filtering as described in Section 4.8 to remove parts of the provenance that are unrelated to this transaction.

Computing $account[T_2, 13]$. Intuitively, the database state as seen by Transaction T_2 at version 13 (right after its commit) should reflect the changes resulting from the transaction's update and commit. The insert operation of this transaction affects relation `overdraft` and, thus, should not have any effect on the `account` relation. For $account[T_2, 13]$, the fourth case of the definition of a historic relation according to Figure 4.8a applies, because $End(T_2) = 13 - 1$ holds. Thus, $account[T_2, 13]$ is

computed by applying the commit operation $\mathcal{C}[T_2, 12]()$ to the database visible to Transaction T_2 at the time of its commit ($account[T_2, 12]$). Through repeated application of the fifth case we can deduce that $account[T_2, 12] = account[T_2, 9]$ since none of the operations of Transaction T_2 that were executed between versions 9 and 12 affect relation `account`. For $account[T_2, 9]$, the third case of Figure 4.8a applies since this version is the result of the update of Transaction T_2 that was executed at version 8. Using u_1 to denote this update, we get $account[T_2, 9] = u_1(account[T_2, 8])$. For version $account[T_2, 8]$, the second case applies. Thus, $account[T_2, 8] = account[8]$. This version in turn is computed as the union (sum) of all versions created by transactions committed before version 8. In our example the only transaction fulfilling this condition is Transaction T_0 and, thus, $account[8] = account[3]$. In summary, we get $account[T_2, 13] = \mathcal{C}[T_2, 12](u_1(account[3]))$. To compute this expression, we have to apply the definitions of update and commit operations according to Definition 3.

Applying u_1. Figure 4.14 shows how to compute the annotations of tuples in the result of the update. The annotation of the tuple $t_1 = (Alice, Checking, 50)$ in the result of u_1 is computed as shown on the top of this figure. Recall that A denotes the update expressions of u_1 ($bal - 40 \rightarrow bal$). Since this tuple does not fulfill the update's condition the expression $(\neg\theta)(t)$ evaluates to true. The outer sum in the right hand side of the expression evaluates to 0, because all tuples that after applying update u_1's expression would be equal to t_1 either are annotated with 0 in the input (they are not part of the input) and/or do not fulfill the update's condition ($typ = \text{'}Savings\text{'} \wedge name = \text{'}Alice\text{'}$).

Now consider the second tuple $t_2 = (Alice, Savings, 30)$ of relation `account`. This tuple fulfills the condition of the update and is updated to ${t_2}' = (Alice, Savings, -10)$. The derivation of the annotation of ${t_2}'$ in the result of the update is shown in Figure 4.20 (bottom). Tuple ${t_2}'$ is annotated with 0 in the input and, thus, the first sum-

mand evaluates to 0. Tuple t_2 is the only tuple that fulfills the condition $t' : t'.A = t_2{}'$ in the outer sum. The normalized annotation of t_2 in the input of the update is $C^2_{T_0,3}(I^2_{T_0,1}(x_2))$. This expression is wrapped into an update annotation $U^2_{T_2,9}$ and $(typ = \text{'}Savings\text{'} \wedge name = \text{'}Alice\text{'})(t_2)$ evaluates to 1. Hence, the annotation of $t_2{}'$ in the result of the update is as shown in Figure 4.20.

Tuples t_1 and $t_2{}'$ are the only tuples that exist in the output of the update. That is, the annotation of all other tuples in the result of the update is 0 and, thus, we do not show these tuples here.

Applying the commit. To compute $account[T_2, 13]$, we have to evaluate the commit operation $\mathcal{C}[T_2, 12]$ over the output of u_1. This operation wraps every summand of an annotation of a tuple in the input of the operation in a commit annotation if it is affected by T_2. This is checked by comparing the transaction in the outmost version annotation of a summand with T_2. In the output of u_1, only the single summand in the annotation of $t_2{}'$ fulfills this condition and, thus, is wrapped in $C^2_{T_2,13}$. The final result is shown in Figure 4.17.

Computing $account[T_2]$. To compute $account[T_2]$, we filter the instance $account[T_2, 13]$ by 1) removing summands that do not contain version annotations of T_2 (were not affected by T_2) and 2) replace subexpressions in summands that correspond to operations of transactions that committed before Transaction T_2's start with fresh variables. Note that 1) is optional in the implementation of provenance filtering in GProM, i.e., the user can choose whether tuples unaffected by Transaction T_2 are shown or not. Here we show the variant which only applies 2). This is achieved by replacing each subexpression of the form $C^{id}_{T,\nu}(k)$ where $T \neq T_2$ with $C^{id}_{T,\nu}(x_{new})$. Here x_{new} is a fresh variable that does not occur in any annotation of the input database (relation versions $T_2[13, account]$ and $T_2[13, overdraft]$).

The single summand in the annotation $C^{1}_{T_0,3}(I^{1}_{T_0,1}(x_1))$ of tuple t_1 was produced by Transaction T_0. We replace the subexpression $I^{1}_{T_0,1}(x_1)$ in this annotation with a new variable, say x_3. Similarly, the only summand in the annotation $C^{2}_{T_2,13}(U^{2}_{T_2,9}(C^{2}_{T_0,3}(I^{2}_{T_0,1}(x_2))))$ of tuple $t_2{}'$ contains a subexpression $C^{2}_{T_0,3}(I^{2}_{T_0,1}(x_2)))$. We replace $I^{2}_{T_0,1}(x_2))$ in this subexpression with a new variable x_4. The result is shown in Figure 4.18. If we would have also applied 1) then tuple t_1 would have been annotated with 0 since it was not affected by Transaction T_2.

Relational Encoding of Provenance. Now that we have discussed in detail how $account[T_2]$ is computed, we proceed to show how this MV-relation is encoded as a standard bag semantics relation. Figure 4.19 shows the encoding of $account[T_2]$. The $\mathbb{N}[X]^{\nu}$-expression represented by a tuple in this encoding is shown to the right of each tuple. The schema of the account relation is extended with so-called provenance attributes that store part of the annotation of a tuple. Each summand in the annotation of a tuple is encoded as one tuple in this encoding. A variable in an $\mathbb{N}[X]^{\nu}$-expression is represented in the relational encoding by the tuple that is annotated with this variable. Existence and nesting of version annotations is encoded in boolean attributes.

In our example, variables in the annotations of tuples correspond to tuples from the account relation that were valid at the start of Transaction T_2. Variable x_3 corresponds to tuple t_1 and variable x_4 corresponds to tuple t_2. To be able to store the tuples corresponding to these variables, provenance attributes for relation `account` are appended to the schema. Here P denotes an attribute renaming function. The details of this function are irrelevant to the discussion. In a nutshell, the function adds a prefix `prov_` to identify attributes as provenance attributes and includes the relation name, input attribute name, and the update in the returned attribute name. For instance, in our example $P(account, cust, u_1) =$`prov_account_cust_u1`. The presence

of version annotations is encoded using boolean attributes $\mathcal{U}_i$, one for each updates of a transaction that affected the relation for which we are tracking provenance. For this example, there is a single attribute $\mathcal{U}_1$ denoting the presence of a version annotation for u_1.

Consider the first tuple in the encoding. It stores the single summand $C^1_{T_0,3}(I^1_{T_0,1}(x_1))$ in the annotation of tuple t_1. Recall that variable x_3 denotes tuple $t_1 = (Alice, Checking, 50)$. Thus, the provenance attributes $P(account, x, u_1)$ for $x \in \{cust, typ, bal\}$ are used to store this tuple. The annotation of t_1 does not contain a version annotation corresponding to u_1 and, thus, $\mathcal{U}_1$ is set to false.

The annotation of the second tuple ($t_2{}'$) contains a variable x_4 (corresponding to tuple $t_2 = (Alice, Savings, 30)$). Thus, the provenance attributes are used to store this tuple. The annotation of this tuple contains a version annotation $U^2_{T_2,9}$, i.e., u_1 has affected the tuple. Thus, attribute $\mathcal{U}_1$ is set to true for this tuple.

Using Transaction Provenance for Debugging. Now consider how Bob can use our transaction debugger to figure out the cause of the error in his transaction. This debugger provides a more convenient visual way to explore the transaction provenance computed by our implementation of reenactment in GProM.

Example 17. *To find the cause of the missing overdraft, Bob needs to understand the internal details of the execution of transactions T_1 and T_2. The debugging panel for Transaction T_2 shows the states of relation* `account` *encoded in account$[T_2]$ as separate tables - one for the state of this relation before the start of Transaction T_2 and one after the execution of the transaction's operations. Note that GProM optionally also exposes intermediate states of a relation after the execution of each update which required only minor modifications to the reenactment approach. However, for this example, only one update of Transaction T_2 affected the account relation and, thus, this functionality is not needed for debugging T_2. Bob observes that this transaction*

sees an outdated balance of \$50 for Alice's checking account which is why the insert's query does not detect the overdraft ($50 + (-10) = 40 \not< 0$)*. Similarly, debugging the execution of Transaction* T_1*, Bob realizes that also this transaction is not seeing the updated balance produced by* T_2*. Thus, Bob has identified the write-skew that caused the error and can proceed to fix it, e.g., by changing his transaction code to update both the checking and savings account of a customer. This would ensure that* T_1 *and* T_2 *cannot be executed concurrently and, thus, prevents future instances of this type of write-skew. For example, one way to achieve this is by adding the following statement to Bob's transaction:*

```
UPDATE account SET bal = bal + 0 WHERE cust = :name AND
 typ <> :type;
```

This example demonstrates the effectiveness of our model in exposing the details of a transaction's execution including its interactions with other concurrently running transactions. As shown in this example, the model correctly encodes data-dependencies and dependencies of tuples on update operations - even under concurrency anomalies such as write-skews. By implementing reenactment and provenance capture for transactions as an SQL extension in GProM we provide a solid platform for developing more specialized applications - such as our transaction debugger - which leverage the transaction provenance information captured by GProM.

Note that it would have not been possible for Bob to determine the cause of the error using an audit log along, since the audit log only records which SQL commands were executed by which transaction. To understand the cause of the error, Bob needs to understand which tuple versions were seen and modified by an update. However, the audit log provides no such information. Time travel by itself is also not sufficient. The committed database versions accessible by time travel (Figure 4.15 and 4.16) also do not directly expose this problem. A very experienced user may recognize that T_2

will see an outdated account balance based on the snapshots exposed by time travel. However, this requires a deep understanding of the SI protocol. Using our debugger, this information is explicitly shown when debugging T_2. Furthermore, if the error is based on an update from a transaction that contains multiple update statements, then it would be virtually impossible to debug an error using time travel and audit logs alone.

[t]

4.9 Summary

We presented MV-semirings, a provenance model for database queries, updates, and concurrent transactions. In our model, each tuple is annotated with a symbolic expression that encodes from which previous tuple versions it was derived and how these tuple versions were combined by the derivation. It presents which updates of which transactions were involved in the derivation. Furthermore, we studied the SI and RC-SI concurrency control protocol for this model. The provenance of a tuple version in this model encodes its complete derivation history including previous tuple versions that were used to compute it and how tuple versions have been used by updates and/or queries involved in its creation. Similar to how semiring annotations generalize extensions of the relational model (including bag semantics) and several provenance models, our model generalizes these extensions under transactional updates. We presented a showcase how our provenance model for transactions is applied for post-mortem debugging of transaction executions.

Employee

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

Bonus

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

Figure 4.6. Running example database instance

Employee

	ID	Name	Position	
$C^1_{T_7,26}(U^1_{T_7,21}(C^1_{T_0,6}(I^1_{T_0,2}(x_1))))$	101	Mark Smith	Software Architect	$e_1{}'$
$C^2_{T_0,6}(I^2_{T_0,3}(x_2))$	102	Susan Sommers	Software Architect	e_2
$C^3_{T_0,6}(I^3_{T_0,4}(x_3))$	103	David Spears	Test Assurance	e_3

Bonus

	ID	EmpID	Amount	
$C^4_{T_7,26}(U^4_{T_7,24}(C^4_{T_8,23}(U^4_{T_8,22}(C^4_{T_1,10}(I^4_{T_1,8}(x_4))))))$	1	101	2500	$b_1{}''$
$C^5_{T_2,14}(I^5_{T_2,12}(x_5))$	2	102	2000	b_2
$C^6_{T_4,18}(I^6_{T_4,16}(x_6))$	3	103	500	b_3

Figure 4.7. New and modified tuples after execution of the example history under RC-SI (version 26).

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

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

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

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

(c) Valid Tuple Versions from Transaction T at ν

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

(d) Tuple Versions Updated By Transaction T

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

Figure 4.8. SI historic database definition

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

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

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

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

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

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

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

Figure 4.9. RC-SI historic database definition

```
UPDATE account SET bal = bal - :amount
WHERE cust = :name AND typ = :type;

INSERT INTO overdraft (
 SELECT a1.cust, a1.bal + a2.bal
 FROM account a1, account a2
 WHERE a1.cust = :name AND a1.cust = a2.cust
       AND a1.typ != a2.typ AND a1.bal + a2.bal < 0
);

COMMIT;
```

Figure 4.10. Bob's Transaction

Transaction	:name	:amount	:type
T_1	Alice	70	Checking
T_2	Alice	40	Savings

Figure 4.11. Bind Parameters for Transactions T_1 and T_2

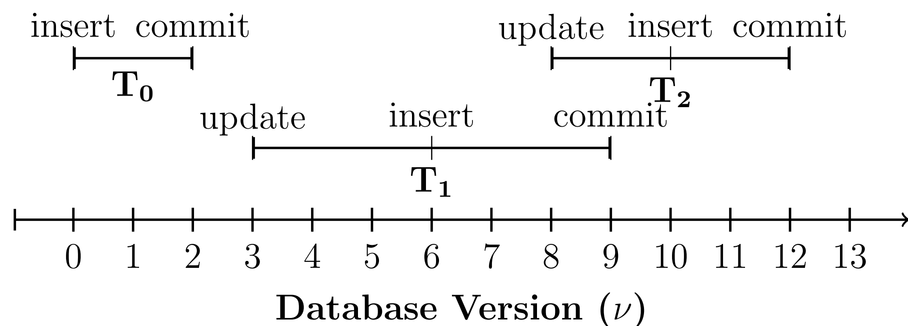


Figure 4.12. Execution Order of Transactions T_1 and T_2

$$T_1 = (\mathcal{U}[typ = \text{'}Checking\text{'} \wedge name = \text{'}Alice\text{'}, bal - 70 \rightarrow bal, T_1, 3](account),$$
$$\mathcal{I}[Q_1, T_1, 6](overdraft),$$
$$\mathcal{C}[T_1, 9])$$

$$T_2 = (\mathcal{U}[typ = \text{'}Savings\text{'} \wedge name = \text{'}Alice\text{'}, bal - 40 \rightarrow bal, T_2, 8](account),$$
$$\mathcal{I}[Q_2, T_2, 10](overdraft),$$
$$\mathcal{C}[T_2, 12])$$

$$Q_1 = \Pi_{cust, bal+bal' \rightarrow bal}(\sigma_{cust=Alice}(account)$$
$$\bowtie_{typ \neq typ' \wedge cust = cust' \wedge bal + bal' < 0}$$
$$\rho_{cust' \leftarrow cust, typ' \leftarrow typ, bal' \leftarrow bal}(account))$$
$$Q_2 = \Pi_{cust, bal+bal' \rightarrow bal}(\sigma_{cust=Alice}(account)$$
$$\bowtie_{typ \neq typ' \wedge cust = cust' \wedge bal + bal' < 0}$$
$$\rho_{cust' \leftarrow cust, typ' \leftarrow typ, bal' \leftarrow bal}(account))$$

Figure 4.13. History corresponding to T_1 and T_2

account[3]

	cust	typ	bal
$C^1_{T_0,3}(I^1_{T_0,1}(x_1))$	Alice	Checking	50
$C^2_{T_0,3}(I^2_{T_0,1}(x_2))$	Alice	Savings	30

overdraft[3]

cust	bal

Figure 4.14. Database before execution of T_1 and T_2 ($\nu = 3$)

account[**10**]

	cust	typ	bal
$C^1_{T_1,10}(U^1_{T_1,4}(C^1_{T_0,3}(I^1_{T_0,1}(x_1))))$	Alice	Checking	-20
$C^2_{T_0,3}(I^2_{T_0,1}(x_2))$	Alice	Savings	30

overdraft[**10**]

cust	bal

Figure 4.15. Database after execution of T_1 ($\nu = 10$)

account[**13**]

	cust	typ	bal
$C^1_{T_1,10}(U^1_{T_1,4}(C^1_{T_0,3}(I^1_{T_0,1}(x_1))))$	Alice	Checking	-20
$C^2_{T_2,13}(U^2_{T_2,9}(C^2_{T_0,3}(I^2_{T_0,1}(x_2))))$	Alice	Savings	-10

overdraft[**13**]

cust	bal

Figure 4.16. Database after execution of T_2 ($\nu = 13$)

account[T_2, **13**]

	cust	typ	bal
$C^1_{T_0,3}(I^1_{T_0,1}(x_1))$	Alice	Checking	50
$C^2_{T_2,13}(U^2_{T_2,9}(C^2_{T_0,3}(I^2_{T_0,1}(x_2))))$	Alice	Savings	-10

overdraft[T_2, **13**]

cust	bal

Figure 4.17. Database visible to T_2 at its commit ($\nu = 13$)

account$[T_2]$

	cust	typ	bal
$C^1_{T_0,3}(x_3)$	Alice	Checking	50
$C^2_{T_2,13}(U^2_{T_2,9}(C^2_{T_0,3}(x_4)))$	Alice	Savings	-10

overdraft$[T_2]$

cust	bal

Figure 4.18. Provenance restricted to Transaction T_2

MV-semiring expression	**account After** T_2			**account before** T_2			
	cust	**typ**	**bal**	**P(cust)**	**P(typ)**	**P(bal)**	$\mathcal{U}_1$
$C^1_{T_0,3}(x_3)$	Alice	Checking	50	Alice	Checking	50	False
$C^2_{T_2,13}(U^2_{T_2,9}(C^2_{T_0,3}(x_4)))$	Alice	Savings	-10	Alice	Savings	30	True

Figure 4.19. Relational encoding of $\mathbb{N}[X]^{\nu}$-relation `account` with annotations restricted to Transaction T_2 ($account[T_2]$)

$$\mathcal{U}[\theta_1, bal - 40 \rightarrow bal, T_2, 8](account)(t_1) \text{ with } \theta_1 := (\text{'}Savings\text{'} \wedge name = \text{'}Alice\text{'})$$

$$= account(t_1) \cdot (\neg\theta_1)(t_1) + \sum_{t':t'.A=t_1} \sum_{i=0}^{n(account(t'))} U_{T_2,9}^{id(account(t')[i])}(account(t')[i]) \cdot (\theta_1)(t')$$

$$= (C^1_{T_0,3}(I^1_{T_0,1}(x_1)) \cdot 1 + 0$$

$$= C^1_{T_0,3}(I^1_{T_0,1}(x_1))$$

$$\mathcal{U}[\theta_2, bal - 40 \rightarrow bal, T_2, 8](account)(t_2{}') \text{ with } \theta_2 := (typ = \text{'}Savings\text{'} \wedge name = \text{'}Alice\text{'})$$

$$= account(t_2{}') \cdot (\neg\theta_2)(t_2{}') + \sum_{t':t'.A=t_2{}'} \sum_{i=0}^{n(account(t'))} U_{T_2,9}^{id(account(t')[i])}(account(t')[i]) \cdot (\theta_2)(t')$$

$$= 0 + U^2_{T_2,9}(C^2_{T_0,3}(I^2_{T_0,1}(x_2)))$$

$$= U^2_{T_2,9}(C^2_{T_0,3}(I^2_{T_0,1}(x_2)))$$

Figure 4.20. Evaluating the update of Transaction T_2

```
INSERT INTO employee VALUES
(Mark Smith, Software Engineer, 0);
UPDATE employee SET bonus = bonus + 500
WHERE position = 'Software␣Engineer';
COMMIT;
```

$$T = (\mathcal{I}[\{(MarkSmith, SoftwareEngineer, 0)\}, T, 1](employee),$$

$$\mathcal{U}[position = \text{'}SoftwareEngineer\text{'}, bonus + 500 \rightarrow bonus, T, 2](employee),$$

$$\mathcal{C}[T, 3])$$

Figure 4.21. Alice's transaction

(a) Database before execution of T

employee

name	position	Bonus
Susan Sommers	Software Engineer	500
David Spears	Test Assurance	500

(b) Database after execution of T

account

name	position	Bonus
Susan Sommers	Software Engineer	1000
David Spears	Test Assurance	500
Mark Smith	Software Engineer	500

Figure 4.22. Database states before and after the execution of Alice's transaction

CHAPTER 5
REENACTMENT

We need an approach to compute provenance for updates and transactions efficiently. In many use cases provenance is needed only once in a while. However, it is hard to predict when we will need it and for which data or operations. The problem with materializing provenance eagerly during transactions processing is that there is a need to change transactions and/or DBMS and there is an overhead for every transaction. We also need to to know how far back in time should a provenance record. We introduce reenactment that can be used to capture provenance for any transaction on demand and it has low overhead for transactions if no provenance is requested.

Reenactment queries are queries that simulate the effect of an update, transaction, or even a whole history. Importantly, these queries are annotation equivalent to the operation(s) they are simulating, i.e., they produce the same result (updated relations) and have the same provenance. Reenactment is the main enabler of our approach for computing the provenance of transactions, because it enables us to compute provenance retroactively by running reenactment queries instead of having to compute and materialize it eagerly while transactions are running. In this chapter, we show if we extend our query model with a new operator that creates version annotations, then any update, transaction, or (partial) history in our model can be equivalently expressed as a query. First, we define reenactment queries for single update operations. Then, we determine how to combine reenactment queries for updates into a reenactment query for a transaction. The reenactment query simulates

interactions among transaction.

5.1 Reenactment Queries

Reenactment captures provenance for an update u or transaction T by executing an annotation equivalent *reenactment query* $\mathbb{R}(u)$ or $\mathbb{R}(T)$, respectively. Annotation equivalent ($\equiv_{\mathbb{N}[X]^\nu}$) means that such a query produces the same result and provenance. Recall that this implies equivalence for any MV-semiring $\mathcal{K}^\nu$. The equivalence under annotated semantics between an operation and its reenactment query has several important implications: instead of computing provenance eagerly during transaction execution , which would require us to pay the runtime and storage overhead of provenance for every transaction, we retroactively compute it by running reenactment queries. Furthermore, since our model generalizes bag-semantics, we can use reenactment to recreate a database state valid at a particular time by simply running a query - including database states that were only visible within one transaction. We introduce a new operator that adds version annotations, because this is required for reenactment since the operators of $\mathcal{RA}^+$+ do not introduce new version annotations.

Definition 19. *The operator $\alpha_{X,T,\nu}(R)$ for $X \in \{I, U, D\}$ takes as input a $\mathcal{K}^\nu$-relation R and wraps every summand in a tuple's annotation in $X_{T,\nu}$. The commit annotation operator $\alpha_{C,T,\nu}(R)$ only wraps summands produced by Transaction T using operator* $\textsc{com}[T,\nu](k)$ *from Definition 16.*

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

5.2 Update Reenactment

We first define reenactment for an update u that is executed over the historic database seen by u's transaction T at the time of the update ($R[T, \nu(u)]$). Here we

abuse notation and treat $R[T,\nu]$ as a syntactic construct that we can substitute with an algebraic expression which computes this version of R. For example, $Q(D[T,\nu])$ denotes the query Q where every access to a relation R is substituted by $R[T,\nu]$.

Definition 20. *Let H be a history over database D. The reenactment query $\mathbb{R}(u)$ for an operation u in H is:*

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

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

Example 18. *Consider the reenactment query for the first update operation $\mathbb{R}(u_1)$ of Transaction T_6 shown in Figure 4.1. $u_1 = \mathcal{U}[cust =' Alice' \wedge typ =' Checking', bal - 1500 \rightarrow bal, T_6, 11](Account)$ of example Transaction T_6. The reenactment query $\mathbb{R}(u_1)$ is:*

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

Theorem 7. *Let u be an update. Then, $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$.*

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

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

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

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

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

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

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

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

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

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

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

□

5.3 SI Reenactment

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

Definition 21. *Let T be a transaction in a history H. The reenactment query $\mathbb{R}(T)$ for T is:*

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

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

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

Pick a relation mention $S[T, \nu]$ in the current $\mathbb{R}^R(u)$

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

Technically, the reenactment of a transaction T is a set of queries. However, abusing terminology we refer to this set as the reenactment query of T and by $T \equiv_{\mathbb{N}[X]^\nu}$

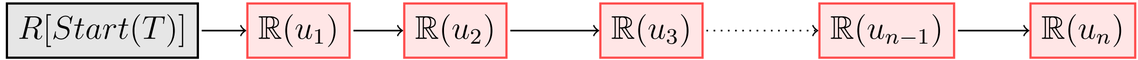


Figure 5.1. Structure of the reenactment query for SI

$\mathbb{R}(T)$ denote that each reenactment query for a relation R is equivalent to the effect that transaction T has on this relation. The structure of the reenactment query for SI transactions updating a single relation R is outlined in Figure 5.1.

Example 19. *Consider Transaction T_5 from the running example shown in Figure 1.1. Let us refer to its operations as u_1 and u_2. We use the following abbreviations in this example:* `Account` $=$ `A`*, cust $=$ c, typ $=$ t, and bal $=$ b. Consider the construction of the reenactment query for T_5 on* `A`*. The last update modifying* `A` *is u_2. Thus, $\mathbb{R}^A(T_5) = \alpha_{C,T,13}(\mathbb{R}^A(u_2))$. Operation u_2 updates relation* `A` *at version 12. The reenactment query for u_2 is:*

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

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

$$\mathbb{R}^A(T_5) = \alpha_{C,T,13}(\mathbb{R}^A(u_2))$$

$$\mathbb{R}^A(u_2) = \alpha_{U,T_5,13}(\Pi_{c,t,(b+300)\rightarrow b}(\sigma_{t='Savings' \wedge b>5000}(\mathbb{R}^A(u_1)))) \quad \cup \sigma_{\neg(t='Savings' \wedge b>5000)}(\mathbb{R}^A(u_1))$$

$$\mathbb{R}^A(u_1) = \alpha_{U,T_5,11}(\Pi_{c,t,(b+100)\rightarrow b}(\sigma_{t='Savings'}(A[10]))) \quad \cup \sigma_{\neg(t='Savings')}(A[10])$$

Theorem 8. *Let T be a transaction. Then, $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$.*

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

Induction Start: For a transaction with a single update u_1, the theorem follows from the equivalence result for updates and a simple check for the equivalence of the annotation operator for commits and commit annotations produced by T.

Induction Step: Assume that we have proven that reenactment is annotation equivalent for transactions with up to i updates. We have to show that the same holds for any $T = u_1, \ldots, u_i, u_{i+1}, c$. Let $T_i = u_1, \ldots, u_i, c$. In the induction start we have already proven that the commit operation of a transaction are equivalent to the commit annotation operator in its reenactment query. Thus, we ignore the existence of commit operations in the following proof. WLOG assume $End(T) = End(T_i)$. We know that $\mathbb{R}(T_i) \equiv_{\mathbb{N}[X]^\nu} T_i = R[T_i, End(T_i)] = R[T_i, \nu(u_i) + 1]$. Since T_i and T have executed the same updates over the same input it follows that $R[T_i, \nu(u_i) + 1] = R[T, \nu(u_i)+1]$. From the definition of historic databases we know that $R[T, End(T)] = R[T, \nu(u_{i+1}) + 1] = u_{i+1}(R[T, \nu(u_{i+1})])$. Using the equivalences stated above we can deduce $u_{i+1}(R[T, \nu(u_{i+1})]) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}(\mathbb{R}^R(u_i)))$. We know that $\mathbb{R}(u_{i+1}) \equiv_{\mathbb{N}[X]^\nu} u_{i+1}$ and, thus, it follows that $R[T, End(T)] \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}^R(u_{i+1})$. Since $\mathbb{R}^R(u_{i+1}) = \mathbb{R}^R(T)$, this concludes the proof. □

5.4 RC-SI Reenactment

Based on our model for RC-SI histories that we did present in Sec. 4.8, we have to construct reenactment queries for updates of a transaction T such that $R_{ext}[T, \nu(u)]$ is the input of every update u over relation R. $R_{ext}[T, \nu(u)]$ contains tuple versions from $R[\nu]$ and also those tuples from R that have been modified by previous updates of the transaction T. Thus, we can compute it as a union between these two sets of tuple versions by filtering out invalid tuple versions. In [7], we introduced a query operator (Version Merge Operator) which filters invalid versions and can be used in reenactment for RC-SI transactions.

(a) Version Merge Operator

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

(b) Check Tuple Versions of a Relation R

$$isMax(R, k) = 0 \text{ if } \exists t', k', j : idOf(R(t')[j]) =$$
$$idOf(k) \wedge versionOf(R(t')[j]) > versionOf(k), 1 \text{ otherwise}$$
$$isStrictMax(R, k) = 0 \text{ if } \exists t', k', j : idOf(R(t')[j]) =$$
$$idOf(k) \wedge versionOf(R(t')[j]) \geq versionOf(k), 1 \text{ otherwise}$$

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

Figure 5.2. Definition of auxiliary RC-SI reenactment operators

Version Merge Operator. This operator merges two version R_1 and R_2 of a relation R such that the output includes 1) each tuple version that exists in both versions only once and 2) the newer version of each tuple which exists in both inputs (shown in Figure 5.2a). We construct $R_{ext}[T, \nu]$ using this operator. This operator uses two functions $isMax$ and $isStrictMax$ that are explained in the following section.

Example 20. *As an example consider relation versions are shown in Figure 4.6 and 4.7 for computing $\mu(Bonus[26], Bonus[19])$. The later only shows new or updated tuples. For instance, b_2 is present in both relations with the same annotation, a single summand. Thus, the first sum in $\mu(Bonus[26], Bonus[19])(b_2)$ will include this annotation (there is no newer version of this tuple in $Bonus[19]$) while*

it will be excluded from the second sum (the same annotation is found in $Bonus[26]$*). As another example consider tuple* b_1 *which was updated to* ${b_1}'$ *by Transaction* T_7*. Thus,* $\mu(Bonus[26], Bonus[19])(b_1) = 0$*, because a newer version of this tuple exists in* $Bonus[26]$ *and* $\mu(Bonus[26], Bonus[19])({b_1}') = Bonus[26])({b_1}')$ *(this is the newest version of this tuple found in* $Bonus[19]$ *and* $Bonus[26]$*).*

Check Tuple Versions of a Relation R. These functions are shown in Figure 5.2b. $isMax(R, k)$ returns 0 when relation R has a newer version of the tuple version encoded as annotation k. Function $isStrictMax$ is a strict version of $isMax$ function that also returns 0 when the tuple version k exists in R. These functions call $idOf(k)$ to access the tuple identifier and $versionOf$ to retrieve the version encoded in the annotation k. These functions are applicable for annotations in a normalized admissible $\mathcal{K}^{\nu}$-relation (see Sec. 4).

RC-SI transactions that modify multiple relations are handled analog to SI. Hence, we only present the construction of reenactment queries for RC-SI transactions that update a single relation R. The reenactment query for the Transaction $T = (u_1, \ldots, u_n, c)$ executed under RC-SI can be defined recursively. It is constructed starting with a commit annotation operator applied to the reenactment query $\mathbb{R}(u_n)$ for the last update of T. Then for $i \in n-1, \ldots, 1$ we replace $R[T, \nu(u_{i+1})]$ in the query constructed so far with $\mu(\mathbb{R}(u_i), R[\nu(u_{i+1})])$. Operator μ computes $R_{ext}[T, \nu(u_{i+1})]$ which is the input seen by u_{i+1}. The structure of the reenactment query for RC-SI transactions for a single relation R is shown in Figure 5.3.

Theorem 9. *If* T *is a RC-SI transaction, then* $T \equiv_{\mathbb{N}[X]^{\nu}} \mathbb{R}(T)$*.*

Proof. Assume that transaction $T = u_1, \ldots, u_n, c$ is updating a single relation R. We need to show that the input $R[T, \nu(u)]$ for an update u is the same as the input

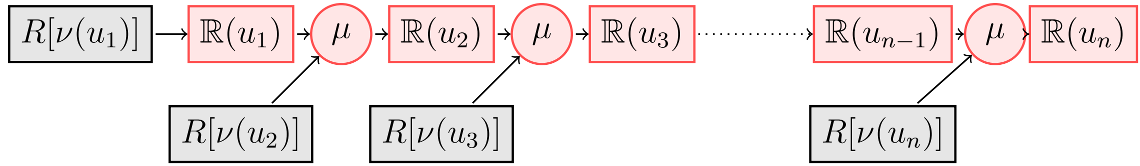


Figure 5.3. Structure of the reenactment query for RC-SI

produced for $\mathbb{R}(u)$ by the reenactment query for Transaction T. We prove this fact by induction over the number of updates in Transaction T. Induction Start: Let $T = u_1, c$. This case is analog to SI.

Induction Step: Assume that $R[T, \nu(u_i)] = R_{ext}[T, \nu(u_i)]$ for any $i \leq n$ is the number of operations in Transaction T. We need to prove that for any transaction $T = u_1, \ldots, u_{n+1}, c$ we have that $R_{ext}[T, \nu(u_{n+1})]$ is equal to the input for the reenactment query $\mathbb{R}(u_{n+1})$ of u_{n+1} within the reenactment query $\mathbb{R}(T)$. In the reenactment query, the input to $\mathbb{R}(u_{n+1})$ is $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])$. Based on the induction hypothesis we have $\mathbb{R}(u_n) = R[T, \nu(u_{n+1})]$. Thus, denoting $\nu(u_{n+1})$ as ν_{n+1}:

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

Note that $R_{ext}[T, \nu_{n+1}](t)$ is also defined as a sum over the elements from $R[T, \nu_{n+1}](t)$ and $R[\nu_{n+1}](t)$. Individual summands are filtered out using VALIDIN and VALIDEX. Thus, to prove that $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})]) = R_{ext}[T, \nu(u_{n+1})]$, we have to show that if the $isMax$ or $isStrictMax$ function returns 1 on a summand then the same is true for VALIDIN or VALIDEX, respectively. Fixing a tuple t and a tuple version (summand) k with tuple identifier id in its annotation, we have to distinguish between five cases based on whether such a tuple version occurs in $R[\nu(u_{n+1})]$ and/or in $R[T, \nu(u_{n+1})]$, and, if it occurs in both, whether one of these versions is newer. We show the proof

for one of these cases. The remaining cases are similar in nature (see [8]). Case 1: For this case, we assume that the first tuple version with identifier $idOf(k)$ was created by an insert of Transaction T before ν_{n+1} and, thus k is only present in $R[T, \nu_{n+1}](t)$. Therefore, function $isMax(\ R[\nu_{n+1}], k)$ returns 1 and k is in $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Similarly, since k is the latest version, we have that $\textsc{validIn}(\ R[T, \nu_{n+1}], t, k, \nu_{n+1})$ returns 1 because k's outmost version annotation is from T. Thus, k is also present in $R_{ext}[T, \nu_{n+1}]$. Having proven that $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})]) = R_{ext}[T, \nu(u_{n+1})]$ it follows that $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$.

The complete proof is presented in Appendix A. □

Green demonstrated [41] that $Q \sqsubseteq_{\mathbb{N}[X]} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}} Q'$ if $\mathcal{K}$ is *naturally ordered* and, thus $Q \equiv_{\mathbb{N}[X]} Q' \Rightarrow Q \equiv_{\mathbb{N}} Q'$. The result is based on the existence of surjective semiring homomorphisms. We do not define what it means for a semiring to be naturally ordered here, but note that many important semirings including all semirings considered here are naturally ordered. Based on the theorem shown below this result translates to queries using the annotation operation defined above and updates in MV-semirings. Thus, reenactment queries also produce the same updated relation as the original operation under bag semantics.

Theorem 10. *For Q and Q' be two $\mathcal{RA}^+$ queries and $\mathcal{K}$ a naturally ordered semiring. Then $Q \equiv_{\mathcal{K}^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}} Q'$. Let Q and Q' be two updates or $\mathcal{RA}^+$ queries that may use the annotation operator α and $\mathcal{K}$ a naturally ordered semiring, then $Q \equiv_{\mathbb{N}[X]^\nu} Q' \Rightarrow Q \equiv_{\mathcal{K}^\nu} Q'$.*

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

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

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

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

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

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

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

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

Figure 5.4. Schema and instance of the relational encoding of $R[T]$

5.5 Relational Reenactment using Time Travel and Audit Logging

We now introduce techniques for retrieving the provenance of transactions using standard DBMS based on a relational encoding of reenactment queries. Our approach uses an audit log to determine which SQL statements were executed when and by which transaction. We demonstrate how reenactment queries can be translated into standard relational algebra queries with time travel that produce a relational encoding of provenance restricted to a transaction (as explained in Section 4.8).

Relational Encoding of $\mathcal{K}^{\nu}$-Relations. We extend the relational encoding of provenance polynomials introduced for the Perm [40] project with additional columns that encode version annotations. To encode the filtered provenance $R[T]$ of a transaction T we: 1) normalize $\mathcal{K}^{\nu}$-expressions according to the operations that were applied to the data and 2) use additional attributes to represent a normalized $\mathbb{N}[X]^{\nu}$-

polynomial.

Normal Form. The basic idea behind this encoding is to represent variables in a normalized polynomial by actual tuple values from the inputs of the query. In particular, we take a polynomial in the normal form introduced in Definition 13 that is a sum of products (and version annotations) and order the variables and version annotations in each summand according to the relation and update they belong to. Given the algebra tree for a reenactment query, variables in products mixed with version annotations are ordered according to the leaves of the algebra tree. We add additional attributes to be able to encode such a product and its version annotations, and represent each summand in a normalized polynomial as a separate tuple.

Definition 22. *Let T be a transaction. An $\mathbb{N}[X]^{\nu}$ annotation in $R[T](t)$ is in ordered normal form if it is normalized according to Definition 14 and variables in each summand k_i are sorted according to $\mathbb{R}^R(T)$.*

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

Definition 23. *Let T be a transaction, $\mathcal{U}_i$ denote a boolean attribute representing the version annotation of update $u_i \in T$, and $\mathcal{U}_C$ denote a boolean attribute representing the commit annotation of T. The name of attribute $\mathcal{U}_i$ encodes $\nu(u_i)$, the type of*

update u_i (insert, delete, update), and the query Q in case the update is an insert. We use $pos(u)$ to denote the position of update u in transaction T. The schema of the relational encoding $\text{REL}(R[T])$ *is defined in Figure 5.4.*

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

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

Definition 24. *Consider the provenance $R[T]$ of transaction T in ordered normal form and let $\textsc{Null}(\mathcal{P}(R))$ and $\textsc{Null}(Q)$ denote a list of null values with the same arity as $\mathcal{P}(R)$ and the provenance schema for Q (the list of $\textsc{Rel}^{\nu(u)}(R, X_j, k_j)$ attributes), respectively. Let $t(x)$ denote the tuple corresponding to a variable x. The relational encoding $\textsc{Rel}(R[T])$ is defined in Figure 5.4.*

The relational encoding of a normalized $\mathbb{N}[X]^\nu$-annotation of a tuple in $R[T]$ is constructed following the same procedure as for its schema. For each summand k in a ordered normalized annotation of a tuple t in $R[T]$ we create a tuple in $\textsc{Rel}(R[T])$ by concatenating t and the relational encoding of k. The encoding of k is constructed iteratively by encoding and stripping of parts of k corresponding to updates in T. Given this relational encoding we need to prove that it is lossless, i.e., the encoded MV-relation $R[T]$ can be recovered from $\textsc{Rel}(R[T])$.

Theorem 11. *The $\textsc{Rel}(R[T])$ operation is lossless.*

Audit Log and Time Travel. We require the DBMS we use for provenance computation to keep an audit log that stores the SQL code for each update plus 1) when the update was executed and 2) the identifier (*xid*) of its transaction. The audit log is used to determine the operations of a transaction and the database version they have accessed. We assume a standard SI based implementation of time travel as supported in similar fashion by multiple DBMS. Each tuple is annotated with a system time interval (transaction time) that encodes when this tuple version is valid in the database. Update operations create new tuple versions and invalidate tuple versions

that are updated by setting their end time to the current time. These modifications are only visible in the updating transaction. When a transaction commits, then new tuple versions are created for all modified tuples with start time set to the transaction commit time. A *snapshot* R_ν of relation R contains all committed tuple versions valid at ν. Snapshots have additional attributes TT_b and TT_e storing validity time intervals as well as Xid and Id storing the transaction and tuple identifiers, respectively.

Relational Implementation of Reenactment. We now discuss how $\mathcal{K}^\nu$-relational reenactment queries can be rewritten as standard relational (bag semantics) queries which produce $\textsc{Rel}(R[T])$ for a transaction T. This rewriting of $\mathcal{K}^\nu$-queries into bag semantics queries (expressible in SQL) extends previous results for rewriting $\mathcal{K}$-relational queries into bag semantics [40, 38]. A query is rewritten by recursively applying rewrite rules for single operators in a top-down fashion. We apply a selection on the boolean version annotation attributes to only return tuple versions from $R[T]$, i.e., that were affected by at least one update of transaction T.

Definition 25. *Let $T = u_1, \ldots, u_n, c$ be a transaction and let $\mathcal{U}_i$ denote the version annotation created by the ith update in T. The relational translation* $\text{TR}(\mathbb{R}^R(T))$ *of the reenactment query $\mathbb{R}^R(T)$ restricted to $R[T]$ is computed from $\mathbb{R}^R(T)$ as shown below. The rewrite operator* $\textsc{Rew}$ *is defined in Figure 5.5 and Figure 5.6.* $\textsc{Null}(\mathcal{P}(q))$ *denotes a singleton relation with null values for all provenance attributes of q except for version annotation attributes which are set to $false$. Furthermore, ID_x is a function that adds a suffix* `'_x'` *to attribute names in a list.*

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

The query produced by the rewrite rules of Figure 5.5 returns the relational encoding of provenance introduced previously. There are two rules for the union operator. The first one deals with reenactment of an update or delete operation u_i

where the annotation attributes of the left union input are the same as for the right input except for the version annotation attribute $\mathcal{U}_i$. The renaming applied in the rewriting of the annotation operator for the commit of the transaction ensures that the schema is that same as defined in Figure 5.4.

This translation of reenactment queries returns the encoding of $R[T]$ that as defined in Section 5.5.

Theorem 12. *Let T be a transaction. Then:*

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

Our reenactment-based implementation for tracking the provenance of transactions enables us to expose internal states of relations as seen by a statement of a past transaction, to show data-dependencies between tuple versions, and show which update statement of a transaction affected a tuple. Furthermore, we use reenactment to support what-if scenarios, i.e., evaluate the effect of hypothetical changes to the data or to the updates of a transaction. This functionality relies on the fact that reenactment can replay any history - no matter whether actual or hypothetical. This enables users to determine the root cause of an error and to test whether a change to a transaction's code fixes the error.

5.6 Summary

We presented reenactment, our novel technique for replaying a transactional history using queries. A reenactment query for a single update operations or a transaction returns the same database state and has the same provenance. Reenactment queries enable us to compute provenance retroactively by executing them instead of computing and materializing provenance information eagerly during their execution.

$$\textsc{Rew}(\{t \rightarrow x_{id}\}) = \Pi_{\textsc{Sch}(t),Id}(\{t \triangleright id\})$$

$$\textsc{Rew}(R[\nu]) = \Pi_{\textsc{Sch}(R),Xid,TT_b \rightarrow V,Id,\textsc{Sch}(R) \rightarrow P(R)}(R_\nu)$$

$$\textsc{Rew}(\sigma_\theta(q)) = \sigma_\theta(\textsc{Rew}(q))$$

$$\textsc{Rew}(\Pi_A(q)) = \Pi_{A,\mathcal{P}(q)}(\textsc{Rew}(q))$$

$$\textsc{Rew}(q_1 \cup q_2) = \textsc{Rew}(q_1) \cup$$

$$(\textsc{Rew}(q_2) \times \rho_{\mathcal{U}_i}(\{(false)\}))$$

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

$$\textsc{Rew}(q_1 \cup q_2) = (\rho_{\textsc{Sch}(q_1),ID_1(\mathcal{P}(q_1))}(\textsc{Rew}(q_1)) \times \textsc{Null}(ID_2(\mathcal{P}(q_2))))$$

$$\cup\ (\Pi_{\textsc{Sch}(q_2),\mathcal{P}(q_1 \cup q_2)}(\rho_{\textsc{Sch}(q_2),ID_2(\mathcal{P}(q_2))}(\textsc{Rew}(q_2))$$

$$\times\ \textsc{Null}(ID_1(\mathcal{P}(q_1)))))$$

$$\textsc{Rew}(q_1 \bowtie_\theta q_2) = \Pi_{\textsc{Sch}(q_1),\textsc{Sch}(q_2),\mathcal{P}(q_1 \bowtie_\theta q_2)}(\rho_{\textsc{Sch}(q_1),ID_1(\mathcal{P}(q_1))}(\textsc{Rew}(q_1))$$

$$\bowtie_\theta\ \rho_{\textsc{Sch}(q_2),ID_2(\mathcal{P}(q_2))}(\textsc{Rew}(q_2)))$$

$$\textsc{Rew}(\alpha_i(q)) = \Pi_{\textsc{Sch}(\textsc{Rew}(q)),true \rightarrow \mathcal{U}_i}(\textsc{Rew}(q))$$

$$\textsc{Rew}(\alpha_{C,T,End(T)}(q)) = \rho_{\textsc{Sch}(\textsc{Rel}(R[T]))}(\textsc{Rew}(q) \times \rho_{\mathcal{U}_C}(\{(true)\}))$$

$$(\text{for q} = \mathbb{R}^R(T))$$

Figure 5.5. Structural rewrite rules for translating $\mathcal{K}^\nu$-semantics reenactment queries into standard relational semantics (bag)

$$\mathcal{P}(\{t \rightarrow x_{id}\}) = Id$$

$$\mathcal{P}(R[\nu]) = \mathcal{P}(R)$$

$$\mathcal{P}(\sigma_C(q)) = \mathcal{P}(q)$$

$$\mathcal{P}(\Pi_A(q)) = \mathcal{P}(q)$$

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

$$\mathcal{P}(\alpha_i(q)) = \mathcal{P}(q) \triangleright \mathcal{U}_i$$

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

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

else:

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

Figure 5.6. Annotation attributes rules for translating $\mathcal{K}^\nu$-semantics reenactment queries into standard relational semantics (bag)

CHAPTER 6
IMPLEMENTATION

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

6.1 System Overview

Figure 6.1 shows an overview of GProM. The user interacts with the system using an extension of the underlying database system's SQL dialect. Specifically, we support new language constructs for computing and managing provenance (similar to Perm [40]). Incoming statements are translated into a relational algebra graph representation which we call *algebra graph model* (*AGM*). Similar to intermediate code representations used by compilers, this model is used as a representation of computation which is independent of the target language. If the statement does not use any provenance features, then the AGM model is translated back into the native

SQL dialect using a vendor specific SQL code generator.

Provenance Computation. Similar to Perm [40] (and other systems [51]) we represent provenance information using a relational encoding of provenance annotations. This representation is flexible enough to encode typical database provenance models including *PI-CS* [40] (and, thus, provenance polynomials [50]), Where- and Why-provenance [25], and many others. The *provenance rewriter* uses provenance-type specific rules to rewrite an input query q into a query q^+ that propagates annotations to produce this encoding of data annotated with provenance.

Supporting Past Queries, Updates, and Transactions. One unique feature of GProM is that the system can retroactively compute the provenance of queries, updates, and transactions. This feature requires that a log of database operations is available (we call this an *audit log*) and that the underlying database system supports *time travel*, i.e., querying past versions of a relation. These features are available in most database systems or can be added using extensibility mechanisms. An audit log paired with time travel functionality is sufficient for computing the provenance of past queries using simple modifications of standard provenance rewrites [74, 26]. Our main contribution is to demonstrate that this is also sufficient for tracking the provenance of updates and transactions. If the user requests provenance for a transaction T, the *transaction reenactor* extracts the list of SQL statements executed by T from the audit log and constructs a *reenactment query* $q(T)$ that simulates the effects of these statements. We use the provenance rewriter to rewrite $q(T)$ into a query $q(T)^+$ that computes the provenance of the reenacted transaction. Note that the construction of $q(T)$ is independent of the provenance[2] rewrite and $q(T)$ is a standard relational query. Using this approach, we can compute any type of provenance for

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

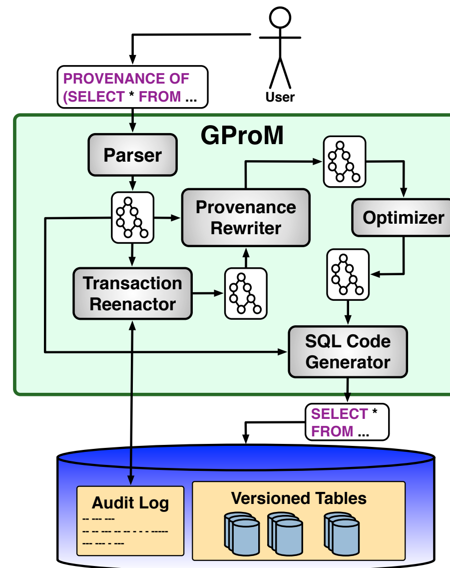


Figure 6.1. GProM architecture

updates, transactions, and across transactions as long as rewrite rules for computing the provenance of queries have been implemented for this provenance type.

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

6.2 Support for Updates And Transactions

GProm is the first system capable of computing provenance for queries, update operations, transactions, and across transaction boundaries. Provenance computation for updates and transactions is implemented in the *transaction reenactment* module of the system. We can retroactively compute the provenance of transactions (and across transactions) as long as two conditions are met: 1) the underlying database supports *time travel*, i.e., we can retrieve a past version of a relation and 2) the database keeps

a log of executed SQL statements (the aforementioned *audit log*). We require the audit log to contain at least the following information for each executed statement: an identifier for the transaction (*xid*) this statement was part of, a timestamp storing when the statement was executed, and the SQL code for the statement.

As observed by Zhang et al. [74], an audit log and time travel combined with standard provenance rewrites can be used to compute the provenance of past queries without the need to store any additional information. Chirigati et al. [26] also use these features to compute provenance of database operations with the goal to combine workflow and database provenance. However, their approach has the disadvantage that it models the provenance of a tuple as old versions of this tuple and does not model additional provenance dependencies to other input tuple versions. For example, consider an update which inserts tuples from a relation R into a relation S (`INSERT INTO S (SELECT * FROM R)`). No previous version exists for tuples inserted into relation S, because temporal databases do not track dependencies across relations. We reenact update operations to unearth such additional dependencies and compute the provenance of transactions. Our transaction reenactment approach produces a query $q(T)$ which reenacts the updates executed by a transaction T. To compute the provenance of transaction T, we rewrite $q(T)$ using rules designed to compute the provenance of queries. One important advantage of this approach is that we only need rewrite rules for computing the provenance of queries. How to implement such rules is relatively well understood [40]. We compute the provenance of a transaction T as follows.

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

Translate updates. We transform each SQL statement u_i into an AGM reenact-

ment query $q(u_i)$. Assume that statement u_i did update relation R_i. Query $q(u_i)$, if evaluated over the state of relation R_i as of the time when u_i was originally executed, returns the updated content of relation R_i. We use the database backend's time travel features to access this version of R_i.

Construct Reenactment Query. The individual update reenactment queries are then merged into a global reenactment query $q(T)$ simulating the whole transaction. We need to reconstruct the input for each update to correctly reenact the transaction. Different concurrency control mechanisms enforce different visibility rules for concurrent modifications and, thus, each concurrency control mechanism requires a different merge process. For example, updates of a transaction T running under snapshot isolation [13] only see modifications by transactions that did commit before T started and modifications by previous updates of T.

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

We present the whole process of capturing provenance for transactions using reenactment based on two examples in Section B.

6.3 Heuristic and Cost-based Optimization

Previous projects using query rewrites, e.g., for provenance [40] or compiling non-relational languages into SQL (e.g., Pathfinder [43]), have demonstrated that queries produced by such rewrites often contain atypical access patterns and operator sequences (e.g., large number of unions over subqueries accessing the same input and long chains of operators). Such queries "confuse" even sophisticated DBMS optimizers.

We propose to build a general purpose heuristic optimizer for our AGM model inspired by Grust et al. [43]. Note that the goal of such an optimizer is not to compete with mature database optimizers, but rather to transform the input query into a form that can be successfully optimized by the database optimizer. For example, we may want to cluster joins in the query or remove redundant duplicate removal operators and analytical functions. Since all rewrite operations in our system operate on the AGM representation of queries, such an optimizer would benefit all provenance computations. The optimizer will also be used to decide which rewrites to apply if multiple equivalent rewrites are available. For the reasons outlined above and confirmed by our experience on rewriting nested subqueries in the Perm project [38], we cannot rely on the database optimizer to be able to detect these alternatives. Finally, we do not have to rely on purely heuristic optimization. For example, we could use the DBMS optimizer to decide between alternative rewrite options at critical points in the optimization process as long as the underlying database system supports inspection of query plans (standard database systems do support this).

One application domain for GProM are systems such Hive, Pig, Shark, Tenzing, Asterix, and many others which provide SQL or SQL-like query capabilities for BigData workloads. Query optimization in these systems is currently rather limited. For example, Hive only applies a set of heuristic plan rewrites. This is likely to change over time while these systems are becoming more mature. Until then, however, we expect our optimizer to be quite effective. The example shown in Appendix B.2 discusses some simple heuristic optimizations.

6.4 Database Independence

One of the goals of GProM is to support a wide variety of database backends. To support this goal, we encapsulate database-specific functionality in pluggable modules. Query rewrites related to provenance computation, optimization, provenance

translation, and update and transaction reenactment all operate on our AGM model. These rewrites do not need to be modified to support additional database backends. What needs to be adapted are 1) the parser (each database vendor supports a different SQL dialect), 2) the SQL code generator (again for dialect compliance), and 3) metadata access, 4) audit log access, and 5) time travel activation. Furthermore, we may want to tweak the optimization for each individual system.

6.5 Summary

We present GProM, a database-independent middleware for computing the provenance of queries, updates, and transactions. Our approach takes query rewrite techniques to the next level by using them for provenance computation, transaction reenactment, provenance translation, provenance storage, and optimization. Notably, this feature only requires the underlying database to support time travel and to provide an audit log. Furthermore, this feature is independent of what type of provenance is computed and, aside from the runtime and storage overhead caused by maintaining the audit log and data required to support time travel, results in no overhead for normal database operations. Our prototype implementation using Oracle demonstrates the feasibility of our approach.

CHAPTER 7
OPTIMIZATIONS

We discuss several optimizations including alternative ways of implementing reenactment queries and techniques for filtering unrelated data from the provenance computation early on.

7.1 Reducing MV to Standard Relational Semantics

We encode a normalized MV-semiring annotation of a tuple as a set of tuples - one for each summand. We add provenance attributes to the result schema to store tuples in the provenance (variables in MV-semiring expressions) and version annotations for each summand. For instance, a tuple t annotated with $x + y$ would be encoded as two tuples encoding the summand x and y, respectively. When computing the provenance of a Transaction T, initial annotations for a relation R are created to represent variables in annotations. We access the snapshot of R as of the start of Transaction T and create annotations by duplicating attribute values using projection. Note that the relational encoding of annotations produced by this step corresponds to the result of applying the filtering step 2 in Sec. 4.8 to $R[Start(T)]$. Here we assume a standard SI based implementation of time travel that allows us to access a *snapshot* R_ν of relation R containing all committed tuple versions valid at ν. Furthermore, we expect a snapshot to store the following information for each tuple version: 1) the transaction that created the tuple version (attribute Xid) and 2) a unique tuple identifier (attribute Id). We instrument the remaining operators to propagate annotations from their inputs. Consider a transaction $T = (u_1, \ldots, u_n, c)$. We apply a selection $\mathcal{U}_1 \vee \ldots \vee \mathcal{U}_n$ to the result to implement filtering step 1 that removes tuples that were not affected by Transaction T (see Sec. 4.8). The details of

our encoding and instrumentation are presented in our technical report [6].

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

Example 21. *Consider a Transaction T with a single update:*

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

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

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

7.2 Prefiltering Provenance

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

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

Example 22. *Applying this optimization to the reenactment query from the previous example yields the query below.*

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

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

query in the transaction.

Example 23. *For the example query from this section we get:*

```
SELECT cust, type, (CASE  WHEN (typ = 'Savings')
                    THEN bal + 100 ELSE bal END) AS bal
FROM (SELECT Id AS rid, cust, type, bal
      FROM Account AS OF Start(T))
      NATURAL JOIN
     (SELECT Id AS rid
      FROM Account AS OF End(T)+1 WHERE Xid = T)
```

.

7.3 Reducing Relation Accesses

We review an optimization for RC-SI reenactment as presented in [7]. By applying this optimization, we can achieve performance comparable to SI reenactment. We would like reenactment queries for RC-SI to be defined recursively without requiring to recalculate the right mix of tuple versions from transaction T and from concurrent transactions after each update. To this end we introduce the version filter operator, that filters out summands k from an annotation based on the version encoded in the outermost version annotation of k. The filter condition θ of a version filter operator is expressed using a pseudo attribute V representing the ν encoded in version annotations. We use this operator to filter summands from annotations based on the version annotations they are wrapped in.

Version Filter Operator. The version filter operator removes summands from an annotation based on the time ν in their outermost version annotation. Let θ be a condition over pseudo attribute V. Given a summand $k = X^{i}_{T,\nu}(k')$ such a condition

is evaluated by replacing V with ν in θ. The version filter operator using such a condition θ is defined as:

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

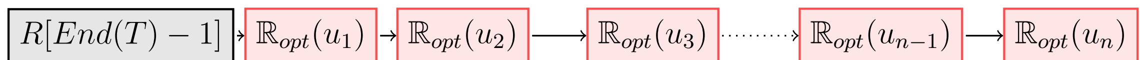
For example, we could use $\gamma_{V<11}(R)$ to filter out summands from annotations of tuples from a relation R that were added after time 10. In contrast to regular selection, a version filter's condition is evaluated over the individual summands in an annotation.

Our optimized reenactment approach for RC-SI is based on the following observation. Consider a tuple t updated by Transaction T and let $u \in T$ be the first update of Transaction T that modified this tuple. Let t' denote the version of tuple t valid before u. Given the RC-SI semantics, t' is obviously present in $R[\nu(u)]$ and was produced by a transaction that committed before $\nu(u)$. Importantly, t' is guaranteed to be in $R[End(T)]$, i.e, the version of R immediately before the commit of Transaction T. To see why this is the case recall that T would have obtained a write-lock on this tuple to be able to update t' to t and this write-lock is held until transaction commit. Thus, it is guaranteed that no other transaction would have been able to update t' before the commit of T. Based on this observation, we can use $R[End(T)]$ as an input to the reenactment query as long as we ensure that the reenactment queries for other updates of T executed before u ignore t'. We achieve this using the version filter operator to filter out tuple versions that were not visible to an update u'. It is applied in the input of the part of the transaction reenactment query corresponding to the update u'. In the optimized reenactment query, the initial input of reenactment is $R[End(T)]$ instead of $R[Start(T)]$. Furthermore, the update reenactment queries are modified as shown below. An optimized reenactment query $\mathbb{R}_{opt}(u)$ for update u passes on unmodified versions of tuples that are not visible to update u. We use $\mathbb{R}_{opt}(T)$ to denote the optimized transaction reenactment query. In the formulas

shown below, R denotes the result of the reenactment query for the previous update or $R[End(T) - 1]$ (in case the update is the first update of the transaction). Note that this optimization is only applicable if the inserts in the transaction do not access the relation that is modified by the updates and deletes of the transaction. That is because the query of an insert may read tuple version that are not in $D[End(T)]$. Hence, we only apply this optimization if the inserts of Transaction T use the `VALUES` clause (the singleton operator $\{t \to k\}$ as defined in Section 4).

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

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



For each insert using the `VALUES` clause a new tuple will be added to the

relation R using `UNION`.

7.4 Summary

We presented optimizations specific to reenactment queries. These optimizations transforms a SQL query which implements provenance capture into a query that can be successfully optimized by the backend database system. Experiment result in Chapter 10 shows the effectiveness of our optimization techniques.

CHAPTER 8
HISTORICAL WHAT-IF QUERIES

What-if queries predict how hypothetical updates to a database would affect the result of an analysis. For instance, *"how would a 1% increase in sales affect the company's revenue this year?"* However, it may not be obvious how such an update could be realized. We introduce *historical what-if queries* as a more practical model that determines the effect of a hypothetical change to past operations of a business (transaction executions). For example, *"how would our revenue be affected if we would have charged an additional $6 for shipping?"* We argue that such queries are easier to formulate as they are based on changes to past actions instead of to the results of these actions as in traditional what-if queries. Furthermore, such queries may lead to more actionable insights as their results can be used to inform future actions, e.g., increasing shipping fees. We develop efficient techniques for answering historical what-if queries and implement these techniques in *Mahif* (a Middleware for Answering Historical what-IF queries). Mahif uses *reenactment*, a replay technique for transactional histories, to determine how a modified history affects the current database state. We optimize this process using program and data slicing techniques that determine which updates and data can be excluded from reenactment without affecting the result. Our experimental evaluation demonstrates the effectiveness of Mahif and of these optimizations.

8.1 Motivational Example

What-if analysis [12, 30] determines how a hypothetical update to a database instance affects the result of a query. Consider the following what-if query: *"How would a 10% increase in sales affect our companys revenue this year?"* While the

result of this query can help an analyst to understand how revenue is affected by sales, its practical utility is limited because it does not provide any insight in how this increase in sales could have been achieved in the first place. We argue that this problem is not specific to this example, but rather is a fundamental issue with classical what-if analysis since the hypothetical update to the database is part of the input. We propose *historical what-if queries*, a novel type of what-if queries where the user postulates a hypothetical change to the transactional history of the database. For example, consider the case of an online retailer that has implemented a promotion waiving shipping fees for orders over $50. This policy was implemented though the update shown below (assume the database schema from Figure 8.1):

```
UPDATE Order SET ShippingFee=0 WHERE Price>=50;
```

A user may be interesting in knowing how a larger order price threshold, say $60, would have affected revenue. This question corresponds to the following historical what-if query: *"How would revenue be affected if we would have used a threshold of $60 instead of $50 for waiving shipping fees?"*. This query postulates changing the history by replacing the `WHERE` clause of the update shown above with `WHERE Price >= 60`. By evaluating the effect of a change to a past action (an update or transaction) instead of to the current state of the database as in classical what-if analytics, the answer to a historical what-if query can be used to decide future actions. For example, if the answer to the query shown above shows that revenue is increased significantly by using the $60 cutoff, then it may be worth considering this higher threshold for future promotions.

Example 24. *As a more extensive example, consider the order table for a online retailer shown in Figure 8.1. The retailer did introduce a new policy for determining shipping fees which was implemented by updating the shipping fees for existing orders as follows: the fee for orders with price equal or greater than $40 was set to $0, orders*

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	5	o_1
12	Alex	UK	40	5	o_2
13	Jack	US	60	3	o_3
14	Mark	US	30	4	o_4

Figure 8.1. Running example database instance

T	U	SQL	Time
T_1	u_1	`UPDATE Order SET ShippingFee=0 WHERE Price>=40 ;`	20
T_1	${u_1}'$	`UPDATE Order SET ShippingFee=0 WHERE Price>=50 ;`	20
T_1	c_1	`COMMIT;`	21
T_2	u_2	`UPDATE Order SET ShippingFee=ShippingFee+5 WHERE Country='UK'AND Price <=100 ;`	22
T_2	c_2	`COMMIT;`	23
T_3	u_3	`UPDATE Order SET ShippingFee=ShippingFee-2 WHERE Price <=30 AND ShippingFee>=10 ;`	24
T_3	c_3	`COMMIT;`	25

Figure 8.2. Transactional history implementing the new ShippingFee policy and a hypothetical change the policy (update ${u_1}'$ replaces u_1)

of less than or equal to \$100 with a destination in the UK were charged an additional \$5 shipping fee, and orders with a price equal or less than \$30 and shipping fee equal or more than \$10 received a \$2 discount for their shipping fee. Figure 8.2 shows three transactions T_1, T_2 and T_3 that implement this policy which were executed sequentially in the order shown here resulting in the database state shown in Figure 8.3. Ignore update u_1' shown with red background for now. Bob, a manager at the retailer, requests a report on how raising the minimum order price from \$40 to \$50 for reducing the shipping fee (Transaction T_1) would impact the total revenue per country. The revenue

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	8	o'_1
12	Alex	UK	40	5	o''_2
13	Jack	US	60	0	o'_3
14	Mark	US	30	4	o_4

Figure 8.3. Database state after executing the original history

Query result

Total	Country	
73	UK	$o'_1 + o''_2$
94	US	$o'_3 + o_4$

Figure 8.4. Q_{total} result

per country can be computed using a query Q_{total}:

```
SELECT sum(Price+ShippingFee) AS Total, Country FROM Order
GROUP BY Country;
```

Bob's request can be expressed as a historical what-if query which modifies the update u_1 in Transaction T_1 to the update $u_1{}'$ that is highlighted in red in Figure 8.2. Figure 8.5 shows the new state of the order database after executing the modified transactional history over the database from Figure 8.1. Figures 8.4 shows the original and result of Q_{total}. The result of this query over the database produced by the hypothetical history is shown in Figure 8.6.

In this work, we present and formalize historical what-if queries, a novel type of hypothetical analysis. In this work we focus on deterministic updates (given the same

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	8	o'_1
12	Alex	UK	40	10	o'_2
13	Jack	US	60	0	o'_3
14	Mark	US	30	4	o_4

Figure 8.5. Database state based on the historical what-if query

Query result

Total	Country	
78	UK	$o'_1 + o'_2$
94	US	$o'_3 + o_4$

Figure 8.6. Q_{total} result

input, multiple executions of an update are guaranteed to return the same result). Our approach is based on reenactment [10, 7], a declarative techniques for replaying past update operations or even transactional histories using temporal queries running on any DBMS supporting for *time travel* (e.g., Oracle, SQLsever, DB2). Time travel can be implemented using triggers for systems that do not support this functionality natively.

Naive method is shown in Figure 8.7 for answering a historical what-if query. The naive method creates a copy of the database before the execution time of the modified update and then executes the modified history on it. Then it execute the requested query on the new version and the original database version. Finally, it computes the difference between results of the previous step as the answer to the historical what-if query.

Disadvantages of the naive method are noticeable. The naive method requires additional storage and evaluating the modified history results in a large amount of write I/O. Moreover, most database systems log modifications of update operations in a write-ahead logging (WAL) for recovery purposes which costs additional overhead and it is not feasible for large amount of update operations in histories as replaying the whole history would be expensive.

Proposed method is presented in Figure 8.8. In order to overcome the naive method limitations, we proposed Mahif as a middleware that answer historical what-if queries using reenactment [10, 7, 9] which is a declarative replay technique of transnational history by a temporal query. In this method, we do not need to copy the database and we just use time travel to get the database state before the modification and use it as an input into the reenactment query which simulate the effect of updates in the modified history. The result of the reenactment is same as the the database state which can be created by executing the modified history. Similar to the naive method, we run the requested query on the result of the reenactment and original database state and then compute their difference as the answer to the historical what-if query. One advantage of using reenactment is it does not need to copy the database and another is it avoids logging overhead caused by running updates. Also, it enables optimization methods such as program slicing that minimize the number of updates to reenact and data slicing which reduces input data into the reenactment. We propose program slicing method that statically analyzes potential dependencies among updates in a history. This method is similar to ideas from programming languages, i.e., symbolic execution [18, 58]. We simulate updates in history by symbolic execution to determine updates that are might be dependent on the modifications and changes in a historical what-if query. Dependent updates may have different input or output whereas independent have same input and result in the original and modified history. In order to improve performance of our approach, we just consider dependent

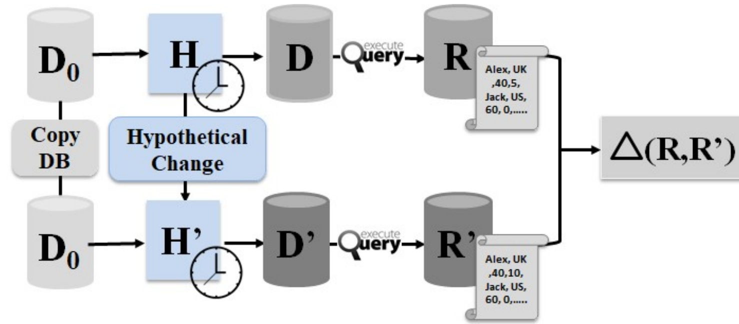


Figure 8.7. The Naive Method

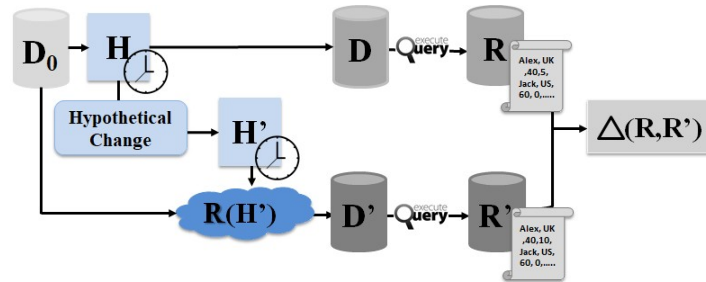


Figure 8.8. The Proposed Method

updates in reenactment as they affect the final result of the historical what-if query. We also provide data slicing optimization to just process data that is necessary for computing the result of historical what-if query and avoiding processing the whole data in a database.

The main contributions of this work are:

- We present our solution and algorithms for answering historical what-if queries.
- We discuss how we can apply *reenactment*, a technique for replaying a transnational history using queries to avoid additional overhead of re-executing update in a transnational history.
- We present an optimization technique *programming slicing* that reduces the number of update operations in a reenactment query construction.
- We provide the *symbolic execution* and our mixed-integer linear programming

(MILP) based solution that encodes the effect of update operations for implementing programming slicing which detects dependent updates to exclude from reenactment.

- We discuss an additional optimization technique *data slicing* which filters input data into the reenactment queries.
- Our experiments demonstrate the effectiveness and efficiency of our methods over real-world and synthetic datasets.

8.2 Historical What-if Queries

We now formally define historical what-if queries and the problem addressed in this work: computing answers to historical what-if queries. We first define a modification, a modified history, and a database delta. Then we define the historical what-if query, and the answer to such query based on these definitions.

Definition 26. *A **modification** $m = u \leftarrow u'$ denotes replacing the update u with the update u'. We use $\mathcal{M}$ to denote a sequence of modifications.*

Definition 27. *A **modified history** $H[\mathcal{M}]$ denotes a result of applying a sequence of modifications $\mathcal{M}$ to the history H.*

Using previously defined applying history to a database, obviously $H[\mathcal{M}](D)$ shows the result of evaluating the modified history $H[\mathcal{M}]$ over a database instance D.

To answer a historical what-if query, we need to compute the difference between the current database and the database that produced by the historical what-if query. We define their differences based on the state of their tuples.

Definition 28. *A **database delta** $\Delta(D, D')$ represents difference between two database states D and D'. An annotated tuple $+t$ denotes that t exists in D', but not in*

D, and $-t$ denotes that t is in D, but not in D'.

$$\Delta(D, D') = \{+t \mid t \notin D \wedge t \in D'\} \cup \{-t \mid t \in D \wedge t \notin D'\}$$

We define a historical what-if query and an answer to such query based on these definitions. We need to compute the delta and differences between $H(D)$ and $H[\mathcal{M}](D)$, and then run the query over this delta.

Definition 29. *A **historical what-if query** $\mathcal{H}$ is a tuple $(H, D, \mathcal{M}, Q)$ where H is a transactional history run over a database instance D, $\mathcal{M}$ denotes a sequence of modifications to H as introduced above, and Q is a query over the schema of D. The answer to $\mathcal{H}$ is defined as:*

$$\Delta(Q, H(D), H[\mathcal{M}](D))$$

Example 25. *Let D and H be database shown in Figure 8.1 and history shown in Figure 8.3, respectively. Consider the single modification $m_1 = u_1 \leftarrow u_1'$ using u_1 and u_1' as shown in Figure 8.2 which implements the policy change of increasing the minimum price for waving shipping fees. Bob's historical what-if query from this example can be expressed as $\mathcal{H} = (H, D, m_1, Q_{total})$ in our framework. Evaluating $H[m_1]$ results in a modified database instance as shown in Figure 8.5. For convenience we have highlighted modified tuple values. The result of evaluating the query Q_{total} and of computing the delta as the answer for the what-if query is shown in Figure 8.9. Alex's order receives additional \$5 shipping fee under this new policy because he is not eligible for the new free shipping fee policy ($u_1{}'$). \$50. Bob can now decide whether to change the shipping fee policy based on this result.*

8.3 Overview of Our Approach

To formulate and evaluate historical what-if queries we have to have access to the a log of SQL commands executed in the past (audit log) and query access

Total	Country
5	UK
0	US

Figure 8.9. $\Delta(Q, H(D), H[\mathcal{M}](D))$

Algorithm 1 Naive Solution

1: **procedure** WhatIf(H, D,D_0 ,Q, $\mathcal{M}$)
2: $D'_0 \leftarrow$ Copy(D_0)
3: $D' \leftarrow H[\mathcal{M}](D'_0)$
4: **return** SymmetricDiff(Q, D, D')
5: **end procedure**

to the past states of tables (time travel). For databases that support these featuree (e.g., Oracle, DB2, and MSSQL), we can use an audit log that stores a history of SQL commands executed in the past and for each command when it was executed and as part of which transaction. Otherwise, we can use triggers to capture SQL commands or exploit other logging mechanisms if available. We can limit the history of SQL commands that were executed after the first modification in $\mathcal{M}$ since that other commands in the past are not affected by the $\mathcal{M}$ and they do not have any effect on the final result of a historical what-if query. To answer a historical what-if query, we need to compute $H[\mathcal{M}](D)$ and $H(D)$, then compute the answer of query Q over these database versions, and finally compute the symmetric difference between $Q(H[\mathcal{M}](D))$ and $Q(H(D))$.

Algorithm 1 shows the naive method. The first line creates D_0 as a copy of original database D as of the start time of $\mathcal{M}$ using time travel. The next line shows the modified history $H[\mathcal{M}]$ is re-executed over the copy that result in new database state D'. In the last step, the query Q over both D and D' databases are executed and

Algorithm 2 Answering Historical What-if Query

1: **procedure** WhatIf(H, D, Q, $\mathcal{M}$)

2: $updateList \leftarrow NULL$

3: **for all** $u \in \mathcal{M}$ **do**

4: $updateList \leftarrow$ ApplyProgrammingSlicing$(H, D, u) \cup depList$

5: **end for**

6: $\mathbb{R}(H) \leftarrow$ GenReenactmentQuery$(H, depList)$

7: $\mathbb{R}(H)^* \leftarrow$ ApplyDataSlicing$(\mathcal{M}, \mathbb{R}(H))$

8: $\mathbb{R}(H[\mathcal{M}]) \leftarrow$ GenReenactmentQuery$(H[\mathcal{M}], depList)$

9: $\mathbb{R}(H[\mathcal{M}])^* \leftarrow$ ApplyDataSlicing$(\mathcal{M}, \mathbb{R}(H[\mathcal{M}]))$

10: **return** SymmetricDiff$(Q, \mathbb{R}(H)^*(D), \mathbb{R}(H[\mathcal{M}])^*(D))$

11: **end procedure**

their symmetric difference is returned as a what-if query result. Considering Example 24, Figure 8.2 shows H and $H[\mathcal{M}]$. Figure 8.1 shows D_0, Figure 8.3 and Figure 8.5 present D and D' respectively. The result of $\Delta(Q_{total}, D, D')$ is shown in Figure 8.9. We now give a high-level overview of our Algorithm 2 for answering a historical what-if query $\mathcal{H} = (H, D, Q, \mathcal{M})$. We first compute *updateList*, an overestimate of the set of updates using the programming slicing approach described in Section 8.6. More detail is provide in Algorithm 3. We then generate reenactment queries for H and $H[\mathcal{M}]$ restricted to *updateList* and add selection conditions to implement the data slicing optimization (Section 8.5). Finally, function `SymmetricDiff` takes a query Q and the two reenactment queries as input, computes $\Delta(H(D), H[\mathcal{M}](D))$ using the reenactment queries, and incrementally maintains $Q(H(D))$ to compute the answer to $\mathcal{H}$.

In the following we introduce three optimizations to the naive approach:

- **Incremental Maintenance** - we present how to use incremental view main-

tenance techniques to improve the performance of our process. We discuss that we do not need to compute the whole new database state and we can reproduce only part of the database that might be affected by the modifications in the historical what-if query.

- **Reenactment** - we use *reenactment* [7, 9], our declarative replay technique to avoid additional storage for copying the database and overhead of re-executing all update operations in the history. It also enable us to apply additional optimization methods such as programming slicing and data slicing in our approach.
- **Data Slicing** - we filter the input data to the reenactment query and exclude it from replaying that are not modified by $\mathcal{M}$ in a historical what-if query. We show in our experiments how data slicing improve performance significantly.
- **Program Slicing** - we describe how to determine a set of update operations that are necessary for replaying and reenactment to avoid considering all of update operations in a history in our approach. In the experiment section, we show the cost of using programming slicing is compromised by the cost of creating a copy of database and re-running modified history.

8.4 Incremental Maintenance and Reenactment

The naive approach evaluates the query Q of a historical what-if query $\mathcal{H}$ over the current database state $H(D)$ and over $H[\mathcal{M}](D)$ and then computes the delta of the results. In our approach, we compute $\Delta(Q, H(D), H[\mathcal{M}](D))$ as an incremental view maintenance problem. We compute $Q(H(D))$ and incrementally maintain this result based on $\Delta(H(D), H[\mathcal{M}](D))$ using an incremental view maintenance technique (e.g., [54]). We produce the part of result that might be changed by the historical what-if query instead of reproducing the whole database state. The part of database that is not changed by the historical what-if query would

be same in $H(D)$ and $H[\mathcal{M}](D)$ and it does not help users to understand the effect of a hypothetical change to past operations of their business. As a result, we just need to compute $\Delta(Q, H(D), H[\mathcal{M}](D))$. We do not need to involve Q in our computation and we can apply it on top of the $\Delta(H(D), H[\mathcal{M}](D))$ result in the final step. As $Q(\Delta(H(D), H[\mathcal{M}](D)))$ would have same output result as $\Delta(Q(H(D)), Q(H[\mathcal{M}](D)))$, our main focus is computing $\Delta(H(D), H[\mathcal{M}](D))$. In the rest of this work, we mostly use $\Delta(H(D), H[\mathcal{M}](D))$ as an answer to the historical what-if query and we ignore Q as it can be applied later without affecting a final result of historical what-if query.
We apply *reenactment*, a declarative replay technique for transactions that we have introduced in previous Chapter 5, to compute $H[\mathcal{M}](D)$. First we need to create a reenactment query for each update statement of the transaction in the history and then merge these individual reenactment queries to get the reenactment query for the transaction and then the whole history.

Given a transactional history H, reenactment generates a *reenactment query* $\mathbb{R}(H)$ which is equivalent to the history under standard bag and set semantics. In order to compute $\mathbb{R}(H)$, first we generate a reenactment query for each update statement $\mathbb{R}(u)$ in the history to simulate the effects of them and then we merge these reenactment queries. For instance, $\mathbb{R}(u)$ for an update statement u is defined as the union of modified versions of all tuples that fulfill the θ function of the update and the version of tuples before execution of the update that do not fulfill the θ function. The merging step for the sequence of updates in the history is computed recursively. For example, $\mathbb{R}(u_i)$ is written on top of the the result of the reenactment of the previous update in the history ($\mathbb{R}(u_{i+1})$). Importantly, reenactment queries can be expressed in standard SQL using time travel and can be used to replay only a part of a history. Thus, we can run the query $\mathbb{R}(H[\mathcal{M}])$ over D to generate $H[\mathcal{M}](D)$. Compared to the naive approach this has the advantage that there is no need to copy the data-

base and we avoid the logging overhead caused by running update operations. More importantly, it enables the data slicing and program slicing optimizations that we discuss next.

Example 26. *Considering the Example 24 where cu, co, p, sf stands for Customer, Country,Price,ShippingFee attributes and O stands for the relation Order. The $\mathbb{R}^O(H)$ for the history in Figure 8.2 is:*

$$\mathbb{R}^O(H) = \mathbb{R}^O(u_3)$$

$$\mathbb{R}^O(u_3) = \Pi_{ID,cu,co,p,(sf-2)\rightarrow sf}(\sigma_{p<=30\wedge sf>=10}(\mathbb{R}^O(u_2))) \cup \sigma_{\neg(p<=30\wedge sf>=10)}(\mathbb{R}^O(u_2))$$

$$\mathbb{R}^O(u_2) = \Pi_{ID,cu,co,p,(sf+5)\rightarrow sf}(\sigma_{co='UK'\wedge p<=100}(\mathbb{R}^O(u_1))) \cup \sigma_{\neg(co='UK'\wedge p<=100)}(\mathbb{R}^O(u_1))$$

$$\mathbb{R}^O(u_1) = \Pi_{ID,cu,co,p,0\rightarrow sf}(\sigma_{p>=40}(O[20])) \cup \sigma_{\neg(p>=40)}(O[20])$$

$\mathbb{R}^O(H[\mathcal{M}])$ would be similar, except in $\mathbb{R}^O(u_1')$ instead of $p >= 40$ in the selection condition, there would be $p >= 50$.
The result of $\Delta(Q, \mathbb{R}^O(H)(D), \mathbb{R}^O(H[\mathcal{M}])(D))$ where Q is the revenue per country, would be:

$$\Delta(Q, \mathbb{R}^O(H)(D), \mathbb{R}^O(H[\mathcal{M}])(D)) = \rho_{sum(p+sf)\rightarrow total}(\gamma_{co,sum(p+sf)}(symDiff))$$

$$symDiff \leftarrow (\mathbb{R}^O(H)(D) - \mathbb{R}^O(H[\mathcal{M}])(D)) \cup (\mathbb{R}^O(H[\mathcal{M}])(D) - \mathbb{R}^O(H)(D))$$

8.5 Data Slicing

Intuitively, any difference between $H(D)$ and $H[\mathcal{M}](D)$ has to be caused directly or indirectly by a difference between H and $H[\mathcal{M}]$. For simplicity of exposition consider a single modification $m = u \leftarrow u'$. For a tuple t to appear in

$\Delta(H(D), H[\mathcal{M}](D))$, it has to be affected by either u or u' or both (but in different ways). Given an expressive enough provenance model that allows us to determine based on the provenance of a tuple t which updates of a history affected it (e.g., our MV-semiring model [7]), we can filter the inputs to the delta computation based on their provenance - only retaining tuples that have either u or u' in their provenance. However, this would require us to compute both $H(D)$ and $H[\mathcal{M}](D)$ with provenance tracking and the overhead associated with that may easily outweigh the advantage of reducing the input size of the delta computation. Since only tuples that match the condition of an update operation (the update's `WHERE` clause) can be affected by the update, a conservative overestimation of $\Delta(H(D), H[\mathcal{M}](D))$ is the set of tuples that are derived from tuples affected by u in the original history or u' in the modified history. Thus, the original version of these tuples in D have to either match the condition of u ($\theta(u)$) or the condition of u' ($\theta(u')$). This means we can filter input to the reenactment queries using a condition $\theta(u) \vee \theta(u')$.

Theorem 13. *Consider a historical what-if query $\mathcal{H} = (H, D, \mathcal{M}, Q)$ with a single modification whereas $\mathcal{M} = m$ and $m = u \leftarrow u'$. Let $\theta(u)$ denote the condition of an update u and $R \leftarrow \sigma_{\theta(u) \vee \theta(u')}(D)$.*
$\Delta(Q, \mathbb{R}(H)(D), \mathbb{R}(H[\mathcal{M}])(D)) = \Delta(Q, \mathbb{R}(H)(R), \mathbb{R}(H[\mathcal{M}])(R))$

Proof. We now prove the theorem by induction by considering direct and indirect changes of tuples by updates in the history. Assume a historical what-if query $\mathcal{H} = (H, D, \mathcal{M}, Q)$ with a single modification $m = u_1 \leftarrow u_1'$, and the database D. Induction Start: Consider $H = \{u_1, u_2\}$, a single tuple $t_0 \in D$ does not meet the condition $\theta(u_1)$ nor $\theta(u_1')$. Assume t_0 were modified by u_2 and created new version t_1 where $t_1 = u_2(t_0)$. Since this tuple was not changed by the historical what-if query, t_1 is same in both $\mathbb{R}(H)(D)$ and $\mathbb{R}(H[\mathcal{M}])(D)$. So, we can remove the tuple t_1 from the delta $\Delta(Q, \mathbb{R}(H)(D), \mathbb{R}(H[\mathcal{M}])(D))$ and the result of historical what-if query would

be same. As a result, we can filter such tuples from input to reenactment queries. Induction Step: Assume $H = \{u_1, \ldots, u_n\}$ and t_0 does not meet the condition $\theta(u_1)$ nor $\theta(u_1')$ but it was modified by other updates in the history $t_{n-1} = u_n(\ldots(u_2(t_0)))$. We can remove the tuple t_{n-1} from the delta $\Delta(Q, \mathbb{R}(H)(D), \mathbb{R}(H[\mathcal{M}])(D))$ and the result of historical what-if query would be same. We need to show that the same holds for $H = \{u_1, \ldots, u_n, u_{n+1}\}$ whereas $t_n = u_{n+1}(t_{n-1})$. Since t_{n-1} is an input into the u_{n+1} is exactly same in $\mathbb{R}(H)(D)$ and $\mathbb{R}(H[\mathcal{M}])(D)$, t_n as the output of u_{n+1} would be also same in both reenactment queries. So, we can remove t_n from the delta $\Delta(Q, \mathbb{R}(H)(D), \mathbb{R}(H[\mathcal{M}])(D))$ without any changes to the result of historical what-if query. □

For multiple modifications ($\mathcal{M}$), we can filter input tuples into their related reenactment queries by applying disjunction of update conditions for each $m \in \mathcal{M}$ and then merge those reenactment queries.

Example 27. *To compute $\Delta(Q, \mathbb{R}^O(H)(D), \mathbb{R}^O(H[\mathcal{M}])(D))$ for the Example 26, we first construct reenactment queries for updates u_1, u_2, u_3 and u_1', u_2, u_3. Based on the conditions of u_1 and u_1', we add a condition $\sigma_{p>=40 \vee p>=50}$ on the top of these queries. In our example, this would exclude tuples o_1 and o_4 from reenactment since any tuple in $\mathbb{R}^O(H)(D)$ and $\mathbb{R}^O(H[\mathcal{M}])(D)$ derived from these tuples will occur in the result of both histories and, thus, not be part of the delta. So, we need to compute $\Delta(Q, \mathbb{R}^O(H)(R), \mathbb{R}^O(H[\mathcal{M}])(R))$ where $R \leftarrow \sigma_{p>=40 \vee p>=50}(D)$. for answering the historical what-if query.*

We can apply data slicing techniques for delete statements. Data slicing for delete statements are different from update statements as they remove some tuples from the database and we do not need to consider tuples that are deleted by both the original and modified statements. If a tuple is removed by both original and modified

statements, it would not be in the result of $\mathbb{R}(H)$ nor $\mathbb{R}(H[\mathcal{M}])(D)$ reenactment queries, so we can filter them in computing the delta without any changes to the result of historical what-if query. To compute the difference between the result of $\mathbb{R}(H)(D)$ and $\mathbb{R}(H[\mathcal{M}])(D)$, we need tuples that are deleted by the original delete statement but not by modified one or vice versa.

Theorem 14. *For a historical what-if query $\mathcal{H} = (H, D, \mathcal{M}, Q)$, consider a single modification $m = u \leftarrow u'$ where u and u' are delete statements. $\theta(u)$ denote the condition of the delete statement u and where $R \leftarrow \sigma_{\theta(u) \wedge \neg\theta(u')}(D)$ and $R' \leftarrow \sigma_{\neg\theta(u) \wedge \theta(u')}(D)$.*
$\Delta(Q, \mathbb{R}(H)(D), \mathbb{R}(H[\mathcal{M}])(D)) = \Delta(Q, \mathbb{R}(H)(R), \mathbb{R}(H[\mathcal{M}])(R'))$

The proof is similar to the proof we provide for update statements. Another data slicing technique can be applied for other delete statement in the history that are not modified by the historical what-if query. Since this kind of statements remove tuples from the database, those tuples can be filtered out from the input tuples for next updates in the history.

8.6 Program Slicing

In addition to data slicing, we can also optimize the process of answering a historical what-if query $\mathcal{H} = (H, D, \mathcal{M}, Q)$ by excluding updates from reenactment if their existence has no affect on the answer of $\mathcal{H}$. This is akin to *program slicing* [24, 71] which is a technique developed by the PL community to determine a slice (a subset of the statements of a program) that is sufficient for computing the values of variables at a given set of locations in the program. Analog, we define slices of histories wrt. historical what-if queries. We use $\mathsf{Del}(H, u)$ to denote the history that is the result of removing update u from history H.

Definition 30 (History Slice)**.** *Let $\mathcal{H} = (H, D, \mathcal{M}, Q)$ be a historical-whatif query. We call any pair $(H_{\mathsf{slice}}, H[\mathcal{M}]_{\mathsf{slice}})$ where $H_{\mathsf{slice}} \subseteq H$ and $H[\mathcal{M}]_{\mathsf{slice}} \subseteq H[\mathcal{M}]$ a* slice

for $\mathcal{H}$ if it fulfills the condition shown below.

$$\begin{aligned} &\Delta(H(D), H[\mathcal{M}](D)) \\ =&\Delta(H_{\mathsf{slice}}(D), H[\mathcal{M}]_{\mathsf{slice}}(D)) \end{aligned} \qquad \textbf{(sufficiency)}$$

That is, a slice for a historical what-if query $\mathcal{H}$ consists of subsets of H and $H[\mathcal{M}]$ that can be substituted for the original history and modified history when evaluating the historical what-if query without changing its result (the *sufficiency* condition in the definition above). Ideally, we would like slices to be *minimal*, i.e., removing any update from H_{slice} or $H[\mathcal{M}]_{\mathsf{slice}}$ breaks sufficiency. Note that there may exist more than one minimal slice for a query $\mathcal{H}$, because the exclusion of one update may prevent us from excluding another update. History slices allow us to optimize the evaluation of a historical what-if query by excluding updates.

We call any update $u \in H$ that is not modified by $\mathcal{M}$ a *dependent* update if that affects the result of $\mathcal{H}$ and use $\mathsf{Dep}(H, \mathcal{M}, D)$ to denote the set of dependent updates for H, $\mathcal{M}$, and D. Here we only consider deterministic updates, thus, if the input of an update is the same in both histories then so is its output. Intuitively, the input of an update u can only differ if it contains one or more tuples affected by the historical what-if query. Given a history H, a single modification $m = u \leftarrow u'$, and a database instance D, we say an update $u_i \in H$ *depends* on m according to D if there exists a tuple affected by u_i that is also affected by either u or u' directly or indirectly. In order to formally define dependent update, first we define partial history and possible tuple versions.

Definition 31. *A* ***partial history*** *represent a part of an update history ($H_i \subseteq H$), and it includes all updates from the first update to update number i. For the $H = \{u_1, \ldots, u_n\}$, we can consider a partial history $H_i = \{u_1, \ldots, u_i\}$ where $1 \leqslant i \leqslant n$.*

Definition 32. *A* ***possible tuple versions*** *is a set of tuples (T_i) for the historical what-if query $\mathcal{H} = (H, D, \mathcal{M}, Q)$ that exist in the partial history $H_i(D)$ or $H_i[\mathcal{M}](D)$.*

For any $i \in [1, n]$ *we define:*

$$T_i = \{t \mid t \in H_i(D) \vee t \in H_i[\mathcal{M}](D)\}$$

We define $T_0 = D$*. Note that* $T_n = H(D) \vee H[\mathcal{M}](D)$*.*

Definition 33. *A* ***dependent update*** *is an update* $u_i \in H$ *that depends on the modification of historical what-if query (*$\mathcal{M}$*) directly or indirectly.* $u_i(t) = t'$ *indicates the update* u_i *updated a tuple* t *and the result is a new tuple* t'*.* u_i *is a dependent update if:*

$$\mathsf{Dep}(H, \mathcal{M}, D) = \{u_i \mid u_i \in H; \exists t \in T_{i-1}, \exists t' \in T_i \wedge u_i(t) = t' \wedge$$

$$((t \notin H_{i-1}(D) \vee t \notin H_{i-1}[\mathcal{M}](D)) \wedge (t' \notin H_i(D) \vee t' \notin H_i[\mathcal{M}](D)))\}$$

Example 28. *Consider the Example 24 and the history in Figure 8.2. We want to examine whether* u_2 *which modifies the relation Order, is a dependent update for the history and the single modification* $m = u_1 \leftarrow u_1'$*.*

$$\mathsf{Dep}(H, \mathcal{M}, D) = \{u_2\}$$

because:

$$u_2 \in H; \exists o_2 \in T_1, \exists o_2' \in T_2 \wedge u_2(o_2) = o_2' \wedge$$
$$o_2 \notin H_1(Order) \wedge o_2' \notin H_2(Order)$$

$o_2 \notin H_1(Order)$ *as it is modified by* u_1 *but it is not updated by* u_1' *so* $o_2 \in H_1[\mathcal{M}](Order)$*. Therefore,* $o_2' \in H_2[\mathcal{M}](Order)$ *but* $o_2' \notin H_2(Order)$*.*

If an update u_i in the history is not a dependent update, there would not be any set of tuples that was modified by u_i that exist in $H_i(D)$ but not in $H_i[\mathcal{M}](D))$ or the other way round. As both $H_i(D)$ and $H_i[\mathcal{M}](D))$ would be same, we can exclude u_i from reenactment queries for answering the historical what-if query.

Theorem 15. *Let u not be a dependent update. Then, removing u from $\mathbb{R}(H[\mathcal{M}])$ and $\mathbb{R}(H)$ reenactment queries does not affect the result of historical what-if query. Let $Q_{H[\mathcal{M}]}$ be the result of removing u from $\mathbb{R}(H[\mathcal{M}])$ and Q_H be the result of removing u from $\mathbb{R}(H)$, then*

$$\Delta(Q_H(D), Q_{H[\mathcal{M}]}(D)) = \Delta(\mathbb{R}(H[\mathcal{M}])(D), \mathbb{R}(H)(D))$$

Proof. We now prove the theorem by induction over the number of updates in the history H. Induction Start: For a history $H = \{u_1, u_2\}$ with a single independent update u_2 after $u_1 \in \mathcal{M}$. Since u_2 is not a dependent update, the input and output tuples of the u_2 would be same. So there would not be any tuple t that is updated by $u_2(t) = t'$ that exists in just $\mathbb{R}(H_2[\mathcal{M}](D))$ or $\mathbb{R}(H_2(D))$ so the delta as the the answer of historical what-if query $(\Delta(Q, \mathbb{R}(H_2)(D), \mathbb{R}(H_2[\mathcal{M}])(D)))$ would not be changed by removing u_2 from reenactment queries $\mathbb{R}(H)$ and $\mathbb{R}(H[\mathcal{M}])$. Induction Step: Assume $H = \{u_1, \dots, u_n\}$ with up to n updates. We just need to reenact dependent updates to answer the historical of what-if query. We have to show that the same holds for $H = \{u_1, \dots, u_n, u_{n+1}\}$ whereas u_{n+1} is not a dependent update. Since u_{n+1} is not a dependent update, the input and output tuples of u_{n+1} would be same in $\mathbb{R}(H_{n+1}(D))$ and $\mathbb{R}(H_{n+1}[\mathcal{M}](D))$. So, there would not be any tuple that is updated by u_{n+1}, where $u_{n+1}(t) = t'$ and $t' \notin \mathbb{R}(H_{n+1}(D)) \vee t' \notin \mathbb{R}(H_{n+1}[\mathcal{M}](D))$. Therefore, the output of u_{n+1} would not change the result of the delta for answering the historical what-if query. The result of the what-if query over H with $n + 1$ updates would be same as the result for the H with just n updates. So, we can exclude u_{n+1} from reenactment queries to compute the answer of historical what-if query and this concludes the proof. □

Example 29. *Consider Example 26, after detecting u_2 as a dependent update in Example 28, we can apply program slicing by removing u_2 from the reenactment query. $\mathbb{R}^O(H) = \mathbb{R}^O(u_3)$, so $\mathbb{R}^O(u_3)$ can be constructed based on the reenactemnt query for*

the first update $\mathbb{R}^O(u_1)$ *instead of the second one* $\mathbb{R}^O(u_2)$.

$$\mathbb{R}^O(u_3) = \Pi_{ID,cu,co,p,(sf-2)\rightarrow sf}(\sigma_{p<=30\wedge sf>=10}(\mathbb{R}^O(u_1)))$$
$$\cup\, \sigma_{\neg(p<=30\wedge sf>=10)}(\mathbb{R}^O(u_1))$$

We propose to statically analyze the history to find a set of updates that can be safely excluded from reenactment. The result of this analysis is a conservative overestimation of the real set of dependent updates. To this end we use symbolic execution [53] to determine whether an update may depend on a modification. We start from a symbolic instance that consists of a single tuple of variables, i.e., we treat the input as a c-table [47]. The effect of an update on a symbolic instance are encoded as constraints over the variables of the instance. To determine whether an update u depends on a modification we test satisfiability of such constraints using a constraint solver. We discuss our approach in the following section.

8.7 Symbolic Execution

In the previous section, we demonstrated that it is sufficient to reenact dependent updates to answer a historical what-if query. However, to check whether an update is dependent we would need to evaluate the history up to and including this update. Obviously, this would defy the purpose of program slicing (to exclude updates from reenactment). In this section we develop a method to compute a superset of the set of dependent updates for a historical what-if query without having to access the database instance. We achieve this goal by using incomplete database techniques [46] to check for each update whether there exists a database instance for which the update is dependent. This is akin to *symbolic execution* [22, 53] which is used in software testing to determine inputs that would lead to a particular execution path in the program. Specifically, we use *Virtual C-tables* [52, 73] (*VC-tables*) as a compact representation of all possible single tuple instances, demonstrate how to eval-

uate updates with possible worlds semantics over such representations, and present a method for determining whether there exists an instance for which a particular update is dependent using a constraint solver.

Incomplete Database and Virtual C-Tables. An incomplete database $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of deterministic databases called possible worlds. Each D_i represents one possible state of the database. Queries (and updates) over incomplete databases $\mathcal{D}$ are evaluated using so-called possible world semantics where the result of the query (update) is the set of possible worlds derived by applying the query (update) to every possible world in $\mathcal{D}$, i.e., for a query Q and incomplete database $\mathcal{D}$ we have

$$Q(\mathcal{D}) = \{Q(D) \mid D \in \mathcal{D}\}$$

For our purpose, we will use an incomplete database consisting of possible worlds containing one tuple per relation. This incomplete database contains one world for any combination of such singleton relations. We then evaluate updates from the original and modified history over this incomplete database and search for worlds in which updates are dependent. However, the number of possible tuples per relation (and, thus, also the number of possible worlds) is exponential in the number of attributes of the relation. For instance, consider a relation with n attributes and a domain with m values. Then there are m^n tuples for this relation that we can form using the values of the domain. Thus, for efficiency we need a more compact representation of an incomplete database. Here, we employ Virtual C-tables [52, 73] which extend C-tables [46] to support scalar operations over values. A VC-table is a relation with tuples whose values are symbolic expressions over a set of variables Σ and where each tuple t is associated with a condition $\theta(t)$ (the so-called local condition). Here we use the grammar shown in Figure 2.4. A VC-database is a set of VC-tables. A VC-db D encodes an incomplete database which consists of any possible world that

can be generated by assigning a value to each variable in Σ, evaluating the symbolic expressions for each tuple in D and including tuples in the possible world whose local condition evaluates to true. We use $Mod(D)$ to denote the set of possible worlds encoded by the VC-database D (and apply the same notation for VC-tables).

Example 30. *Reconsider relation Order from Example 24. Here, we just consider three attributes that are used by updates in the history* ($Country, Price, ShippingFee$)*. A VC-table over this schema is shown on the top left in Figure 8.10. This VC-table contains a single tuple with three variables* $x_{Country}$*,* x_{Price}*, and* $x_{ShippingFee}$ *with a local condition true (shown on the right of the tuple). Consider the variable assignment* $x_{Country} = UK$*,* $x_{Price} = 10$*, and* $x_{ShippingFee} = 0$*. Applying this assignment, we get the possible world consisting of the tuple* $(UK, 10, 0)$*.*

Updates on VC-Tables. Previous work on VC-tables has only considered evaluation of queries. For our purpose we need to be able to evaluate updates over VC-tables with possible world semantics. That is, the possible worlds represented by the result of applying an update to a VC-table are the possible worlds derived by computing the update over every possible world from the input. There exist two possible ways how an update can affect a tuple t in a VC-table: (i) either the update's condition evaluates to false and the values of t are not modified or (ii) the update's condition evaluates to true and update's function *Set* is applied to the values of t. Since in a VC-table the result of an updates condition depends on variable assignments, we have to provision for both cases. For that the result of the update contains tuple t with an updated local condition that ensures that t is only included if ϕ evaluates to false on t and *Set*(t) with a location condition that ensures that *Set*(t) is only included if ϕ evaluates to true. The symbolic expressions of *Set*(t) are computed by substituting any reference to attribute A in an expression in *Set* with the currrent value $t.A$. There may exist multiple input tuples t, t', t'', ... which are all projected onto the same out-

put, i.e., there exists t_{out} such that $t_{out} = Set(t) = Set(t') = \ldots$. Tuple t_{out} exists in the result of the update as long as $\theta(u)$ holds for at least one of these inputs. That is the local condition of t_{out} is a disjunction of conditions $\phi(t) \wedge \theta(t)$ for any input t where $Set(t) = t_{out}$. We formally define updates below. We use $\phi(R, t)$ to denote the local condition of tuple t in relation R and for convenience define $\phi(R, t) = false$ for any $t \notin R$. Furthermore, $\theta(u)[t]$ for update u and tuple t denotes the result of substituting references to attributes in $\theta(u)$ with their value in t.

Definition 34. *Let R a VC-table and u be an update applied to R. We define:*

$$u(R) = \{Set(t) \mid t \in R\} \cup \{t \mid t \in R\}$$

$$\phi(u(R), t) = \phi(R, t) \vee \bigvee_{t' : Set(t') = t} \phi(t') \wedge \theta(u)[t']$$

Note that we can simplify the resulting instance by evaluating constant subexpressions in symbolic expressions and by removing any tuple t for which $\phi(t) = false$. Furthermore, observe that in worst-case evaluating of a sequenced of n udpates over a VC-table can lead to an instances that is 2^{n-1} times larger than the input since we generate two output tuples for each input tuples in the worst case (if no two inputs are projected onto the same output). We can avoid this blow-up by slightly extending our model to allow for global conditions. A global condition $\Phi(R)$ is a logical expression over the variables Σ which has to evaluate to true for an assignment to yield a possible world. To more encode an update result compactly we introduce new variables for every result tuple t of an update and then add global constraints that ensure that these new variables store the values of the tuple after an update. Specifically, for an update u, tuple t from the update's input, and attribute A, we add a fresh variable $x_{A,id}$ and add a conjunct $x_{A,id} =$ if $(\theta(t))$ then $Set(t.A)$ else $t.A$ to the global condition Φ. Note that using this encoding, the result of a sequence of n updates over a relation with m attributes has the same number of tuples as the input

and the number of conjuncts in the global condition is bound by $n \cdot m$. Furthermore, each conjunct is of size linear in the size of the expressions θ and Set.

Example 31. *Continuing with Example 30, consider the first update statement u_1 from Figure 8.2 with $\theta := Price >= 40$ and $Set := (ShippingFee \leftarrow 0)$. After execution of the u_1 over the relation shown on the top left of Figure 8.10, we get an instance with two tuple. First is the result of applying the update to t (i.e., shipping fee is updated to 0). The local condition of this tuple is generated by substituting the value of attribute shipping fee from the input tuple $x_{ShippingFee}$ for the reference to this attribute in θ. The second tuple is a copy of the input tuple and represents the case when the input does not fulfill the condition of the update. Thus, the local condition of this tuple enforces that the condition θ fails for the input tuple which is the case when $\neg(x_{Price} \geq 40)$. After the applying the second update u_2 we get the relation shown in the middle of Figure 8.10. Each of the four tuples is generated from one of the two tuples from result of u_1 and either represents the case when u_2's condition holds on this tuple or does not hold on this tuple. An example for how to use a global condition to encode the result compactly is shown on the bottom of Figure 8.10. Here we further optimized this idea by only introducing new variables for attributes that potentially get affected by an update.*

We now prove that our definition for updates over VC-tables complies with possible world semantics.

Theorem 16. *Let D be a VC-database and u an update. We have:*

$$Mod(u(D)) = u(Mod(D))$$

Proof. WLOG consider an assignment χ to the variables Σ and let D_χ denote the possible world corresponding to this assignment. We have to show that $u(D_\chi) = \chi(u(D))$. □

Note that by induction, Theorem 16 implies that evaluating a history H over a VC-database also has possible world semantics.

Symbolic Execution. To compute $\mathsf{Dep}^*(H, \mathcal{M})$ for a modification $\mathcal{M}$ and history H we create a VC-database D with single tuple with fresh variables for relation in the database's schema. We then independently evaluate H and $\mathcal{M}$ over this instance and store the intermediate result after each update. We use $D_{H,i}$ ($D_{H[\mathcal{M}],i}$) to denote the result of evaluating the first i updates from H ($H[\mathcal{M}]$) over D. Then to test for an update u_i whether it should be included in $\mathsf{Dep}^*(H, \mathcal{M})$ we need to check whether there exists a database D where u_i is in $\mathsf{Dep}(H, \mathcal{M}, D)$.

which implies that there exists a tuple in $t \in D$

Recall that $u_i \in \mathsf{Dep}(H, \mathcal{M}, D)$

Definition 35. *Let H be a history and $m = u_{orig} \leftarrow u_{mod}$ a modification, we define*

$$(\phi \vee \phi') \wedge (((\theta_i \wedge \theta_{orig}) \vee (\theta_i \wedge \theta_{mod}))) \wedge ((\neg\phi \vee \neg\phi') \vee (\bigvee_{i=1}^{n} x_i \neq y_i))$$

We use VC-Tables for symbolic execution of update operations and determining dependency of updates on modifications by the historical what-if queries. we can exclude dependent updates from reenactment as their output is the same in H and $H[\mathcal{M}]$.

symbolic execution of an update u on a VC-Table for each tuples creates two tuples in the VC-Table. One tuple will represent how u modifies variables of the table and the other tuple represent tuples that are not modified by the u. For the first tuple, it applies the Set_u on each related variables of the tuple of VC-Table and conjunct its θ_u with the previous $\theta_R(t)$ so $\theta = \theta_u \wedge \theta_R(t)$. For the second tuple, it keeps all the variables of VC-table as before and conjunct its $\neg\theta_u$ with the previous $\theta_R(t)$ therefor $\theta = \neg\theta_u \wedge \theta_R(t)$.

We can determine dependent updates in the history by generating VC-tables for each u and u' whereas $m = u \leftarrow u'$ and $m \in \mathcal{M}$. Then, we apply symbolic execution on these VC-tables for the remaining updates in the history. For each examine update (u_i), if we can generate a possible word for a tuple that is modified either by both u_i and u or u_i and u', u_i is a dependent update. Since, a possible world shows there is a possibility that a tuple was modified by the historical what-if query and the examined update which must be considered in the answer of the the historical what-if query.

Example 32. *In order to detect dependent update in Figure 8.10, we examine generating a possible world for a tuple in the VC-Table that is modified by both the first (u_1) and the second update (u_2) in the history in Figure 8.2. After symbolic execution of the second update where* $\theta := x_{Country} =' UK' \wedge x_{Price} <= 100$*, there are for tuples in the VC-Table. The first tuple which has the conditional function* $x_{Price} >= 40 \wedge (x_{Country} =' UK' \wedge x_{Price} <= 100)$ *representing it is modified by both updates. The possible world can be generated by evaluating* $(x_{Country} \leftarrow' UK', x_{Price} \leftarrow 40, x_{ShippingFee} \leftarrow 5)(x_{Price} >= 40 \wedge (x_{Country} =' UK' \wedge x_{Price} <= 100))$*. As* $40 >= 40 \wedge ('UK' ==' UK' \wedge 40 <= 100) := true$*. The possible world* D'' *after executing the first and second update statements can be* $D'' = \{'UK', 40, 5\}$*. So, the second update is dependent on the first update as it is possible that a tuple is modified by both updates.*

Theorem 17. *Suppose there is a single modification* $m = u \leftarrow u'$ *and* $m \in \mathcal{M}$*. Two VC-tables such as vtb and vtb′ with schema of table R* ($\textsc{Sch}(R)$) *are generated based on* $H = \{u_{orig}, u_1, \ldots, u_i, \ldots, u_n\}$*.* u_i *is a dependent update if*

$$\{u_i \mid u_i \in H \wedge \exists t \in vtb, t' \in vtb' \wedge$$

$$(\{\theta_{u_i} \wedge \theta_{u_{orig}}\} \subseteq \theta_R(t) \wedge Set(\theta_R(t)) = true) \vee$$

$$(\{\theta_{u_i} \wedge \theta_{u_{new}}\} \subseteq \theta_R(t') \wedge Set(\theta_R(t')) = true)\}$$

Algorithm 3 Compute Dependencies

```
1: procedure ComputeDependencies(H, D, u)
2:     S[0] ← InitSymbolicIns(H, D)
3:     S[1] ← H[0](S[0])                                   ▷ H[0] = u
4:     S[1] ← θ(H[0](S[1]))
5:     depList ← u
6:     for i ← 1, H.length − 1 do
7:         if θ(H[i](S[i])) then
8:             depList ← H[i]
9:         end if
10:        S[i + 1] ← H[i](S[i])
11:    end for
12:    return depList
13: end procedure
```

Algorithm 3 shows how we implement detecting dependent updates. It first initialize the symbolic instance (VC-table). The next state for this symbolic instance is generates based on symbolic execution of update statements in the H. Note that H contains u and update statements which were executed after u and $H[0]$ equals to u. $S[i+1] \leftarrow H[i](S[i])$ implement symbolic execution of the update statement which is discussed in the Section 8.8. $\theta(H[i](S[i]))$ checks whether the condition of the update statement $H[i]$ is mutual satisfiable with the condition of the first update $H[0]$ that was applied in $S[1] \leftarrow \theta(H[0](S[1]))$. In the following section. We propose a method which uses a constraint programming formulation expressing symbolic execution of update statements as a mixed integer linear program (MILP).

8.8 MILP Compilation

In this section, we discuss how to use a standard MILP (Mixed integer linear

programming) solver to test satisfiability of the conditions generated by our dependency checking solution presented in the previous section. However, these conditions are typically not linear. We now introduce a compilation scheme that translates such logical expressions into linear constraints solvable by a standard MILP solver. This compilation scheme consists of a set of rules that are shown in Figure 8.11. Here M denotes an integer constant that is larger than all integer values used as attribute values. These rules are applied recursively to a constraint. Each rule generates a set of linear constraints. The MILP generated by these rules for an expression e consists of the union of all linear constraints produced by the rules for the subexpressions of e. The objective function is set to $sum(v)$ which will for the constraint solver to return a solution which satisfies the condition e if such a solution exists. For each subexpression e' of e the compilation produces a variable v' for which any solution to the MILP sets v' to the value that e' evaluates to. Thus, for any solution to the MILP we have that v stores the value of expression e for the values chosen for variables in the expression. In this rules we use v, var_1, . . . to denote integer variables and b, b_1, . . . to denote boolean variables. The translation rules applied here are mostly known rules applied in linear programming and some have been used in by the database community (e.g., [59]).

8.9 Summary

We present Mahif, a middleware for answering historical what-if queries using reenactment that is a declarative replay technique for simulating the effect of changes in database's history. We propose optimization methods such as programming and data slicing to improve performance of our approach. Programming slicing reduces number of update statements in a history to reenact. Our programming slicing applies symbolic execution idea. To implement the symbolic execution of database's history, we propose the basic algorithm and transformation rules to encode update statements

as a MILP problem. The other optimization method is data slicing filtering the amount of processing data to improve our approach without compromising accuracy.

Initial VC-Table

Country	Price	ShippingFee	
$x_{Country}$	x_{Price}	$x_{ShippingFee}$	*true*

VC-Table after first update

Country	Price	ShippingFee	
$x_{Country}$	x_{Price}	0	$x_{Price} \geq 40$
$x_{Country}$	x_{Price}	$x_{ShippingFee}$	$\neg(x_{Price} \geq 40)$

VC-Table after second update

Country	Price	ShippingFee	
$x_{Country}$	x_{Price}	5	$x_{Price} \geq 40 \wedge (x_{Country} = UK \wedge x_{Price} \leq 100)$
$x_{Country}$	x_{Price}	$x_{ShippingFee} + 5$	$\neg(x_{Price} \geq 40) \wedge (x_{Country} = UK \wedge x_{Price} \leq 100)$
$x_{Country}$	x_{Price}	0	$x_{Price} \geq 40 \wedge \neg(x_{Country} = UK \wedge x_{Price} \leq 100)$
$x_{Country}$	x_{Price}	$x_{ShippingFee}$	$\neg(x_{Price} \geq 40) \wedge \neg(x_{Country} = UK \wedge x_{Price} \leq 100)$

Compact VC-Table with global condition after second update

Country	Price	ShippingFee	
$x_{Country}$	x_{Price}	$x_{ShippingFee,2}$	*true*

$$\Phi = (x_{ShippingFee,1} = \text{if } (x_{Price} \geq 40) \text{ then } 0 \text{ else } x_{ShippingFee})$$

$$\wedge\, (x_{ShippingFee,2} = \text{if } (x_{Country} = UK \wedge x_{Price} \leq 100) \text{ then } x_{ShippingFee,1} + 5 \text{ else } x_{ShippingFee,1})$$

Figure 8.10. Running example for evaluating updates over VC-Tables.

$$\frac{e := e_1 < e_2}{\begin{array}{r} v_1 - v_2 + b \times M \geq 0 \\ v_2 - v_1 + (1-b) \times M > 0 \end{array}} \quad \frac{e := e_1 = e_2}{\begin{array}{r} v_1 - v_2 + b \times M > 0 \\ v_2 - v_1 + (1-b) \times M \geq 0 \\ v_2 - v_1 + b \times M > 0 \\ v_1 - v_2 + (1-b) \times M \geq 0 \end{array}} \quad \frac{e := e_1 \leq e_2}{\begin{array}{r} v_1 - v_2 + b \times M > 0 \\ v_2 - v_1 + (1-b) \times M \geq 0 \end{array}}$$

$$\frac{e := \neg e_1}{b + b_1 = 1} \quad \frac{e := e_1 \wedge e_2}{\begin{array}{r} b_1 + b_2 - 2b - 1 \leq 0 \\ b_1 + b_2 - 2b \geq 0 \end{array}} \quad \frac{e := e_1 \vee e_2}{\begin{array}{r} b_1 + b_2 - 2b \leq 0 \\ b_1 + b_2 - b \geq 0 \end{array}}$$

$$\frac{e := e_1 + e_2}{v_1 + v_2 - v = 0} \quad \frac{e := \text{if } (e_c) \text{ then } e_1 \text{ else } e_2}{\begin{array}{r} v_{if} + v_{else} - v = 0 \\ v_{if} - v_1 \leq 0 \\ v_{if} - v_1 + M - M \cdot b_c \geq 0 \\ v_{if} - M \cdot b_c \leq 0 \\ v_{if} + b_c \cdot M \geq 0 \\ v_{else} - v_2 \leq 0 \\ v_{else} - M + M \cdot b_c \leq 0 \\ v_{else} - v_2 + M \cdot b_c \geq 0 \\ v_{else} + M - Mb_c \geq 0 \end{array}} \quad \frac{e := x}{v_x}$$

Figure 8.11. Compilation rules for translating constraints into an MILP (the remaining comparison operators are omitted since they can be expressed using negation)

CHAPTER 9
APPLICATIONS

In this chapter, we introduce set of applications which use the MV-semiring model and reenactment such as provenance-aware versioned dataworkspaces and debugging transactions.

9.1 Provenance-aware Versioned Dataworkspaces

Data preparation, curation, and analysis tasks are often exploratory in nature, with analysts incrementally designing workflows that transform, validate, and visualize their input sources. This requires frequent adjustments to data and workflows. Unfortunately, in current data management systems, even small changes can require time- and resource-heavy operations like materialization, manual version management, and re-execution. This added overhead discourages exploration. We present *Provenance-aware Versioned Dataworkspaces* (*PVD*s), our vision of a sandboxed environment in which users can apply — and more importantly, easily undo — changes to their data and workflows [60]. A PVD keeps a log of the user's operations in a light-weight *version graph* structure. We describe a model for PVDs that admits efficient automatic refresh, merging of histories, reenactment, and automated conflict resolution. We also highlight the conceptual and technical challenges that need to be overcome to create a practical PVD.

Exploratory data curation systems should support the analyst in tracking, understanding, and eventually resolving uncertainty in a way that keeps her focused on her data and analysis. Current data management platforms support neither this exploratory mode of operation, nor do they expose uncertainty introduced by curation operations. Provenance-aware workflow systems do help the analyst to keep

track of her operations, but only once a workflow has been developed and applied to inputs. Pay-as-you-go construction and modification of workflows is not supported. Provenance-aware databases can track uncertainty and potential errors in data, but would require the user to manually expose the uncertainty in curation operations. In short, provenance is a critical tool for enabling exploration, but current systems are lacking in several respects:

- **Regret-free exploration** The user should be able to operate in a *sandboxed* environment where she can change past decisions and data derived based on these decisions should be automatically refreshed. Both base data and derived data are versioned.
- **Full accountability through provenance tracking** The system should maintain both a record of the transformations executed by the user and their dependencies as well as be able to provide provenance at the data-level.
- **Automatic conflict detection and resolution** The system should automatically detect conflicts that exist in the data as well as conflicts that are based on automatic refresh of derived data. Furthermore, when detecting a conflict, the system should propose potential resolution strategies.
- **Merging of transformation pipelines** The system should enable analysis pipelines to be merged. For instance, a user may want to update an analysis based on recent changes to a database which requires merging the changes into the pipeline.
- **Uncertainty as a first-class concept** Whenever an operation introduces uncertainty, the system should track and propagate this uncertainty through further operations, and be able to explain whether an output is uncertain and how the uncertainty affects an analysis result. This requires fine-grained provenance.

Note that existing provenance-aware workflow systems can track transformations ap-

plied by a user to data in a workflow and may even keep track of changes to the workflow [35]. However, automatic refresh of derived data based on changes to previous steps in a workflow (e.g., Alice's change to Q_{JT}) is not supported. We present the **Virtual Version Graph** model (VVG), a simple, yet powerful model for representing version histories that supports derived objects which are updated automatically as well as provides a clean semantics for changing past decisions (e.g., by modifying a transformation). Furthermore, we discuss Provenance-aware Versioned Dataworkspaces (PVD), our vision for a new type of sandboxed curation and analysis environment based on the VVG model. Data curation and analysis over large data sets can be hindered by the cost of rerunning steps in a pipeline. The PVD approach can lazily materialize relation versions once they are needed (e.g., to refresh a visualization shown to the analyst).

9.2 Post-mortem Debugging of Transactions

Aside from providing a solid platform for research on provenance and related fields, GProM and the research fueling the system have also enabled novel applications of provenance that would not have been possible before. In this section, we introduce postmortem debugging of transactions as an important example for this type of application. Debugging transactions and understanding their execution is of immense importance for developing OLAP applications, to trace causes of errors in production systems, and to audit the operations of a database. Debugging transactions, just like debugging of parallel programs, is hard because errors may only materialize under certain interleavings of operations. This problem is aggravated by the wide-spread use of lower isolation levels. Nonetheless, even for serializable histories an error may only arise for some execution orders. To debug an error, we have to reproduce the interleaving of operations which lead to the error. This problem can be addressed by supporting post-mortem debugging for

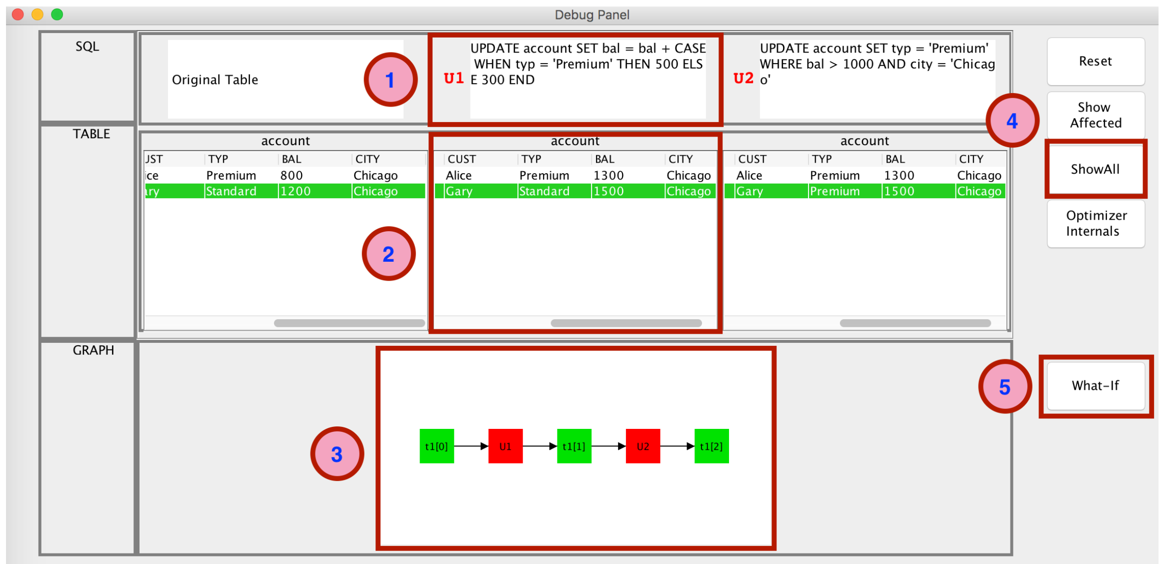


Figure 9.1. Screenshot of the Debugger GUI

transactions, i.e., enabling a user to retroactively inspect transaction executions to understand how the statements of a transaction affected the database state. While there are debuggers for procedural extensions of SQL, e.g., Microsoft's T-SQL Debugger (http://msdn.microsoft.com/en-us/library/cc645997.aspx), these debuggers treat SQL statements as black boxes, i.e., they do not expose the dataflow within an SQL statement. Even more important, they do not support post-mortem debugging of transaction executions within their original environment.

Supporting post-mortem debugging for transactions is quite challenging, because past database states are transient and the dataflow within and across SQL statements is opaque. While temporal databases provide access to past database versions, this is limited to committed versions. In [61], we present a non-destructive,

```sql
UPDATE account
SET bal = bal + CASE WHEN typ = 'Premium'
                     THEN 500 ELSE 300 END

UPDATE account SET typ = 'Premium'
WHERE bal > 1000 AND city = 'Chicago';
```

(a) Transaction T

account

cust	typ	bal	city
Alice	Premium	800	Chicago
Gary	Standard	1200	Chicago

(b) Before T

account

cust	typ	bal	city
Alice	Premium	1300	Chicago
Gary	Premium	1500	Chicago

(c) After T

Figure 9.2. Example transaction

post-mortem debugger for transactions that relies on GProM's reenactment techniques to reproduce the intermediate states of relations seen by the operations of a transaction. The approach uses provenance to expose data-dependencies between tuple versions and to explain which statements of a transaction affected a tuple version. Advanced debuggers for programming languages allow code to be changed during a debugging session to test a potential fix for a bug. We exploit the fact that GProM supports reenactment of hypothetical transactions to support such what-if scenarios,

i.e., changes to a transaction's SQL statements. Being based on GProM's reenactment functionality, our approach uses the temporal database and audit logging capabilities available in many DBMS and does not require any modifications to the underlying database system nor transactional workload.

Example 33. *Transaction T shown in Figure 9.2 adds a bonus to bank accounts ($500 for premium customers and $300 for standard customers) and gives premium status to all accounts whose balance is larger than $1000. Figure 9.2 also shows the state of the account relation before and after the update. For instance, after the execution of Transaction T, Gray's account enjoys premium status. However, Gary has only received a $300 bonus, the bonus for standard accounts. Our debugger can be used to inspect the internal states of the transaction that lead to this behaviour. Figure 9.1 shows a screenshot of the debugger's GUI for Transaction T. We show one column for each of T's operations plus a column for the initial states of the relations accessed by T. Each such column shows the SQL code of the statement (①) and the relation (②) modified by the statement (the version created by the statement). For each tuple version, we show which transaction created that version. In Figure 9.1, the user has selected Gary's account. Thus, the debugger shows a provenance graph (③) for this tuple and highlights the updates that affected it. From the intermediate states of the relations and the provenance graph it is immediately clear that the bonus payment was applied when Gary's status was still standard. Our debugger supports two types of what-if scenarios by clicking the "What-if" button (⑤): 1) the user can edit the data in a table and 2) the user can modify, delete, or add an update statement.*

9.3 Summary

We give an overview of research contributions related to our approach. The reenactment technique that is applicable for a wide range of use cases that are not necessarily all related to provenance. Our work provides solid support for classical ap-

plications of provenance and has enabled novel applications. We discuss provenance-aware versioned dataworkspaces and post-mortem debugging of transactions as exciting use cases of the system.

CHAPTER 10
EXPERIMENTS

In our experiments we study 1) the performance of provenance capture 2) the performance of querying provenance and 3) the overhead for transaction execution comparing our approach (using reenactment, audit logging and history maintenance) with an approach that directly stores provenance. All experiments are run with DBMS X as a backend (name omitted due to licensing restrictions) and 4) the performance of answering historical what-if queries and the effectiveness of our optimization methods. We use a synthetic workload to evaluate how our approach scales in various parameters and a TPC-C workload to test its performance for realistic transactions. All experiments were executed on a machine with 2 x AMD Opteron 4238 CPUs (12 cores in total), 128 GB RAM, and 4 x 1TB 7.2K HDs in a hardware RAID 5 configuration.

10.1 Setup and Workload

Datasets and Workload. We use a relation with five numeric columns. Values for these attributes are chosen from a uniform distribution. We created variants $R10K$, $R100K$, and $R1000K$ with 10K, 100K, and 1M tuples and no significant history ($H0$). Additionally, we generated three variants of $R1000K$ with different history sizes $H10$, $H100$, and $H1000$ (100K, 1M, and 10M tuples history). At first, we only consider transactions that consist solely of update statements. We vary the following parameters: U is the number of updates per transaction, e.g., $U10$ is a transaction with 10 updates. T is the number of tuples affected by each update. Unless stated otherwise, we use $T1$. The tuple to be updated is uniform randomly selected using the primary key. The default mode for experiments is `SERIALIZABLE` (SI).

Compared Methods. We compare different configurations for capturing provenance for a single transaction - each using a subset of the optimizations described in Sec. 6. Experiments were repeated 100 times and we report the average runtime. **NoOpt (N)**: Computes the provenance of all tuples in a relation, even tuples that were not affected by the transaction, i.e., we do not apply the filter condition on the version annotation attributes. **Prefilter (P)**: Only returns provenance of tuples affected by the transaction using a selection on the disjunction of the conditions of the transaction's updates (see Sec. 7.2). The database system was instructed to materialize the intermediate result corresponding to each update in the reenactment query using temporary relations. **Prefilter+Opt (PO)**: This is the same as *Prefilter*, but we merge operators (particularly, projections) to reduce the number of query blocks. **HistJoin (HJ)**: We use a join to compute partial provenance as described in Sec. 7.2. This configuration merges operators where possible. The maximum allocated execution time for each method is 1,000 sec.

10.2 Performance of Provenance Capture

In this set of experiments we execute the transactional workload beforehand and measure the performance of capturing provenance for transactions with this workload. We study how our reenactment approach scales in database and history size as well as complexity of the transaction (number of operations, amount of modified tuples, types of updates).

Relation Size and Updates/Transaction. We compute the provenance of transactions varying the number of updates per transaction ($U1$ up to $U1000$) and the size of the database ($R10K$, $R100K$, and $R1000K$) without significant history ($H0$). Figure 10.1 shows the runtime of these provenance computations. We scale linearly in R and U. By reducing the amount of data to be processed by the reenactment query and by merging operators, the PO approach is up to three orders of magnitude

faster than the naive N configuration.

History Size. We capture provenance for transactions with 10 updates ($U10$) over relations with 1M tuples ($R1000K$) and history sizes: $H0$, $H10$, $H100$, and $H1000$. As shown in Figure 10.2, N exhibits almost constant performance. The runtime is dominated by evaluating the reenactment query over 1M tuples (all tuples in one version of the relation) hiding the impact of scanning the history. Since we have not created any indexes on the history relations, the PO approach only has the advantage of processing less tuples in the provenance computation, but still has to scan most of the history to find tuples that were updated.

Comparing Optimization Techniques. Figure 10.3 shows results for varying the number of updates ($U1$ to $U1000$) using *R1000K-H1000*. Compared with P, PO benefits from avoiding materialization. This optimization is more effective for larger transactions, as reenactment queries for such transactions are increasingly complex. While resulting in ∼20% improvement for U100, it improves the runtime by a factor of roughly 10 for U1000. The cost of PO is affected by the first selection that is applied to 1M tuples (no index on the history relation). This condition is linear in the number of update operations. The runtime of HJ is almost not affected by parameter U, because it is dominated by the join between historic relations. PO outperforms HJ by a factor of about 3. For $U1000$, the *NoOpt* method did not finish within acceptable time slot.

Affected Tuples Per Update. Figure 10.4 shows results for $U10$ where each update modifies 10, 100, 1000, or 10000 tuples in *R1000K-H1000* relation. As evident from Figure 10.4, the runtime is not significantly affected when increasing the number of affected tuples per update. It is dominated by scanning the history and filtering out updated tuples (PO) or the self-join between historic relations (HJ). Increasing the T parameter by 3 orders of magnitude results in runtime increase of about 150% (PO)

and 20% (HJ).

Index vs. No Index. Figure 10.5 shows the effect of replicating the indexes defined for the *R1000K-H1000* relation to its corresponding history relation. We vary U ($U1$ to $U1000$). We omit the N (no benefit from indexes) and P (consistently outperformed by PO) configurations. Using indexes improves execution time of queries that apply PO considerably.

Inserts and Deletes. We now also use inserts and deletes in addition to updates. We use the $R1000K$ relation in this experiment. Each operation is chosen randomly (25% probability) from: **1)** An update as used in the previous experiments ($T1$); **2)** an insert that inserts one new tuple; **3)** an insert that inserts the result of a query over a different relation (1 tuple inserted); and **4)** a delete that removes 1 randomly chosen tuple. Figure 10.6 shows the results for $U20$ varying history size ($H10$ to $H1000$). The results indicate that performance is comparable to performance for updates.

TPC-C. We capture provenance for the TPC-C benchmark. We execute a TPC-C workload over an instance with 32 warehouses. The resulting database is roughly 16GB large. The benchmark defines 5 transaction types, out of which 2 are read-only. We compared the N and PO methods for 3 transaction types that execute updates. Figure 10.7 shows the result for computing the provenance of a single transaction of each type. Each of these transactions only modifies a few tuples. Thus, the cost for PO is quite low. The cost for N is dominated by scanning the full input relation.

Isolation Levels. Figure 10.8 compares the performance of capturing provenance for *SI* and *RC-SI*. We use *R1000K-H1000* and vary the number of updates per transaction (*U1* to *U1000*). As expected, *SI* reenactment is more efficient than *RC-SI* reenactment, because for *RC-SI* we have to check for each tuple and update whether the tuple is visible to the update. The impact is more noticeable for efficient con-

figurations such as P and larger number of updates (*U1000*). The runtime of N is dominated by a full scan of the large input and history tables and by having to produce 1M output rows. For *U1000*, the method N did not finish within the maximum allocated time slot.

10.3 Querying Provenance

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

Aggregation of Provenance Information. Figure 10.9 shows the results for running an aggregation over the provenance computation (denoted as *Ag*-). These results indicate that the performance of aggregation on provenance information is comparable to provenance computation. Even more, aggregation considerably improves performance for (O). For *U1000*, *Ag-O* results in 95% improvement over O (because it reduces the size of the output) while *Ag-HJ* improves performance by $\sim$ 13% compared to *HJ*.

Filtering Provenance. A user may only be interested in part of the provenance that fulfills certain selection conditions, e.g., bonuses larger than a certain amount. Figure 10.10 shows the runtime of provenance computation and querying (denoted as *Q*-). Performance of querying the results of provenance capture is actually slightly better than just computing provenance, because it reduces the size of the output and selection conditions over provenance are pushed into the SQL query implementing the provenance computation.

Querying Versions Annotations. A user can also query version annotations which are shown as boolean attributes in the provenance, e.g., to only return provenance for

tuples that were updated by a certain update of the transaction. Figure 10.11 shows the performance results for such queries. We fix an update $u \in T$ and only return provenance of tuples modified by this update. This reduces the runtime of O queries significantly by reducing the size of the output.

10.4 Overhead and Eager Provenance Capture

We use audit logging and time travel to reconstruct provenance of past transactions. We now quantify the runtime and storage overhead of DBMS X's built-in temporal and audit features. We measure the execution time of 10,000 transactions with $U10$ and $T1$ run over the $R1000$ instance. Figure 10.15 shows the total runtime for three configurations: without temporal and audit logging features (W/O), with temporal features, and with both the temporal and audit logging features. If history maintenance is activated then this results in about 12% runtime overhead (see Figure 10.15). This result agrees with DBMS X's documentation which states 5% overhead for mixed read-write workloads. Also activating audit logging results in a total overhead of $\sim$ 19%.

We now compare our approach with eager provenance capture during transactions execution. We consider two configurations: **1Step** stores a separate provenance record for each tuple version and statement in an extra relation. Each record is linked to the provenance record for the previous tuple version. The provenance of a transaction is reconstructed by recursively joining these provenance records; **Full** stores the complete derivation history of each tuple in an additional column. Results are shown in Figure 10.30.

Transaction Execution Overhead. Using the workload from Sec. 10.4, we compare the overhead for transaction execution incurred by these two eager methods with our method. The performance of our method and *1Step* remains stable when increasing

the size of the history. In contrast, the overhead of *Full* increases with the history size, as the size of provenance per tuple increases and the attribute storing provenance has to be updated by every operation. Both *1Step* and *Full* do significantly slow down the transaction processing showing up to a factor of 7 higher overhead than our approach.

Storage Size. We compare the storage size used by the three methods for a table with $1M$ rows varying the size of the history ($H10$, $H100$, and $H1000$) and number of tuples affected by each update ($T1$, $T10$, and $T100$). For our method we show the total storage space as well as the breakdown into a relation plus history and the audit log. Only the size of the audit log is affected by the T parameter. Thus, we only show our method for $T10$ and $T100$ since the other methods require the same storage for all T values. The results shown in Figure 10.30 demonstrate that in the worst case (1 tuple affected per update), our method requires up to $\sim$4 times more storage than the best approach. This overhead is caused by the audit log storing one SQL statement per modified tuple. However, if more tuples are affected by each statement then our method requires about the same or less space than the alternatives.

Retrieving Provenance. We now compare the performance of using reenactment (the PO method) for retrieving provenance with *1Step* and *Full*. Figure 10.14 shows the result for capturing provenance of transactions with $U10$ and $T1$ varying the history size (H). We created relevant indexes for each method. Optimized reenactment outperforms both alternatives, because *Full* requires filtering tuples based on the transaction identifier that is stored in the provenance column and *1Step* requires a recursive query or multi-way join to reconstruct the provenance of a transaction from provenance records for each update.

10.5 Answering Historical What-if Queries

We have conducted experiments to 1) evaluate the performance of our ap-

proach and compare it with the naive approach, 2) examine the effectiveness of the proposed optimization methods, 3) study how our approach scales in relation sizes and evaluate the performance of our approach by other important factors.

Datasets and Workload. We use a taxi trips dataset from the City of Chicago's open data portal [3]. The dataset contains data about trips reported to the City of Chicago as a regulatory agency. The original dataset has 113M rows and 23 attributes. the dataset contains trip information such as Company, Taxi ID, Trip Start Timestamp (when the trip started), Trip Seconds (time of the trip in seconds), Trip Miles (distance of the trip in miles), Community Area, Tips, Tolls, Extras (tips , tolls and extra charges for the trip respectively), and Trip Total(total cost of the trip). We did data cleaning and data transformation. We removed incomplete data and removed tuples which do not contain one of the mentioned attributes. We also transformed type of some attributes to the number and created a new Company ID attribute for each unique combination of Company and Taxi ID. We created three different tables based on the different granularity of trip start time (Month, Day, Hour). These tables have three grouping attributes (Company ID, Community Area and Trip Start Timestamp) using for grouping data and used aggregation functions (e.g. Sum, Min, Max, AVG and COUNT) over remaining attributes to create ten aggregated attributes like Sum_TripTotal which shows sum of all Trip Total for each group of data. These generated tables have three different sizes about $70K$, $1M$, and $7M$ rows based on the granularity of their Trip Start Timestamp. We created temporal tables for these three tables with a lower granularity of Trip Start Timestamp to simulate update operations that modifies these tables to update their aggregated attributes.

Update Operations and Transactional History. We consider sequence of update

[3]https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew downloaded on 6/11/2018

operations to simulate the transactional history that were executed under snapshot isolation concurrency control protocol. Historical update operations are defined by a SET clause that assigns an attribute a relative value, and a WHERE clause has a point predicate on two categorical attributes and a range predicate on the trip start time attribute (e.g. `UPDATE TAXI_TRIPS SET SUM_TRIPTOTAL = SUM_TRIPTOTAL + 10 WHERE COPMANYID=1 AND COMMUNITYAREAS=1 AND STARTDAY>= 20130101 AND STARTDAY<=20130103;`).

In all experiments, we just used number of updates in the partial history which includes the update that is changed in the historical what-if query by the user and update operations that are executed after it. U is the number of updates in history (e.g. $U1000$ shows 1000 updates). Note that we just consider updates in the history that modify the same relation in the database that the historical what-if query changed. So $U100$ represents, 100 updates in the history modifying the same relation and we just consider these 100 updates in our experiments and we filter all other operations that are executed on different relations in the database. D is the percentage of updates that are dependent on the update was modified by the historical what-if query and its default is 10% or $D10$. For instance, $D10$ and $U100$ means that 10 updates out of 100 updates in the history are determined dependent by our programming slicing optimization. T is the percentage of tuples in the relation that are affected by the historical what-if query and its default is 10%. Historical what-if queries affect different number of tuples to adjust D and T variables. All other update statement affects a single tuple.

Compared Methods. We consider the following methods in our experiments.

Reenact All (RA) creates a reenactment query for H and $H[\mathcal{M}]$. Then, we then use time travel to access the database state as of the time the modified update, run both reenactment queries over this instance, and then compute the delta.

Reenact Data Slicing (RDS): same as the previous method except that we restrict reenactment on the part of data to be relevant by our data slicing optimization.
Reenact Program Slicing (RPS): same as the *RA* method except that we only reenact a subset of updates in the history determined by our program slicing optimization. **Reenact Program Slicing + Data Slicing (RPS+DS)**: we apply both optimization techniques, data slicing and programming slicing.
Naive (N): it creates a copy of the database as of the start time of the transaction which is modified by the what-if query (Creation), re-executes $H[\mathcal{M}]$ over this database copy (Exe), and then computes the delta $\Delta(Q, H(D), H[\mathcal{M}](D))$ by running queries (Delta).

Total Cost. We consider three different relation sizes ($70K$, $1M$, and $7M$) and vary the number of updates in the history ($U10$ to $U6000$). We compare our method with the naive method for three different percentage of affected data by the historical what-if query such as $T0$ in Figure 10.16, $T10$ in Figure 10.17, and $T25$ in Figure 10.18. *Mahif* has better performance for most cases. Especially, for low number of updates in the history *Mahif* outperforms the *Naive* method. For the high number of updates in the history $U6000$ still *Mahif* has a better performance for the largest relation size ($7M$). In general, *Mahif* has more steady performance for the same number of update operations and different relation sizes whereas it does not need any additional storage. In contrast to *Mahif*, *Naive* performance changes for different relation sizes. *Naive* depends on the relation size considerably as it must recreate the whole relation and process all tuples in the old and the new relation. Moreover, the *Naive* method need to copy the whole relation for its implementation which has an additional storage cost. Another factor is the percentage of affected data by the historical what-if query (T) that has a greater impact on *Mahif* compared to the *Naive* method that does not implement data slicing or filtering data.

Breakdown Naive. We examine the performance of each step in the *Naive* method shown in Figure 10.19. The naive method has three main steps: 1) create a copy of the relation as of the start time of the transaction that is modified by the historical what-if query (*Creation*), 2) executes $H[\mathcal{M}]$ over the relation to produce $H[\mathcal{M}](D)$ (*Exe*), and finally computes the delta $\Delta(Q, H(D), H[\mathcal{M}](D))$(*Delta*). *Creation* produces all tuples in relation and *Delta* processes them so they are dependent on the number of tuples in the relation. *Exe* runs all update operations in the $H[\mathcal{M}]$ over the relation and its performance affected by the number of executed update operations and tuples in the relation that are modified by them. *Creation* and *Delta* steps are just dependent on relation sizes whereas *Exe* also depends on the number of updates in the history (U). *Exe* significantly increases for the large number of updates ($U6000$) and the large relation size ($7M$) and it becomes the most time consuming step in the *Naive* method.

Breakdown Mahif. Figure 10.20 presents the performance of each step of *Mahif* including 1) detect dependent updates in the history by symbolic execution (*SymEx*) 2) generate reenactment query based on the symbolic execution result and applies data slicing (*GR+DS*) 3) execute the reenactment query that is produced in the previous step to compute $\Delta(Q, H(D), H[\mathcal{M}](D))$ (*RPS+DS*). *SymEx* processes all update operations in the history to detect dependent updates and *GR+DS* generates reenactment query for all dependent updates and then applies data slicing so both of them are affected by the number of processing update operations. *SymEx* and *GR+DS* steps depend on U while *RPS+DS* depends on U and the relation sizes as it executes the reenactment query of all dependent updates over the relation.

We evaluate the effectiveness of our proposed optimization methods and the effect of related variables such as number of dependent updates and the percentage of affected data on these methods.

Optimization. In this experiment we evaluate the effect of our optimization techniques. The percentage of modified tuples by the historical what-if query is 0.000045 roughly 0%. We compare the runtime of *RA* with *RPS+DS* method in Figure 10.21. In this experiment we vary the number of updates in history ($U10$, $U100$, and $U1000$) and relation sizes ($70K$, $1M$, and $7M$). Our fully optimized method *RPS+DS* performance is outperforming *RA* by at least by a factor of ~ 16 ($U10$ and $70K$ relation size). The optimized method *RPS+DS* significantly improves performance for larger number of update statements and bigger relation sizes up to a factor of 500 ($U1000$ and $7M$). This experiment confirms the effectiveness of our optimization techniques. In other experiments, we consider different optimization methods and we do not show the result of reenacting all updates without applying any optimization.

Dependent Updates. Figure 10.22 shows the performance of *RPS* and *RPS+DS* reenactment methods for different percentage of dependent updates. We use 1000 updates to the relation with $1M$ rows, and 0% affected tuples by the historical what-if query. D shows the percentage of dependent updates detected by the symbolic execution. $D0.1$ means 1 out of 1000 updates are dependent while by $D100$, all 1000 updates are dependent. The result shows increasing the number of dependent updates degrade *RPS* performance significantly and it increases the runtime by a factor of $\sim$ 37 as the number of updates to reenact increase. It has much less effect on *RPS+DS* and increase the runtime by a factor of ~ 2 because it also applies the data slicing technique which filters input data and improves performance. Data slicing improves the total performance of *RPS+DS* significantly as 0% of tuples are affected by the historical what-if query and data slicing filters data that are not affected by the historical what-if query. Note that this experiment also can represent the effect of number of modified updates by the user in the historical what-if query as it would also result in reenacting more updates. If a user request a historical what-if query which changes more updates in the history, all of these modified updates must be

considered for reenactment and we can not filter or exclude them by program slicing. So, it would be similar to increasing number of dependent updates that must be used in reenactment.

Affected Data. The percentage of affected tuples by the historical what-if query is examined in Figure 10.23. The experiment is executed for 1000 updates on the relation with $1M$ rows. $T0$ means 0% of tuples ($\sim$ 45 out of 1M) are modified by the historical what-if query whereas $T93$ shows 93% of tuples are affected by it. The result demonstrates varying T does not change the performance of *RPS* noticeably as it just depends on the number of update operations and excluding independent updates from reenactment and it does not filter data. It increases the runtime of *RDS* and *RPS+DS* considerably as they apply data slicing and they are dependent on the number of tuples that are modified by the historical what-if query.

Relation Size. We vary the relation sizes ($70K$, $1M$, and $7M$) and the number of updates in the history ($U10$ to $U6000$). We present the result of our optimized methods for three different percentages of affected data. $T0$ in Figure 10.24, $T10$ in Figure 10.25, and $T25$ in Figure 10.26. Note that the result of *RDS* method for $U6000$ does not display because of our database system limitation that can not execute the reenactment for this method and it runs out of process memory. The result shows the runtime increases notably for higher T. *RPS+DS* runtime increases up to two orders of magnitude for $U6000$ update history and $7M$ relation size from $T0$ to $T25$. This shows the importance of amount of data that need to be processed. Additionally, this experiment shows for higher number of updates in the history *RPS* runtime performance is better than *RDS*. For $T0$, $70K$, and $U6000$, *RPS* outperforms *RDS* by a factor of 17. For $T25$, $70K$, and $U1000$, *RPS* outperforms better than *RDS* by a factor of 2.

Number of Modified Attributes and Conditions. We examine the effect of

increasing the number of modified attributes (Figure 10.28) and conditions in the WHERE clause (Figure 10.29) for each update in the history. We present results for $T0$, the relation with $1M$ rows, and 1000 updates in the history. This experiment evaluates the runtime of each step in *Mahif* similar to what we present in Figure 10.20. *SymEx* increases slightly by changing the number of modified attributes and conditions from 4 to 7 as they increase the MILP problem complexity by adding new constraints. Increasing the number of attributes that are modified by updates increases *SymEx* more than extending the number of conditions of each update as updating an attribute create a constraint recursively whereas adding new condition creates a simple new constraint. Note that *RPS+DS* is faster in Figure 10.29 as adding conditions improves the data slicing method to filter more data.

Aggregate Query. In this experiment, we can apply different type of Q in answering historical what-if queries $(\Delta(Q, H(D), H[\mathcal{M}](D)))$. In all other experiments we assume Q is selecting all the rows and all columns. We compare this kind of query with an aggregate query that runs on top of the answer of historical what-if query in Figure 10.27. For our database, we aggregate and adds all the Sum_TripTotal together. We consider different relation sizes ($70K$, $1M$, and $7M$) and number of updates ($U10$ to $U6000$) for $T10$. The runtime of different optimization methods decrease slightly.

10.6 Summary

Our evaluation confirms the efficiency and scalability of our approach - the presented techniques for retroactively computing provenance scale to relations with millions of tuples, large transactions (1000 update statements), large number of updated tuples, and large histories. The proposed optimizations increase performance by several orders of magnitude. The runtime overhead for transaction execution incurred by our approach due to auditing and history maintenance is below 20% for

our experimental workloads - a small price to pay compared to eager materialization of provenance while transactions are executed (about 133% and higher). Our experiments shows our historical what-if query approach outperform the naive method in most cases whereas it does not need any additional storage. The proposed optimization methods for answering historical what-if queries are very effective and for large number of updates and database relations they improve performance considerably. Our approach is scalable and other factors like the query(Q) type in a historical what-if query, number of modified attributes, or conditions of update operations do not affect the performance of our approach notably.

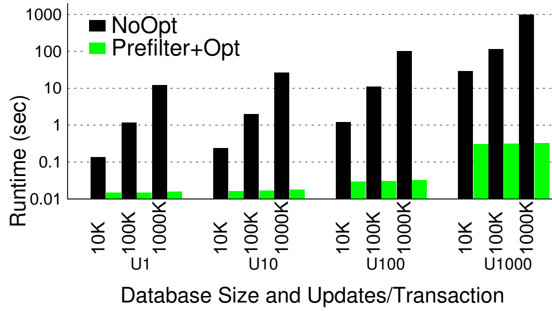


Figure 10.1. Relation size

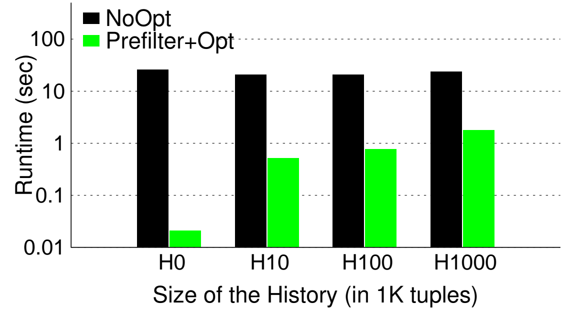


Figure 10.2. History size

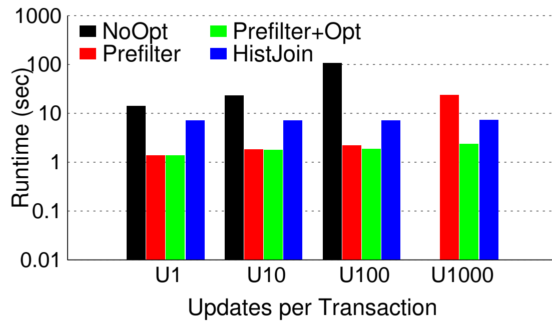


Figure 10.3. Optimization methods

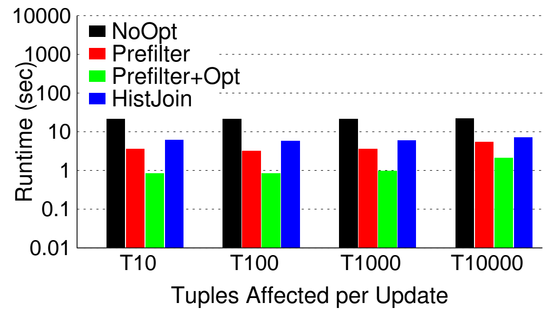


Figure 10.4. Affected tuples

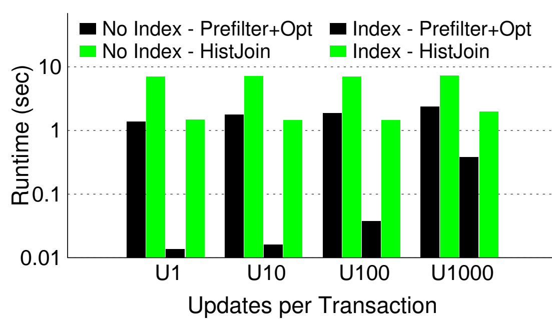


Figure 10.5. Index vs. no index

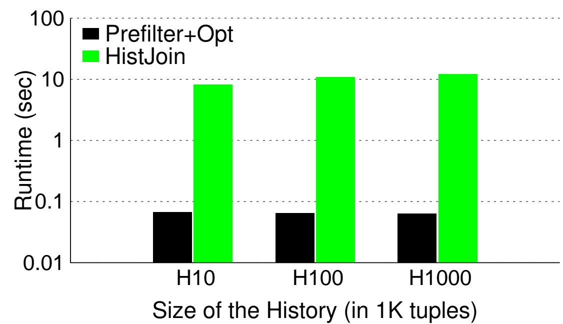


Figure 10.6. Inserts and deletes

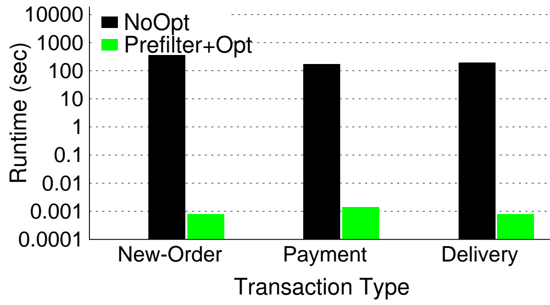


Figure 10.7. Provenance for TPC-C

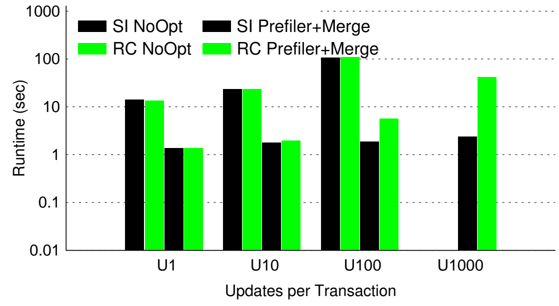


Figure 10.8. Isolation Levels

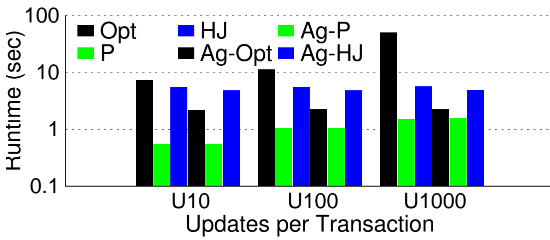


Figure 10.9. Aggregation

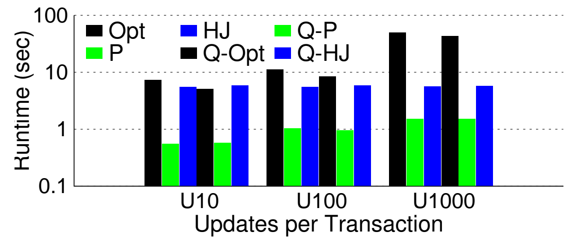


Figure 10.10. Query Provenance

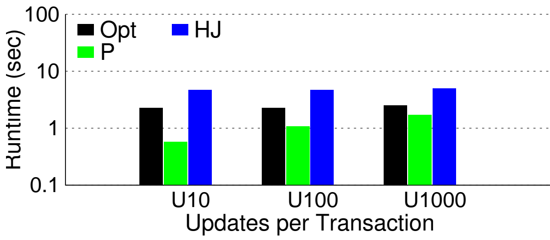


Figure 10.11. Query Vers. Ann.

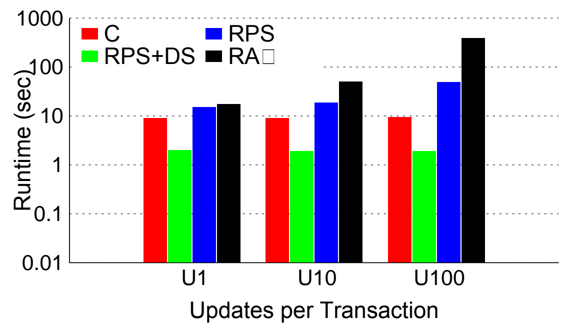


Figure 10.12. Updates/Transaction

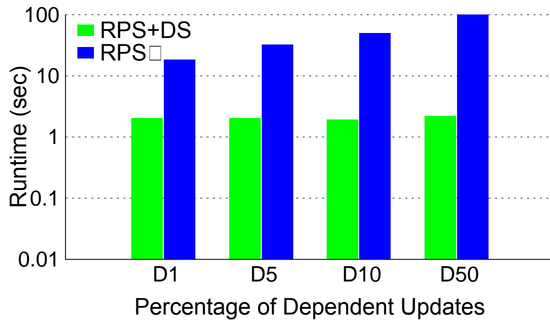


Figure 10.13. Dependent Updates

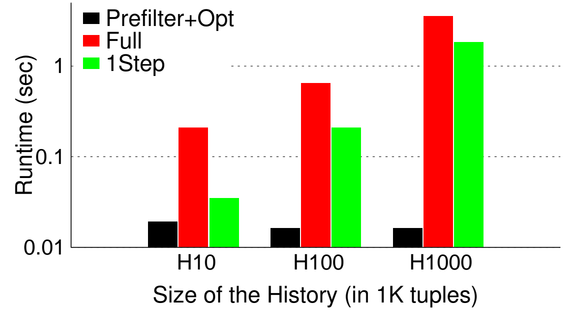


Figure 10.14. Provenance Retrieval

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

Figure 10.15. Runtime Overhead

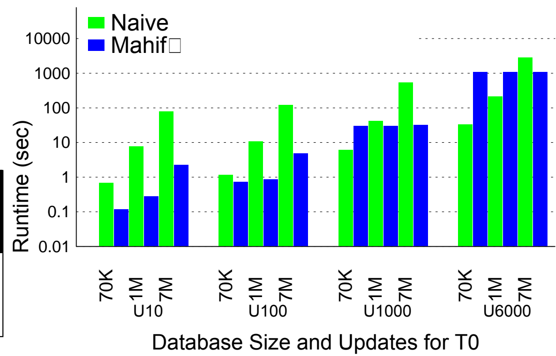


Figure 10.16. Naive vs. Mahif

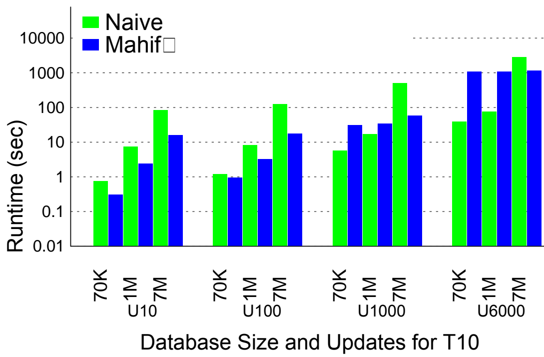


Figure 10.17. Naive vs. Mahif

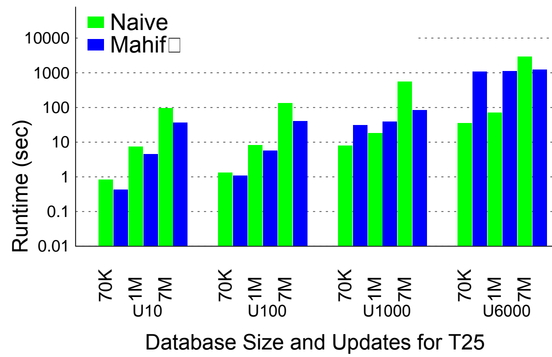


Figure 10.18. Naive vs. Mahif

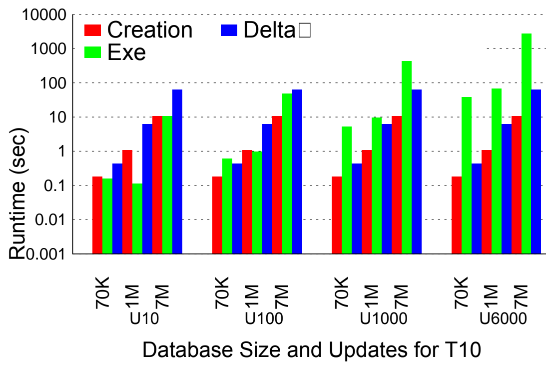


Figure 10.19. Breakdown Naive

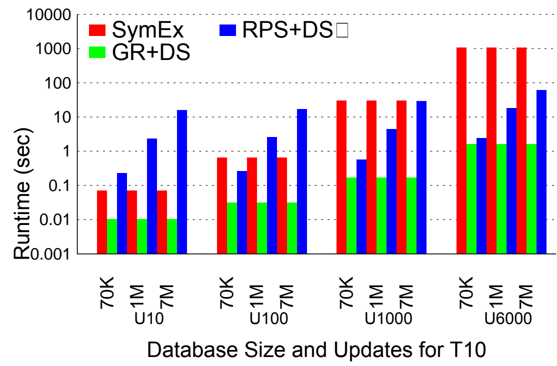


Figure 10.20. Breakdown Mahif

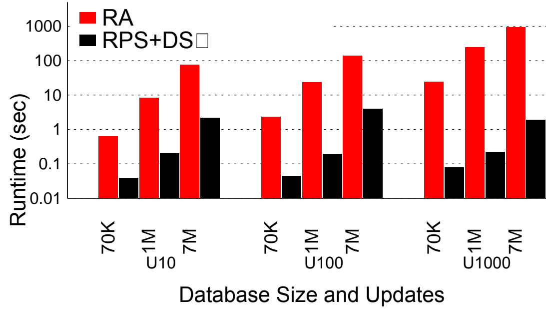


Figure 10.21. Optimization

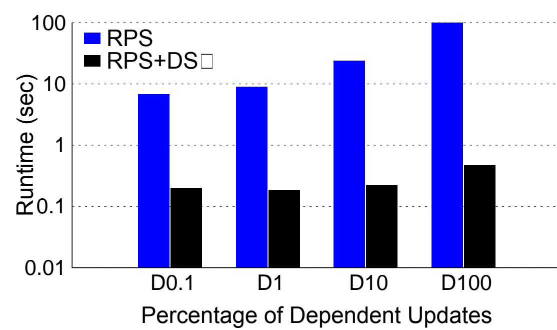


Figure 10.22. Dependent Updates

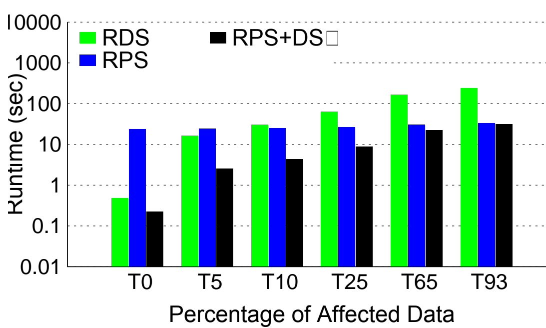


Figure 10.23. Affected Data

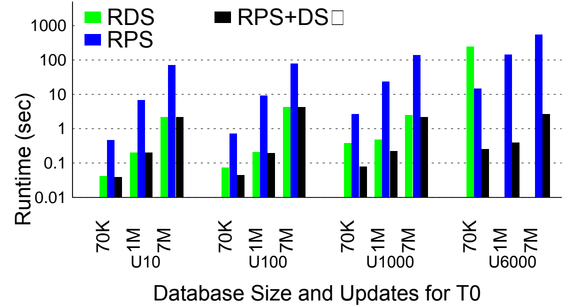


Figure 10.24. Relation size

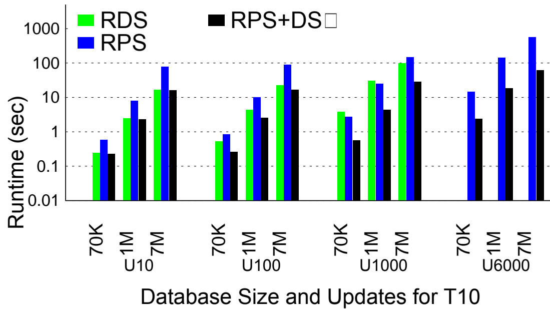


Figure 10.25. Relation size

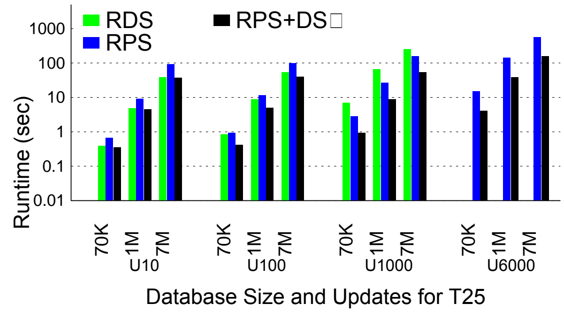


Figure 10.26. Relation size

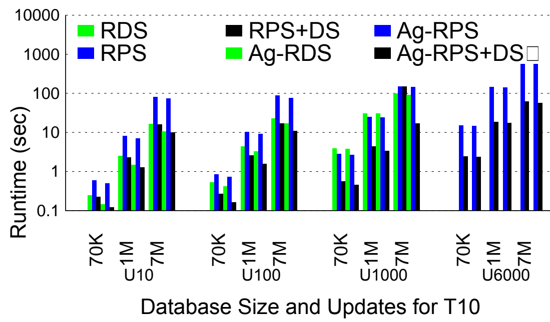


Figure 10.27. Aggregate Query

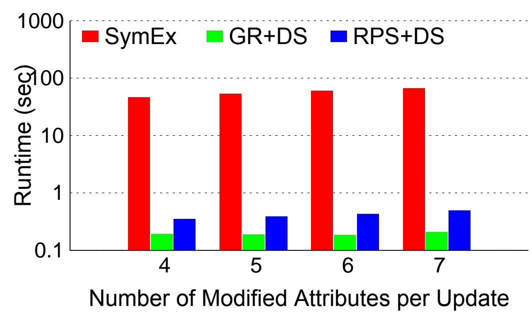


Figure 10.28. #Attributes

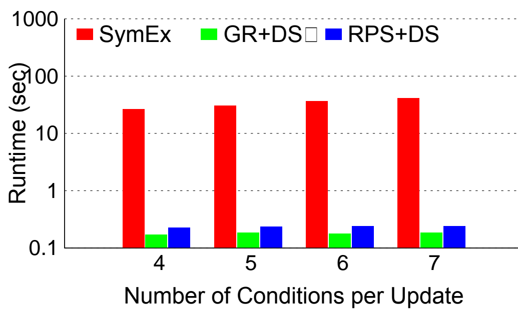


Figure 10.29. #Conditions

Total Runtime (sec)

+ Relative Overhead (rel)

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

Storage Size (MB)

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

Figure 10.30. Eager vs. reenactment

CHAPTER 11
CONCLUSIONS AND FUTURE PLANS

We present the first solution for capturing provenance for transactions run under SI and RC-SI. Our approach is based on *reenactment*, i.e., replaying updates and transactions as queries with MV-semiring annotated semantics for updates and transactions. The resulting MV-semirings provide full account of the derivation history of tuples that were produced by concurrent transactions. Using audit logging, time travel, and a relational encoding of reenactment, we retroactively capture the provenance of tuples produced by transactional histories using a standard DBMS without incurring any additional runtime or storage overhead apart from audit logging and maintaining transaction time histories for time travel. Our experimental results confirm that our implementation in the GProM system can efficiently compute the provenance of transactions and scales to large databases, histories, and transactions. Additionally, we present how our model and reenactment approach can be used in different applications such as post-mortem debugging of transactions, PVDs, and answering historical what-if queries (determine the effect of hypothetical changes to past operations of a business).

The diverse types of research threads and applications for which we have employed GProM, demonstrate the potential of our system and the feasibility of its modular, extensible design. We can extend our approach to support distributed databases or key-value stores. Our approach can be employed in such systems in order to convert concurrent execution of updates to batch of sequential updates for recovery purposes or improving their performance. Reenactment has many potential applications. One of useful applications for our reenactment approach is transaction backout where we need to undo the effect of a past committed transaction and we

need to determine dependent transactions as they need to be undone too. Our approach has a potential to be merged in other systems to improve their performance such as Vizier, a framework for user-friendly and effective data curation which enables user to re-use and adapt workflows for new data. In addition, reenactment can be applied to address other type of analytical queries such as how-to queries to determine how should the input change to achieve the desired output data. We need to suggest major updates to the database based on past transnational history.

APPENDIX A

PROOFS

Proof. 2 Let $h : \mathcal{K} \to \mathcal{K}'$ be a semiring homomorphism, then h commutes with any Q in the above algebra if h is applied to Q. Let I be a $\mathcal{K}$ database instance. Then, $h(Q)(h(I)) = h(Q(I))$ □

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

Proof. 3 h_U is a surjective semiring homomorphism. □

Proof. Note that h_U evaluates the symbolic expression of an representative k of a congruence class $[k]_\sim$. To prove that h_U is well-defined we have to show that $k \equiv_\sim k' \Rightarrow h_U(k) = h_U(k')$, i.e., all representative of a congruence class are mapped to the same element of K. If $k \equiv_\sim k'$ then there has to exist at least one sequence of applications of the equivalences that define the congruence relation of $\mathcal{K}^\nu$ (Figure 2.8) such that applying this sequence to k we get k'. We prove the implication by induction over these equivalences. Most of these equivalences follow directly from the construction of h_U and the definition of MV equivalences:

Laws of commutative semirings

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

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

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

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

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

Evaluation of expressions with operands from K

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

Equivalences involving version annotations

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

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

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

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

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

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

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

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

Proof. 4 Any semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism $h^\nu : {\mathcal{K}_1}^\nu \to {\mathcal{K}_2}^\nu$ as defined below. If h is surjective then so is h^ν.

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

□

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

Since the symbolic expressions of ${\mathcal{K}_1}^\nu$ and ${\mathcal{K}_2}^\nu$ are generated by the same

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

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

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

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

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

□

Proof. 5 Let h^ν be a lifted homomorphism as defined in Theorem 4. h^ν commutes with updates. □

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

Update:

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

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

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

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

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

<u>Inserts</u>:

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

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

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

<u>Deletes</u>:

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

<u>Commits</u>:

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

where

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

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

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

otherwise we get

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

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

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

□

Proof. 6 Let h^{ν} be a lifted homomorphism (Theorem 4). h^{ν} commutes with histories.

□

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

It remains to be shown that h^{ν} commutes with the computation of $R[\nu]$ over the results of past transactions, i.e., applying h^{ν} to the result of this computation is the same as applying it to every input $R[T,\nu]$ of the computation. By iteratively

pushing the homomorphism through all transactions involved in a history the result follows.

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

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

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

It remains to be shown that

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

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

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

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

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

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

Theorem 7: *Let u be an update and $\mathbb{R}(u)$ its reenactment query. Then, $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$.*

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

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

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

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

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

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

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

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

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

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

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

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

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

The proofs for inserts and deletes are analogous. □

Proof. 10 For Q and Q' be two $\mathcal{RA}^+$ queries and $\mathcal{K}$ a naturally ordered semiring. Then

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

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

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

□

Proof. Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be naturally ordered semirings. It is proven that $Q \sqsubseteq_{\mathbb{N}[X]} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}} Q'$. Furthermore, $Q \sqsubseteq_{\mathcal{K}_1} Q' \Rightarrow Q \sqsubseteq_{\mathcal{K}_2} Q'$ if there exists a surjective semiring homomorphism $\mathcal{K}_1 \rightarrow \mathcal{K}_2$. The first part of the theorem follows from the fact that UNV is a surjective semiring homomorphism (Theorem 3) and that the property of being naturally ordered is preserved for $\mathcal{K}^\nu$ semirings (see Lemma 18 below).

The second parts holds if we can demonstrate that 1) if $\mathcal{K}$ is naturally ordered then so is $\mathcal{K}^{\nu}$, 2) a lifted homomorphism h^{ν} is surjective if h is surjective and commutes with updates, the annotation operator, and histories. In particular, since any valuation $\chi : X \rightarrow \mathcal{K}$ can be uniquely extended to a homomorphism $Eval_{\chi} : \mathbb{N}[X] \rightarrow \mathcal{K}$ [41], 1) and 2) imply the second part. As mentioned above 1) is proven in Lemma 18. The lifting of homomorphisms was shown to preserve surjectivity (Theorem 4) and these homomorphisms commute with updates (Theorem 5) and histories (Theorem 6). The fact that h^{ν} commutes with the annotation operator is proven in Lemma 19. □

Lemma 18. *Let $\mathcal{K}$ be a naturally ordered semiring, then $\mathcal{K}^{\nu}$ is naturally ordered.*

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

Consider two elements k and k' of $\mathcal{K}^{\nu}$ in normal form (a sum of elements that do not contain addition). Let $k_1, \ldots, k_n$ be the summands in k and $k_1', \ldots, k_m'$ be the summands in k'. WLOG consider $m = 1$, because if an additive inverse can be found for the sum of $k_1', \ldots, k_m'$ then there has to exist an inverse for each element k_i'. Treat every summand as an ordered tree whose leafs are elements of $\mathcal{K}$ and $\times$-operations are considered n-ary. Furthermore, order operands of such monomials as follows: 1) elements of $\mathcal{K}$ precede any elements wrapped in version annotations and are ordered based on an arbitrary extension of the natural order of $\mathcal{K}$ to a total order; 2) elements

wrapped in version annotations are ordered based on some order over $\mathbb{A}$ based on the outermost version annotation; 3) two elements with the same version annotation are ordered based on the order of their children. For example, for $\mathcal{A}_1(\mathcal{A}_2(x_1) \times x_2 \times \mathcal{A}_2(x_3))$ assuming $x_1 < x_2 < x_3$ in the extension of the natural order on $\mathbb{N}[X]$ we would order the elements of the monomial as follows: $\mathcal{A}_1(x_2 \times \mathcal{A}_2(x_1) \times \mathcal{A}_2(x_3))$. We now prove that inverses for an element k' cannot exists by induction over the structure of such summands (trees).

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

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

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

Case 1 ($k^{n+1} = \mathcal{A}(k^n)$): Note that the congruence relation of $\mathcal{K}^\nu$ does not manipulate individual version annotations. Thus, $-k^{n+1}$ would have to be of the form $\mathcal{A}(-k^n)$ such that $\mathcal{A}(k^n) + \mathcal{A}(-k^n) = \mathcal{A}(k^n + -k^n) = \mathcal{A}(0) = 0$. However, this leads to a

contradiction because k^n is of depth n and thus no additive inverse of k^n can exist.

<u>Case 2 ($k^{n+1} = \prod_{i=1}^{m} k_i$)</u>: We prove this case by induction over m. If $m = 2$ then $k^{n+1} = k_1 \times k_2$ and WLOG we have to distinguish 2 cases: 1) $k_1 = \mathcal{A}(k'_1)$ and $k_2 \in \mathcal{K}$ or 3) $k_1 = \mathcal{A}(k'_1)$ and $k_2 = \mathcal{A}(k'_2)$. Note that $k_1, k_2 \in \mathcal{K}$ conflicts with the fact that k^{n+1} is of height $n + 1$ and, thus, we do not have to consider this case. In any case we can construct $-k^{n+1}$ as either $-k_1 \times k_2$ or $k_1 \times -k_2$. For any $k \in \mathcal{K}$ no inverse exists. Thus, we have to find the inverse of an element $k_i = \mathcal{A}(k'_i)$. However, since k_i of height less than n we know that none such inverse exists. The inductive step is analog. □

Lemma 19. *Let h^ν be a lifted semiring homomorphism (as defined in Theorem 4). h^ν commutes with the annotation operator α.*

Proof.

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

For commits recall that

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

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

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

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

□

Proof. 11 The $\text{Rel}(R[T])$ operation is lossless. □

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

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

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

Now consider the schema created for $R[T]$. Applying the definition shown in Figure 5.4 the schema is $\textsc{Sch}(R) \triangleright P(R) \triangleright \mathcal{U}_1 \triangleright \mathcal{U}_C$ where $P(R)$ contains attributes Xid, Id, V and a provenance renaming of the attributes of R. The attribute name of $\mathcal{U}_1$ encodes $\nu(u_1)$ and the type of the update. The attribute name of $\mathcal{U}_C$ encodes $End(T)$. According to the definition of $\textsc{Rel}(R[T])$ every summand in the annotation of tuple t is encoded as a separate tuple $t \triangleright \textsc{Rel}^{End(T)}(T, R, R[T](t)[i])$. Applying the definition of $\textsc{Rel}^{End(T)}(T, R, R[T](t)[i])$, a summand $C^{id}_{T,End(T)}(U^{id}_{T,\nu(u)}(C^{id}_{T',End(T')}(x_{id})))$ would be encoded as $t \triangleright T' \triangleright End(T') \triangleright id \triangleright t(x_{id}) \triangleright True \triangleright True$. From T', $End(T')$ and id we can directly reconstruct $C^{id}_{T',End(T')}(x_{id})$. Based on the value $\mathcal{U}_1$ ($True$) and $\nu(u_1)$ and the type of the update (U) encoded in the name $\mathcal{U}_1$ it can be determined that the element we have constructed so far should be wrapped in $U^{id}_{T,\nu(u)}$. Finally, $End(T)$ is determined based on the name of $\mathcal{U}_C$.

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

determined using f_{id} as explained in Section 4.7.

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

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

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

$u_{n+1} = \mathcal{I}[Q, T, \nu](R)$: Every summand k_i in an annotation is either of the form 1) $C^{id}_{T,End(T)}(I^{id}_{T,\nu(u)}(k_{i_1} \times \ldots \times k_{i_m}))$ with $k_{i_j} = 1$, $k_{i_j} = C^{id'}_{T',End(T')}({k_{i_j}}')$, or $k_{i_j} = x_{id''}$ (if produced by a constant relation operator) where m is the numer of relation mentions and constant relation operators in Q; or 2) $C^{id}_{T,End(T)}({k_i}')$ where k'_u is produced by a sequence of n updates. The schema for the relational encoding is $\textsc{Sch}^{End(T)}(T, R) =$ $\textsc{Sch}(R) \triangleright ID_{\mathcal{P}}(\textsc{Sch}^{\nu(u_{n+1})}(T, R) \triangleright \textsc{Sch}^{\nu(u_{n+1})}(T, X_1) \triangleright \ldots \triangleright$

$\textsc{Sch}^{\nu(u_{n+1})}(T, X_m) \triangleright \mathcal{U}_{n+1}) \triangleright \mathcal{U}_C$. Cases 1) and 3) can be distinguished from case 2) based on whether all attributes in $\textsc{Sch}^{\nu(u_{n+1})}(T, R)$ are null or not. In case 1) $k_i{}'$ is an annotation produced by less than or equal to n updates and, thus, can be reconstructed based on the induction assumption. In case 2) we can construct the monomial $k_{i_1} \times \ldots \times k_{i_m}$ in the same fashion as in the base case as long as it is possible to determine which attributes in $\textsc{Sch}^{\nu(u_{n+1})}(T, R_1) \triangleright \ldots \textsc{Sch}^{\nu(u_{n+1})}(T, R_m)$ belong to which $\textsc{Sch}^{\nu(u_{n+1})}(T, R_l)$. This is possible using the query Q of the insert which is encoded in $\mathcal{U}_{n+1}$. The tuple id in $I^{id}_{T,\nu(u)}$ is reconstructed using the deterministic scheme introduced in Section 4.7.

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

Proof. 12 Let T be a SI transaction. Then:

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

□

Proof. We prove the theorem through induction over the number of operations in a transaction.

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

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

translation $TR(\mathbb{R}^R(T))$ of this reenactment query is

$$TR(\mathbb{R}^R(T)) = \sigma_{\mathcal{U}_1}(\text{Rew}(\mathbb{R}^R(T)))$$

$$\text{Rew}(\mathbb{R}^R(T)) = \rho_{\text{Sch}(\text{Rel}(R[T]))}(\text{Rew}(q) \times \rho_{\mathcal{U}_C}(\{(true)\}))$$

$$\text{Rew}(q) = (\Pi_{A,\mathcal{P}(R),True \rightarrow \mathcal{U}_1}(\sigma_\theta(\text{Rew}(R[\nu_u]))))$$

$$\cup\, (\sigma_{\neg\theta}(\text{Rew}(R[\nu_u])) \times \rho_{\mathcal{U}_1}(\{(false)\}))$$

$$\text{Rew}(R[\nu_u]) = \Pi_{\text{Sch}(R),Xid,TT_b \rightarrow V,Id,\text{Sch}(R) \rightarrow \mathcal{P}(R)}(R_{\nu_u})$$

Consider $k = R[T](t)$ for an arbitrary tuple t. As shown in the proof for Theorem 11 each summand k_i in k will be of the form

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

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

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

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

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

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

$\underline{u_1 = \mathcal{I}[Q(R_1, \ldots, R_n), T, \nu](R)}$: The reenactment query for u_1 is

$R[T, \nu_u] \cup \alpha_{I,T,\nu_u+1}(Q(D[T, \nu_u]))$. The relational rewrite for $TR(\mathbb{R}^R(T))$ is

$$TR(\mathbb{R}^R(T)) = \sigma_{\mathcal{U}_1}(\textsc{Rew}(\mathbb{R}^R(T)))$$

$$\textsc{Rew}(\mathbb{R}^R(T)) = q_1 \cup q_2$$

$$q_1 = (\rho_{\textsc{Sch}(R),ID_1(\mathcal{P}(R))}(\textsc{Rew}(R[\nu_u])) \times \textsc{Null}(ID_2(\mathcal{P}(Q) \triangleright \mathcal{U}_1))))$$

$$q_2 = (\Pi_{\textsc{Sch}(q_2),ID_1(\mathcal{P}(R)),ID_2(\mathcal{P}(Q)\triangleright\mathcal{U}_1))}(\rho_{\textsc{Sch}(Q),ID_2(\mathcal{P}(Q)\triangleright\mathcal{U}_1)}(\Pi_{\textsc{Sch}(Q),\mathcal{P}(Q),True\rightarrow\mathcal{U}_1}(\textsc{Rew}(Q)) \times \textsc{Null}(ID_1(\mathcal{P}(R[\nu_u])))))$$

$$\textsc{Rew}(R[\nu_u]) = \Pi_{\textsc{Sch}(R),Xid,TT_b\rightarrow V,Id,\textsc{Sch}(R)\rightarrow P(R)}(R_{\nu_u})$$

$$\textsc{Rew}(R_i[\nu_u]) = \Pi_{\textsc{Sch}(R_i),Xid,TT_b\rightarrow V,Id,\textsc{Sch}(R_i)\rightarrow P(R_i)}(R_{i_{\nu_u}})$$

Since $\mathcal{U}_1$ is true in the right input of the union and false in the left input, because of $\textsc{Null}(ID_2(\mathcal{P}(Q) \triangleright \mathcal{U}_1)))$, any result returned by $TR(\mathbb{R}^R(T))$ originates in the right input. It remains to be shown that $\textsc{Rew}(Q)$ produces the correct result, because the additional version annotation attributes ($\mathcal{U}_1$ and $\mathcal{U}_C$), derived in the same fashion as for update, are *true*. Note that for queries the rewrite rules are the rewrite rules introduced in Perm which were shown to derive a relational encoding of provenance

polynomials [40] except that a snapshot of relations is accessed and that the provenance attributes for a relation R contain additional attributes Xid, V, and Id. Since these additional attributes are not treated any different from the other provenance attributes of R in the rewrite rules, the correctness of $\text{Rew}(Q)$ follows from the correctness of the Perm rewrites.

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

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

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

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

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

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

The first case is analog to the base case for updates with the only exception that the relational encoding of k'_i is more complex. However, observe that the proof of the

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

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

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

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

Lemma 20. *Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be commutative semirings and $h : \mathcal{K}_1 \rightarrow \mathcal{K}_2$ a semiring homomorphism. Then the lifted homomorphism $h^\nu : {\mathcal{K}_1}^\nu \rightarrow {\mathcal{K}_2}^\nu$ as defined in [6] commutes with any RC-SI history H.*

Proof. As mentioned above and proven in [6], h^ν commutes with queries, updates, and SI histories. In the definition of $R[T, \nu]$, the committed relation version $R[\nu]$ is defined analog to SI histories. The same is true for predicate $\textsc{updated}(T, t, k, \nu)$ and $\textsc{validAt}(T, t, k, \nu)$. Based on Theorem 5.5 of [6] any lifted homomorphism commutes with $\textsc{updated}(T, t, k, \nu)$ and $\textsc{validAt}(T, t, k, \nu)$ as well as with the operations used in the definition of $R[\nu]$. Since these results do not depend on the admissibility of the input relation (which is based on the concurrency control protocol and thus different for SI and RC-SI), it only remains to show that the lifted homomorphism h^ν commutes with the operations of $R_{ext}[T, \nu](t)$, i.e., it can be pushed into the committed relation version accessed by $R_{ext}[T, \nu](t)$. We have

$$
\begin{aligned}
& h^\nu(R_{ext}[T, \nu])(t) \\
=& h^\nu(\sum_{i=0}^{n(R[\nu](t))} R[\nu](t)[i] \times \textsc{validEx}(T, t, R[\nu](t)[i], \nu) \\
& + \sum_{i=0}^{n(R[T,\nu-1](t))} R[T, \nu - 1](t)[i] \textsc{validIn}(T, t, R[T, \nu - 1](t)[i], \nu - 1))
\end{aligned}
$$

Any homomorphism h^ν commutes with addition. Thus,

$$
\begin{aligned}
= & \sum_{i=0}^{n(h^\nu(R[\nu](t)))} h^\nu(R[\nu](t)[i] \times \textsc{validEx}(T, t, R[\nu](t)[i], \nu)) \\
& + \sum_{i=0}^{n(h^\nu(R[T,\nu-1](t)))} h^\nu(R[T, \nu - 1](t)[i] \times \textsc{validIn}(T, t, R[T, \nu - 1](t)[i], \nu - 1)))
\end{aligned}
$$

and since h^ν also commutes with multiplication, we have

$$= \sum_{i=0}^{n(h^\nu(R[\nu](t)))} h^\nu(R[\nu](t)[i]) \times h^\nu(\textsc{validEx}(T, t, R[\nu](t)[i], \nu))$$
$$+ \sum_{i=0}^{n(h^\nu(R[T,\nu-1](t)))} h^\nu(R[T, \nu - 1](t)[i]) \times h^\nu(\textsc{validIn}(T, t, R[T, \nu - 1](t)[i], \nu - 1))$$

Given that $h^\nu(\textsc{updated}(T, t, k, \nu)) = \textsc{updated}(T, t, h^\nu(k), \nu)$, it follows that

$$h^\nu(\textsc{validEx}(T, t, R[\nu](t)[i], \nu))$$
$$=\textsc{validEx}(T, t, h^\nu(R[\nu])(t)[i], \nu)$$

Furthermore, the condition $\textsc{validIn}$ is based only on the outermost version annotation in a summand k. Since lifted homomorphisms by design do not manipulate version annotations it follows that:

$$h^\nu(\textsc{validIn}(T, t, R[T, \nu - 1](t)[i], \nu - 1))$$
$$=\textsc{validIn}(T, t, h^\nu(R[T, \nu - 1])(t)[i], \nu - 1)$$

Thus, we have

$$= \sum_{i=0}^{n(h^\nu(R[\nu])(t))} h^\nu(R[\nu])(t)[i] \times \textsc{validEx}(T, t, h^\nu(R[\nu](t))[i], \nu)$$
$$+ \sum_{i=0}^{n(h^\nu(R[T,\nu-1])(t))} h^\nu(R[T, \nu - 1])(t)[i] \times \textsc{validIn}(T, t, h^\nu(R[T, \nu - 1])(t)[i], \nu - 1)$$

This implies that h^ν can be pushed into $R_{ext}[T, \nu]$ and given that h^ν commutes with all other operations used to define $R[T, \nu]$ it follows that h^ν commutes with histories. □

Furthermore, we have introduced two new query operators that are used in reenactment. We now prove that lifted homomorphisms commute with these query operators. This means we only need to prove $\mathbb{N}[X]^\nu$-equivalence of operations with their reenactment queries, because this then automatically implies $\mathcal{K}^\nu$-equivalence for any naturally ordered semiring $\mathcal{K}^\nu$. The new query operators we have introduced are

the *version merge* operator $\mu(R, S)$ that merges two versions R and S of the same relation by only keeping the newest versions of tuples and the *version filter* operator $\gamma_\theta(R)$ which removes summands (tuple versions) which do not fulfill the condition θ expressed over the versions (pseudo attribute V) encoded in the version annotations.

Lemma 21. *Let $h^\nu : \mathcal{K}_1{}^\nu \to \mathcal{K}_2{}^\nu$ be a lifted homomorphism, then h^ν commutes with $\mu(R, S)$ if R and S are normalized admissible $\mathcal{K}_1{}^\nu$-relations.*

Proof.

$$\begin{aligned}
&h^\nu(\mu(R, S))(t) \\
=&h^\nu(\sum_{i=0}^{n(R(t))} R(t)[i] \times isMax(S, R(t)[i]) \\
&+ \sum_{i=0}^{n(S(t))} S(t)[i] \times isStrictMax(R, S(t)[i]))
\end{aligned}$$

Any homomorphism commutes with addition and multiplication. Furthermore, since h^ν preserves the structure of MV-semiring expressions, we know that $h^\nu(k^\nu)$ for any normalized MV-semiring element k^ν is a subset of the summands of k^ν. That is every summand in k^ν is preserved unless $h(k) = 0$ for all elements $k \in K$ that occur in the summand, because in this case the summand's expression is equivalent to 0 resulting in the summand being removed. Thus, as long as we can prove that $isMax(R, k) = isMax(R,\ h^\nu(k))$ and $isStrictMax(R, k) = isStrictMax(R, h^\nu(k))$ it follows that:

$$\begin{aligned}
=&\sum_{i=0}^{n(h^\nu(R(t)))} h^\nu(R(t)[i] \times isMax(S, R(t)[i])) \\
&+ \sum_{i=0}^{n(h^\nu(S(t)))} h^\nu(S(t)[i] \times isStrictMax(R, S(t)[i]))
\end{aligned}$$

Consider the definition of $isMax$, $isStrictMax$, $idOf$ and $versionOf$. Note that $h^\nu(idOf(k)) = idOf(k)$ and $h^\nu(versionOf(k)) = versionOf(k)$, because by construction of h^ν we have $h^\nu(X^{id}_{T,\nu}(k')) = X^{id}_{T,\nu}(h^\nu(k'))$ and thus $h^\nu(idOf(X^{id}_{T,\nu}(k'))) =$

$idOf(X^{id}_{T,\nu}(k'))$ as well as $h^\nu(versionOf(X^{id}_{T,\nu}(k'))) = versionOf(X^{id}_{T,\nu}(k'))$. From this immediately follows that $isMax(R,k) = isMax(R,h^\nu(k))$ and $isStrictMax(R,k) = isStrictMax(R,h^\nu(k))$ which concludes the proof. □

Lemma 22. *Let $h^\nu : \mathcal{K}_1{}^\nu \to \mathcal{K}_2{}^\nu$ be a lifted homomorphism, then h^ν commutes with $\gamma_\theta(R,S)$ if R and S are normalized admissible $\mathcal{K}_1{}^\nu$-relations.*

Proof. Substituting the definition of γ_θ we get:

$$
\begin{aligned}
& h^\nu(\gamma_\theta(R))(t) \\
=& h^\nu\Big(\sum_{i=0}^{n(R(t))} R(t)[i] \times \theta(R(t)[i])\Big) \\
=& \sum_{i=0}^{n(h^\nu(R(t)))} h^\nu(R(t)[i]) \times h^\nu(\theta(R(t)[i]))
\end{aligned}
$$

Recall that $\theta(k)$ is evaluated over the version ν of the outermost version annotation of each summand k_i in the normalized annotation k. Thus, we get

$$
\begin{aligned}
=& \sum_{i=0}^{n(h^\nu(R)(t))} h^\nu(R)(t)[i] \times \theta(h^\nu(R)(t)[i]) \\
=& \gamma_\theta(h^\nu(R)))(t)
\end{aligned}
$$

□

Finally, the following lemma establishes that if a Transaction T uses only updates, deletes, and inserts with singleton relations (operator $\{t \to k\}$ corresponding to an SQL statement of the form `INSERT INTO ... VALUES ...`) then $R[End(T)-1]$ contains all immediate predecessors of all tuple versions created by T 's updates and deletes. This is the first prerequisite for proving the correctness of $\mathbb{R}_{opt}(T)$, because $\mathbb{R}_{opt}$ avoids the use of the version merge operator by only using accesses to relation

versions as of $End(T) - 1$. In the following definition we make use of a predicate $\textsc{hasCreated}(T, t, k)$ which determines whether a summand k in the annotation of a tuple t has been created by Transaction T. Formally,

$$\textsc{hasCreated}(T, t, k) \Leftrightarrow \exists k' : k = X^{id}_{T,\nu}(k')$$

Definition 36. *Let H be a RC-SI history and $T \in H$. Consider a summand k in the annotation $R[T, End(T)](t)$ created by T ($\textsc{hasCreated}(T, t, k)$ is true). The immediate predecessor $\textsc{immPred}(T, t, k)$ is defined as the latest tuple version k' with identifier $idOf(k') = idOf(k)$ created by a transaction $T' \neq T$ in the annotation of a tuple t'. If no such version exists (e.g., T did insert k) then $\textsc{immPred}(T, t, k)$ is undefined.*

In other words, the immediate predecessor of a tuple version k is the last version of this tuple created by another transaction before the creation of k.

Lemma 23. *Let T be a transaction where each insert's query is of the form $\{t \rightarrow k\}$. If T is executed as part of a RC-SI history H then there exists a tuple t' such that $\textsc{immPred}(T, t, k)$ is present in $R[End(T) - 1](t')$.*

Proof. For any tuple version k created by Transaction T, there has to exist an operation u_i in T that first created a tuple version k' with $idOf(k) = idOf(k')$. Naturally, $R[\nu(u_i)(t')]$ for some tuple t' has to contain $\textsc{immPred}(T, t, k)$ if it is defined. We proof the lemma by contradiction. Assume that $R[End(T) - 1](t')$ does not contain $\textsc{immPred}(T, t, k)$. This can only be the case if there exists a Transaction T'' with $End(T'') < End(T)$ that did update or delete k. However, since u_i modified k we know that T would have to hold a write lock on the tuple version corresponding to k after $\nu(u_i)$ and under RC-SI write locks are held until transaction commit. Thus, no such Transaction T'' can exist. $\square$

Proof. **9** Let T be a RC-SI transaction. Then,

$$T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T) \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}_{opt}(T)$$

□

Proof. We first prove that $T \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(T)$ and then equivalence with $\mathbb{R}_{opt}$.

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

Assume that transaction $T = u_1, \ldots, u_n, c$ is updating a single relation R. As was shown in [6], the extension to multiple relations is straightforward. To prove equivalence it suffices to show that a reenactment query for an update $\mathbb{R}(u)$ is equivalent to the update u and that each such reenactment query is executed over the same input relation as in the original history H. The semantics for updates is the same under SI and RC-SI. The proof of $u \equiv_{\mathbb{N}[X]^\nu} \mathbb{R}(u)$ was already given in [6]. It remains to show that the input $R[T, \nu(u)]$ is the same as the input produced for $\mathbb{R}(u)$ by the reenactment query for Transaction T.

We prove this fact by induction over the number of updates in Transaction T.

Induction Start: Let $T = u_1, c$. This case is analog to SI and thus was already proven in [6].

Induction Step: Assume that $R[T, \nu(u_i)] = R_{ext}[T, \nu(u_i)]$ with $i \in \{1, \ldots, m\}$ where m is the number of operations in the Transaction T is correctly constructed by the reenactment query for T for any transaction with $m < n$ operations. We need to prove that for any transaction $T = u_1, \ldots, u_{n+1}, c$ we have that $R[T, \nu(u_{n+1})]$ is equal to the input for the reenactment query $\mathbb{R}(u_{n+1})$ of u_{n+1} within the reenactment query

$\mathbb{R}(T)$. In the reenactment query, the input to $\mathbb{R}(u_{n+1})$ is $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])$.

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

Based on the induction hypothesis we have

$$\mathbb{R}(u_n) = R[T, \nu(u_{n+1})]$$

. Thus, denoting $\nu(u_{n+1})$ as ν_{n+1}:

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

Note that $R_{ext}[T, \nu_{n+1}](t)$ is also defined as a sum over the elements from $R[T, \nu_{n+1}](t)$ and $R[\nu_{n+1}](t)$. Individual summands are filtered out using validIn and validEx. Thus, to proof that $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})]) = R[T, \nu(u_{n+1})]$, we have to show that if either the $isMax$ or $isStrictMax$ function returns 1 on a summand then the same is true for validIn respective validEx and vice versa.

Fixing a tuple t, we have to distinguish between five cases for each tuple version (summand) k in the annotation of tuple t as shown below. Figure. A.1 shows the versions of a tuple version with an identifier id in $R[T, \nu_{n+1}]$ and $R[\nu_{n+1}]$ for each of the cases.

1. k is the latest version of all tuple versions with identifier $idOf(k)$ and was created by Transaction T before ν_{n+1}. In this case k is only present in $R[T, \nu_{n+1}](t)$. For this case we assume that the first tuple version with identifier $idOf(k)$ was

created by an insert of Transaction T. Thus, there cannot exist an outdated version k' with this identifier in the annotation of any tuple t' in $R[\nu_{n+1}]$.

2. k is the latest version of all tuple versions with identifier $idOf(k)$ and was created by a Transaction T before ν_{n+1}. In this case k is only present in $R[T, \nu_{n+1}](t)$. The previous tuple version with identifier $idOf(k)$ was created by a Transaction $T' \neq T$. Hence, there has to exist an outdated version k' with this identifier in the annotation of some tuple t' in $R[\nu_{n+1}]$.

3. k is the latest version of all tuple versions with identifier $idOf(k)$ and was created by a Transaction T' that committed after $Start(T)$, but before ν_{n+1}. In this case k is only present in $R[\nu_{n+1}](t)$. For this case we assume that the previous tuple version with identifier $idOf(k)$ was created by an insert of Transaction $T' \neq T$. Thus, there cannot exist an outdated version k' with this identifier in the annotation of any tuple t' in $R[T, \nu_{n+1}]$.

4. k is the latest version of all tuple versions with identifier $idOf(k)$ and was created by a Transaction T' that committed after $Start(T)$, but before ν_{n+1}. In this case k is only present in $R[\nu_{n+1}](t)$. The first tuple version with identifier $idOf(k)$ was created by an insert of a Transaction $T'' \neq T$ where $End(T'') < \nu_{n+1}$. Hence, there has to exist an outdated version k' with this identifier in the annotation of some tuple t' in $R[T, \nu_{n+1}]$.

5. k is the latest version of all tuple versions with identifier $idOf(k)$ and was created by a Transaction T' that committed before $Start(T)$. In this case k is present in both $R[T, \nu_{n+1}](t)$ and $R[\nu_{n+1}](t)$.

Case 1: Since k is the only summand with identifier id in $R[T, \nu_{n+1}]$ and does not occur in $R[\nu_{n+1}]$, function $isMax(\ R[\nu_{n+1}], k)$ returns 1 and k is in $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Similarly, since k is the latest version, we have that VALIDIN($R[T, \nu_{n+1}], t, k, \nu_{n+1})$ re-

Occurrence of summand with identifier id

Case	$R[\nu_{n+1}]$	$R[T, \nu_{n+1}]$
1	none present	latest version k with $idOf(k) = id$
2	outdated version k' with $idOf(k) = id$	latest version k with $idOf(k) = id$
3	latest version k with $idOf(k) = id$	none present
4	latest version k with $idOf(k) = id$	outdated version k' with $idOf(k) = id$
5	latest version k with $idOf(k) = id$	latest version k with $idOf(k) = id$

Figure A.1. Cases of how tuple versions with a fixed identifier id can occur in $R[\nu_{n+1}]$ and $R[T, \nu_{n+1}]$

turns 1 because k has a version annotation from T as the outmost version annotation. Thus, k is also present in $R_{ext}[T, \nu_{n+1}]$.

Case 2: Summand k is the only summand with identifier id in $R[T, \nu_{n+1}]$. While there exists a summand k' with identifier id in the annotation of some tuple t' in $R[\nu_{n+1}]$, we know that $\nu(k') < \nu(k)$. Thus, function $isMax(R[\nu_{n+1}], k)$ returns 1 and k is in $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Function VALIDIN($R[T, \nu_{n+1}]$) returns 1 for the same reason as in case 1 above.

Now consider summand k' with $idOf(k') = id$ that occurs as a summand in the annotation of tuple t' in $R[\nu_{n+1}]$. We have to show that both $isStrictMax$ and VALIDEX return 0 for this outdated tuple version. $isStrictMax(R[T, \nu_{n+1}], k') = 0$, because the summand k occurs in $R[T, \nu_{n+1}](t)$, $idOf(k) = idOf(k')$, and $versionOf(k) > versionOf(k')$. Also VALIDEX($R[\nu_{n+1}], t', k', \nu_{n+1}$) returns 0, because UPDATED(T, t', k', ν_{n+1}) evaluates to true.

Case 3: Since k is the only summand with identifier id in $R[\nu_{n+1}]$ and does not occur in $R[T, \nu_{n+1}]$, function $isStrictMax$ $(R[T, \nu_{n+1}], k)$ returns 1 and k is in

$\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Similarly, since k is the latest version of a tuple version with identifier $idOf(k)$, we have that VALIDEX($R[\nu_{n+1}], t, k, \nu_{n+1})$ returns 1 because UPDATED(T, t, k, ν_{n+1}) evaluates to false. Thus, k is also present in $R_{ext}[T, \nu_{n+1}]$.

Case 4: Summand k is the only summand with identifier id in $R[\nu_{n+1}]$. While there exists a summand k' with identifier id in the annotation of some tuple t' in $R[T, \nu_{n+1}]$, we know that $\nu(k') < \nu(k)$. Thus, function $isStrictMax(R[T, \nu_{n+1}], k)$ returns 1 and k is in $\mu(\mathbb{R}(u_n), R[\nu(u_{n+1})])(t)$. Function VALIDEX$(R[\nu_{n+1}])$ returns 1 for the same reason as in case 3 above.

Now consider summand k' with $idOf(k') = id$ that occurs as a summand in the annotation of tuple t' in $R[T, \nu_{n+1}]$. We have to show that both $isMax$ and VALIDIN return 0 for this outdated tuple version. $isMax(R[\nu_{n+1}], k') = 0$, because there is summand k in $R[\nu_{n+1}](t)$, $idOf(k) = idOf(k')$, and $versionOf(k) > versionOf(k')$. Also VALIDIN$(R[T, \nu_{n+1}],\ t', k', \nu_{n+1})$ returns 0, because k does not have a version annotation from T as its outermost version annotation.

Case 5: Summand k was created by a Transaction T' with $End(T') < Start(T)$. Thus, k is present in both $R[T, \nu_{n+1}](t)$ and $R[\nu_{n+1}](t)$ and based on the definition of this case no other summand k' with $idOf(k) = idOf(k')$ occurs in the annotation of any tuple t' in $R[T, \nu_{n+1}](t)$ or $R[\nu_{n+1}](t)$. Thus, $isMax(R[\nu_{n+1}], k)$ returns 1 because there is no newer version of k in $R[\nu_{n+1}]$ while $isStrictMax(R[T, \nu_{n+1}], k)$ returns 0, because there exists k in $R[T, \nu_{n+1}]$. Similarly, VALIDIN returns 0 because T has not created tuple version k whereas VALIDEX evaluates to 1, because T has not updated k.

Having proven all cases we get:

$$\sum_{i=0}^{n(R[T,\nu_{n+1}](t))} R[T,\nu_{n+1}](t)[i] \times isMax(R[\nu_{n+1}], R[T,\nu_{n+1}](t)[i])$$
$$+ \sum_{i=0}^{n(R[\nu_{n+1}](t))} R[\nu_{n+1}](t)[i] \times isStrictMax(R[T,\nu_{n+1}], R[\nu_{n+1}](t)[i])$$
$$= \sum_{i=0}^{n(R[T,\nu_{n+1}](t))} R[T,\nu_{n+1}](t)[i] \times \textsc{validIn}(T, t, R[T,\nu_{n+1}](t)[i], \nu_{n+1})$$
$$+ \sum_{i=0}^{n(R[\nu_{n+1}](t))} R[\nu_{n+1}](t)[i] \times \textsc{validEx}(T, t, R[\nu_{n+1}](t)[i], \nu_{n+1})$$

Reordering the two sums we get

$$= \sum_{i=0}^{n(R[\nu_{n+1}](t))} R[\nu_{n+1}](t)[i] \times \textsc{validEx}(T, t, R[\nu_{n+1}](t)[i], \nu_{n+1})$$
$$+ \sum_{i=0}^{n(R[T,\nu_{n+1}](t))} R[T,\nu_{n+1}](t)[i] \times \textsc{validIn}(T, t, R[T,\nu_{n+1}](t)[i], \nu_{n+1})$$
$$= R_{ext}[T,\nu_{n+1}](t)$$
$$= R[T,\nu_{n+1}](t)$$

Thus, we have shown that $T \equiv_{\mathbb{N}[X]^{\nu}} \mathbb{R}(T)$.

$\underline{T \equiv_{\mathbb{N}[X]^{\nu}} \mathbb{R}_{opt}(T)}$: Let $T = u_1, \ldots, u_n, c$ be a transaction in a RC-SI history H. Recall that $\mathbb{R}_{opt}$ is evaluated over $R[End(T)-1]$. As shown in Lemma 23 for any given tuple identifier id, $R[End(T)-1]$ contains the predecessor of the earliest version of a tuple with identifier id created by Transaction T (if such a tuple version exists). Thus, the reenactment is correct as long as the following three conditions hold: 1) the first update in T that creates a new version of with identifier id updates this version in $\mathbb{R}_{opt}$; 2) the reenactment query for each update $u_i \in T$ is not applied to any tuple version k from $R[End(T)-1]$ with $\nu(k) > \nu(u_i)$; and 3) each tuple version k is passed on by the reenactment query for each u_j with $\nu(k) > \nu(u_j)$.

We prove this by induction over the position of an update in $T = u_1, \ldots, u_n, c$.

Induction Start: Consider u_1, the first update of T. Update u_1 is either an update, delete, or simple insert (the insert's query is a singleton operator $\{t \rightarrow k\}$). Let ν_1 denote $\nu(u_1)$ and ν_e to denote $End(T) - 1$.

u_1 is an update: First consider the case where u_1 is an update. The part of the reenactment query for T corresponding to u_1 is

$$\alpha_{U,T,\nu_1+1}(\Pi_A(\sigma_\theta(\gamma_{V \leq \nu_1}(R[\nu_e])))) \cup \sigma_{\neg\theta}(\gamma_{V \leq \nu_1}(R[\nu_e])) \cup \gamma_{V > \nu_1}(R[\nu_e]))$$

Based on Lemma 23, $R[\nu_e]$ contains all versions of tuples that got updated by u_1. Consider a tuple version k in the annotation of a tuple t in $R[\nu_e]$. Depending on whether $\nu(k) \leq \nu_1$ holds or not, this tuple version will be visible to u_1 or not. If k is visible to u_1 then whether k will be updated depends on whether t fulfills the update's condition or not. If $\nu(k) > \nu_1$ then k will not fulfill the condition $V \leq \nu_1$ of the version filter operators in the first two branches of the union. Tuple version k fulfills the condition of the third branch ($V > \nu_1$) and, thus, will be passed on unmodified to the output of the part of the reenactment query corresponding to u_1. This implies that the second and third correctness conditions introduced above hold (non-visible tuple versions are not updated and passed on unmodified). If k was visible to u_1 and was updated by u_1, then we know that $\nu(k) \leq \nu_1$. Thus, k fulfills the condition $V \leq \nu_1$ of the version filter operator in the first two branches of the union, but only fulfills the selection condition (θ) of the first branch of the union and, thus, is updated (first condition). Note that if k was visible to u_1, but was not updated by u_1 then either k will be "routed" through the second branch of the union (if k is the latest version of a tuple with identifier $idOf(k)$ present in $R[\nu_e]$) or k will not be in $R[\nu_e]$ (if $R[\nu_e]$ contains a newer version of a tuple with identifier $idOf(k)$).

u_1 is a delete: The part of the reenactment query for T corresponding to a delete u_1

is

$$\alpha_{D,T,\nu_1+1}(\sigma_\theta(\gamma_{V\leq\nu_1}(R[\nu_e]))) \cup \sigma_{\neg\theta}(\gamma_{V\leq\nu_1}(R[\nu_e])) \cup \gamma_{V>\nu_1}(R[\nu_e])$$

Consider a tuple version k in the annotation of a tuple t in $R[\nu_e]$. Note that the third branch of the union is identical for updates and deletes. Hence, if $\nu(k) \leq \nu_1$, the second and third conditions hold. The cases where k is affected by u_1 or k is visible, but not affected, are also analog to the proof for updates.

u_1 is a simple insert: The part of the reenactment query for T corresponding to a delete u_1 is

$$\alpha_{I,T,\nu_1+1}(\{t \rightarrow k\}) \cup R[\nu_e]$$

All tuples from $R[\nu_e]$ are present in the output (second and third condition) and, since an insert creates new tuple versions, the first condition trivially holds.

Induction Step: We have to show that under the assumption that updates u_j with $j \leq i$ are reenacted correctly by $\mathbb{R}_{opt}$, then the same holds for u_{i+1}. Let ν_{i+1} denote $\nu(u_{i+1})$. Again this has to be shown for the three cases of u_{i+1} being an 1) update, 2) delete, or 3) simple insert. Observe that the input to the part of the reenactment query corresponding to u_j is equal to $T[\nu_e]$ except that some tuple versions have been replaced by updated tuple versions by the part of the reenactment query corresponding to updates u_1 to u_i. Since this is the only difference to the induction start, we only have to prove this additional case. Consider such a version k of tuple t produced by u_j with $j \leq i$. It follows that $\nu(k) \leq \nu_{i+1}$. Thus, k fulfills the conditions of the first two branches of the union for updates and deletes. Based on the induction hypothesis, if u_{i+1} produces a tuple with identifier $idOf(k)$ then k is the previous version of this tuple. Thus, the update's respective deletion's condition θ evaluates to true for t and k will be updated respective deleted. If k does not fulfill the condition

then the second branch of the union passes on k unmodified. It follows that u_{i+1} is correctly reenacted by $\mathbb{R}_{opt}(T)$. □

Note that based on the results of [6] equivalence under $\mathbb{N}[X]^\nu$ implies equivalence under any naturally ordered MV-semiring $\mathcal{K}^\nu$. Furthermore, it was proven [6] that if $\mathcal{K}$ is naturally ordered, then so is $\mathcal{K}^\nu$. The first result follows from commutation of queries and transactional histories with lifted homomorphisms. Based on Lemmas 20, 21 and 22 such homomorphisms also commute with the new query operators we have introduced and RC-SI histories. Thus, $\mathbb{N}[X]^\nu$ implies equivalence under any naturally ordered MV-semiring $\mathcal{K}^\nu$ for any of the operations used in this work.

APPENDIX B

EXAMPLES

B.1 First Provenance Capture Example

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

Example 34. *Consider a bank that stores information about US accounts in a relation* `USacc` *and Canadian accounts in relation* `CanAcc`*. Example instances for these relations are shown in Figure B.1. The bank decides to give free US accounts to Canadian customers that already have an US dollar account (*`Type = 'US␣dollar'`*) with the Canadian branch. The new accounts in relation* `USaccs` *are created from the data of the corresponding* `CanAcc` *tuples. These newly created accounts should get "premium" status if their balance is over 1,000,000 US Dollar. Finally, "standard" accounts with low balance (below 100 US Dollar) should be removed as it is the bank's policy to require a minimum 100 dollar balance for "standard" accounts in the US. A transaction implementing these changes is shown in Figure B.4 (left hand side). Assume that this transaction has been run under isolation level* `SERIALIZABLE` *(recall that this is actually snapshot isolation) and was assigned the transaction identifier (xid)* `0A0202F5`*. Figure B.2 shows the content of the audit log produced by running this transaction. Recall that our prototype implementation uses Oracle's fine grained auditing feature for the audit log. Oracle stores the audit log in a relation called* `fga_log$`*. An internal version counter called the system change number (SCN) is used to represent versions. Note that all statements are assigned the same SCN in the example, because in a* `SERIALIZABLE` *transaction each statement sees the same snapshot (version) of the database modulo changes of previous statements of the same transaction.*

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

USacc

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

CanAcc

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

Figure B.1. Example Instance For Running Example

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

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

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

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

Translate updates. In the next step, we generate an individual reenactment query

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

Figure B.2. Example Audit Log

$q(u_i)$ for each update u_i of the transaction. The translation for updates has been explained in Section 6.2. An insert statement adds new tuples to the relation. This is simulated by computing the union of the version of the relation before the insert and the newly inserted tuples. A delete retains the unmodified versions of all tuples that do not fulfill the `WHERE` clause condition. The reenactment queries generated for the running example are shown on the right hand side of Figure B.4.

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

```
WITH
u1 AS
(SELECT ID, Owner, Balance, 'Standard' AS Type
FROM CanAcc AS OF SCN 3652
```

```
WHERE Type = 'US␣dollar'
UNION ALL
SELECT * FROM USacc AS OF SCN 3652),

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

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

Rewrite For Provenance Computation. The generated reenactment query is then rewritten for provenance computation according to the type of provenance that was requested. The query shown below has been rewritten to compute PI-CS provenance [40]. Note that the user has requested that only tuples modified by the transaction should be returned. This is realized by propagating an attribute `updated` which is set to 1 for updated tuples and to 0 for tuples which have not been updated. This attribute is used to select only updated tuples.

```
WITH
u1 AS
(SELECT ID, Owner, Balance, 'Standard' AS Type,
        ID AS prov_CanAcc_ID,
        Owner AS  prov_CanAcc_Owner,
        Balance AS prov_CanAcc_Balance,
```

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

u2 AS
(SELECT ID, Owner, Balance, 'Premium' AS Type,
        prov_CanAcc_ID,
        prov_CanAcc_Owner,
        prov_CanAcc_Balance,
        prov_CanAcc_Type,
```

```
        prov_USacc_ID,
        prov_USacc_Owner,
        prov_USacc_Balance,
        prov_USacc_Type
        1 AS updated
FROM u1
WHERE Balance > 1000000
UNION ALL
SELECT * FROM u1
WHERE (Balance > 1000000) IS NOT TRUE)

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

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

```
...
```

```
u2 AS
(SELECT ID, Owner, Balance,
        CASE
          WHEN Balance > 1000000 THEN 'Premium'
          ELSE Type
        END AS Type,
        prov_CanAcc_ID,
        prov_CanAcc_Owner,
        prov_CanAcc_Balance,
        prov_CanAcc_Type,
        prov_USacc_ID,
        prov_USacc_Owner,
        prov_USacc_Balance,
        prov_USacc_Type,
        1 AS updated
FROM u1)
...
```

Executing the Rewritten Query. Figure B.5 shows the result of executing the rewritten query. In the result, the updated version of each tuple produced by the transaction is paired with its provenance in relations `USacc` and `CanAcc`. Full attribute names for provenance attributes are shown in Figure B.6.

B.2 Second Provenance Capture Example

We explain the whole process of capturing provenance for transactions using reenactment based on a simple example.

Example 35. *Alice, a developer working for the HR department at a company, has*

executed a transaction under the snapshot isolation (SI) *concurrency control protocol. The transaction inserts a new employee, "Mark Smith" who is a "Software Engineer" and also increases the bonus for all employees that hold a position as Software Engineer by $500 (shown in Figure 4.5). Per company policy every new employee should receive a basic bonus of $500 which does not include the extra $500 bonus for being a software engineer. However, Alice made a mistake and accidentally did set Mark's bonus to $0. After a while, Mark Smith discovers that he has only received $500 instead of $1000 bonus.* `employee` *relation state before the execution of the transaction shown in Figure 4.6 and after in Figure 4.7. To figure out why Mark has not received the correct amount of bonus payments, Alice requests the provenance for this transaction.*

We now show how Alice's provenance request is processed and then show how the provenance captured helps Alice to understand what has happened.

Querying the Audit Log. When Alice sends her provenance request, the system gathers information about the transaction from the audit log. It determines which statements were executed by the transaction, at what time, and also determines under which isolation level the transaction was executed (isolation level `SERIALIZABLE` which corresponds to SI in systems like, e.g., Oracle).

Overview. Here we only show how the construction of an SQL query that returns the relational encoding of an MV-semiring relation. The major difference between this process and MV-semiring reenactment is implemented in 3 steps:

1. In a first step we construct a reenactment query for each statement of the transaction. The difference to reenactment of updates according to Definition 6 is that these queries are not instrumented for provenance capture.

2. Next, the reenactment query is constructed for the transaction from the reenactment queries of its statements. This step is analog to the construction from Definition 7 with the exception that the query is not instrumented for provenance tracking. That is, the query produced by this step returns the updated state of the relation produced by the transaction for which we are capturing provenance.

3. In a final step, the reenactment query for the transaction is instrumented for provenance capture, i.e., to return the relational encoding of the $\mathbb{N}[X]^{\nu}$-relation that is the result of this transaction. For details about the relational encoding and the rewriting employed to instrument the query we refer the interested reader to [6].

Step 1 - Translate updates. Based on the transaction information gathered from the audit log, GProM generates the reenactment query $\mathbb{R}(u_i)$ for each update u_i of the transaction. An insert statement computes the union of the previous state of the relation and the set of tuples to be inserted. Thus, $\mathbb{R}(u_1)$, the reenactment query for the insert statement of the transaction, simulates the insert by computing the union of the version of the relation before the insert and the newly inserted tuples. $\mathbb{R}(u_2)$, the reenactment query for the second statement of the transaction, computes the union of the modified tuples of the input relation that match the condition of the update and the set of tuples that do not match the condition (these tuples are not modified). Here, `AS OF` denotes the use of time travel to retrieve a past version of a relation. Suppose Alice's transaction did start at time '2017-01-08'. The reenactment queries for u_1 and u_2 are shown below.

$\mathbb{R}(u_1)$:
```
SELECT 'Mark␣Smith' as name,
       'Software␣Engineer' as position,
       0 as bonus
```

```
        FROM dual
        UNION ALL
        SELECT * FROM employee AS OF '2017-01-08';
```

$\mathbb{R}(u_2)$:
```
SELECT name, position, bonus+500 AS bonus
        FROM employee AS OF '2017-01-08'
        WHERE position = 'Software␣Engineer'
        UNION ALL
        SELECT *
        FROM employee AS OF '2017-01-08'
        WHERE (position = 'Software␣Engineer') IS NOT TRUE;
```

Step 2 - Construct Transaction Reenactment Query. To reenact the transaction, we merge the reenactment queries for updates of the transaction in a way that respects the visibility rules enforced by the concurrency control protocol. The individual update reenactment queries are merged into a reenactment query for the transaction by recursively replacing in each reenactment query $q(u_i)$ all accesses to relation `employee` with the query $q(u_{i-1})$ producing the version of the relation read by u_i. The result of this process is shown below.

```
WITH
u1 AS
(SELECT SELECT 'Mark␣Smith' as name,
      'Software␣Engineer' as position,
      0 as bonus
      FROM dual
      UNION ALL
      SELECT * FROM employee AS OF '2017-01-08'),
```

```
u2 AS
(SELECT name, position, bonus+500 AS bonus
      FROM u1
      WHERE position = 'Software␣Engineer'
      UNION ALL
      SELECT *
      FROM u1
      WHERE (position = 'Software␣Engineer') IS NOT TRUE);

SELECT * FROM u2;
```

Step 3 - Rewrite For Provenance Capture. The generated reenactment query is then rewritten for provenance capture. The rewritten query for the example is shown below. Recall that in the relational encoding of MV-semiring provenance, the boolean attribute $\mathcal{U}_i$ stores whether u_i affected a tuple. $\mathcal{U}_i$ is set to *True* for updated tuples and to *False* for tuples which have not been updated. The user can choose whether the generated query should show intermediate states of relations as seen by the updates of the transaction or not. Assume that Alice has specified in her provenance request that intermediate states should be shown. The result of this query is shown in Figure B.7. Note how attributes `P(name,`u_2`)`, `P(position,`u_2`)`, and `P(bonus,`u_2`)` show the intermediate state of relation employee as produced by the execution of first statement of Alice's transaction. This is the version seen by the second update of her transaction.

```
WITH
u1 AS
(SELECT 'Mark␣Smith' AS name,
```

```
        'Software␣Engineer' AS position,
        0 As bonus,
        NULL AS P(name,u1),
        NULL AS P(position,u1),
        NULL AS P(bonus,u1),
        True AS U1
FROM dual
UNION ALL
SELECT name, position, bonus,
       name AS P(name,u1),
       position AS  P(position,u1),
       bonus AS P(bonus,u1),
       False AS U1
FROM employee AS OF '2017-01-08'),

u2 AS
(SELECT name, position, bonus+500 AS bonus,
        P(name,u1),
        P(position,u1),
        P(bonus,u1),
        name AS P(name,u2),
        position AS  P(position,u2),
        bonus AS P(bonus,u2),
        U1,
        True AS U2
FROM u1
WHERE position = 'Software␣Engineer'
```

```
UNION ALL
SELECT name, position, bonus,
       P(name,u1),
       P(position,u1),
       P(bonus,u1),
       name AS P(name,u2),
       position AS  P(position,u2),
       bonus AS P(bonus,u2),
       U1,
       False AS U2
FROM u1
WHERE (position = 'Software Engineer') IS NOT TRUE);

SELECT * FROM u2;
```

Optimizations. GProM implements optimizations for instrumented queries. This optimizations transforms a SQL query which implements provenance capture into a query that can be successfully optimized by the backend database system. We present some optimization techniques that are specific to reenactment in Section 7. For instance, one of these optimizations reduces the number of set operations in the reenactment query by using an alternative way of reenacting `UPDATE` statements. Instead of computing the union between the updated tuples and the tuples that did not get updated, we use the SQL `CASE` construct to check the update's condition for each input tuple and only modify attribute values for tuples which fulfill this condition. The common table expression u_2 in the previous query can be optimized as follows:

...

```
u2 AS
(SELECT name, position,
        CASE
          WHEN position = 'Software␣Engineer' THEN bonus+500
          ELSE bonus
        END AS bonus,
        P(name,u1),
        P(position,u1),
        P(bonus,u1),
        name AS P(name,u2),
        position AS  P(position,u2),
        bonus AS P(bonus,u2),
        U1,
        CASE
          WHEN position = 'Software␣Engineer' THEN True
          ELSE False
        END AS U2
FROM u1)
...
```

Note that in this example Alice has requested GProM to return all tuples of relation employee - no matter whether affected by her transaction or not. If she is only interested in tuples affected by her transaction, then GProM would apply one of the optimizations described in Section 7 in the main paper to filter out tuples that will not be part of the result early on. As an example, consider the optimization "Prefiltering with Update Conditions". This optimization applies a selection condition to the input of reenactment that is the disjunction of the conditions of all updates of the

transaction. It is applicable as long as the transaction consists only of updates, simple inserts (`VALUES` clause), and inserts with queries that do not access any of the relations affected by the update statement. Since Alice's transaction fulfills this condition, we can apply this optimization. Her transaction contains a single update with a condition `position = 'Software␣Engineer'`. Applying the optimization we get a modified version of `u1` (changes are shown in red):

```
WITH
u1 AS
(SELECT 'Mark␣Smith' AS name,
        'Software␣Engineer' AS position,
        0 As bonus,
        NULL AS P(name,u1),
        NULL AS P(position,u1),
        NULL AS P(bonus,u1),
        True AS U1
FROM dual
UNION ALL
SELECT name, position, bonus,
       name AS P(name,u1),
       position AS  P(position,u1),
       bonus AS P(bonus,u1),
       False AS U1
       FROM employee AS OF '2017-01-08'
       WHERE position = 'Software Engineer'),
...
```

Step 4 - Executing the Rewritten Query. Figure B.7 shows the result of execut-

ing the rewritten query. In the result, the updated version of each tuple produced by the transaction is paired with its provenance in relations `employee`. The provenance annotation of each tuple is encoded in additional attributes that are added to the schema. A *"provenance attribute"* `P(attr, ,`u_i`)` stores the value of attribute *attr* for the version of a tuple in the provenance seen by the update statement i. For instance, attributes `P(bonus,`u_1`)` and `P(bonus,`u_2`)` store the bonus attribute values of a tuple before the execution of updates u_1 (respective u_2). The boolean attribute $\mathcal{U}_i$ stores whether the version annotation for update u_i is part of the provenance, i.e., whether u_i affected a tuple.

Example 36. *Figure B.7 shows the result for Alice's provenance request. The third tuple shows provenance for the tuple storing Mark Smith's information. Based on the result, Alice can determine that both statements of the transaction affected this tuple as both* $\mathcal{U}_1$ *and* $\mathcal{U}_2$ *are True. The result also unearths that the bonus of the tuple as inserted by the first statement of the transaction (*`P(bonus,`u_2`)`*) is 0. That is, in the intermediate state of the relation* `employee` *which is the input for the second update, the initial bonus for Mark Smith was 0. Thus, Alice has discovered the cause of the erroneous bonus payment for Mark.*

To avoid replaying the whole history, our approach applies time travel to start replay from the database version valid at the start of a transaction. Note that time travel does not provide access to uncommitted, intermediate versions of relations as seen by the updates of a transaction. Thus, time travel would not help Alice to find the cause of the error, since the 0 bonus for Mark is not contained in any committed database version. In this particular example, the audit log reveals the error in the insert statement. However, in general that is not the case. For instance, if Alice's transaction would have used an update to set the bonus after the insert using some complex `WHERE`-clause condition, then the error would not have been directly

observable in the audit log. Even though in some cases it is technically possible to find the cause of an error using time travel or audit logging alone, by showing intermediate versions and exposing tuple dependencies as well as dependencies of tuples on updates, our MV-semiring approach greatly simplifies debugging for these cases.

USacc after u_1

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

USacc after u_2

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

USacc after u_3

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

Figure B.3. Updated Example Instances

Transaction	**Update Reenactment Queries**
u_1:	$q(u_1)$:

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

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

u_2:

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

$q(u_2)$:

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

u_3:

```
DELETE FROM USacc
WHERE Balance < 100;
```

$q(u_3)$:

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

Figure B.4. Example Transaction and Translated Updates

Result of Example Provenance Query

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

Figure B.5. Provenance for the Running Example Transaction

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

Figure B.6. Provenance Attribute Names

employee			Provenance for the First Update			Provenance for the Second Update			u_1	u_2
name	**position**	**bonus**	**P(name,u_1)**	**P(position,u_1)**	**P(bonus,u_1)**	**P(name,u_2)**	**P(position,u_2)**	**P(bonus,u_2)**	$\mathcal{U}_1$	$\mathcal{U}_2$
Susan Sommers	Software Engineer	1000	Susan Sommers	Software Engineer	500	Susan Sommers	Software Engineer	500	False	True
David Spears	Test Assurance	500	David Spears	Test Assurance	500	David Spears	Test Assurance	500	False	False
Mark Smith	Software Engineer	500	NULL	NULL	NULL	Mark Smith	Software Engineer	0	True	True

Figure B.7. Relational encoding of the provenance and intermediate results for relation employee

BIBLIOGRAPHY

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

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

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

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

[5] L. Antova, C. Koch, and D. Olteanu. From complete to incomplete information and back. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 713–724. ACM, 2007.

[6] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Formal foundations of reenactment and transaction provenance. Technical report, IIT, 2016.

[7] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, pages 841–850, 2016.

[8] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation (long version). *CoRR*, abs/1608.08258, 2016.

[9] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Using reenactment to retroactively capture provenance for transactions. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):599–612, 2018.

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

[11] 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.

[12] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *VLDB*, pages 220–231, 2000.

[13] 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.

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

[15] N. Bidoit, M. Herschel, and A. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 713–722. ACM, 2015.

[16] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with nedexplain. In *Extending database technology (EDBT)*, 2014.

[17] P. Bourhis, D. Deutch, and Y. Moskovitch. Analyzing data-centric applications: Why, what-if, and how-to. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 779–790. IEEE, 2016.

[18] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. *SIGARCH Comput Archit News*, 42(1):239–254, 2014.

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

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

[21] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *ICDT*, pages 177–188. ACM, 2013.

[22] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[23] A. Chapman and H. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 523–534. ACM, 2009.

[24] J. Cheney. Program slicing and data provenance. *IEEE Data Eng. Bull.*, 30(4):22–28, 2007.

[25] 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.

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

[27] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

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

[29] D. Deutch and A. Gilad. Qplain: Query by explanation. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1358–1361. IEEE, 2016.

[30] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[31] D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for datalog provenance. In *ICDT*, pages 201–212, 2014.

[32] D. Deutch, Y. Moskovitch, and V. Tannen. Provenance-based analysis of data-centric processes. *The VLDB Journal*, 24(4):583–607, 2015.

[33] M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva. Supporting annotations on relations. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 379–390. ACM, 2009.

[34] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[35] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *Provenance and Annotation of Data*, pages 10–18. Springer, 2006.

[36] F. Geerts, A. Kementsietsidis, and D. Milano. imondrian: A visual tool to annotate and query scientific databases. In *International Conference on Extending Database Technology*, pages 1168–1171. Springer, 2006.

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

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

[39] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul. Efficient stream provenance via operator instrumentation. *Transactions on Internet Technology (TOIT)*, 14(1):7:1–7:26, 2014.

[40] 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.

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

[42] T. J. Green and V. Tannen. The semiring framework for database provenance. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 93–99. ACM, 2017.

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

[44] I. Gupta, V. Raghavan, M. Ghosh, et al. Leveraging metadata in no sql storage systems. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 57–64. IEEE, 2015.

[45] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB JournalThe International Journal on Very Large Data Bases*, 26(6):881–906, 2017.

[46] T. Imieliński and W. Lipski. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1988.

[47] T. Imieliński and W. Lipski Jr. Incomplete Information in Relational Databases. *JACM*, 31(4):761–791, 1984.

[48] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 687–700. ACM, 2008.

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

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

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

[52] O. Kennedy and C. Koch. Pip: A database system for great and small expectations. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 157–168. IEEE, 2010.

[53] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[54] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014.

[55] 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.

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

[57] D. Kulkarni. A provenance model for key-value systems. In *Presented as part of the 5th {USENIX} Workshop on the Theory and Practice of Provenance*, 2013.

[58] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*, pages 575–586, 2014.

[59] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[60] X. Niu, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, O. Kennedy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, pages 49–53. USENIX Association, 2016.

[61] X. Niu, B. S. Arab, S. Lee, S. Feng, X. Zou, D. Gawlick, V. Krishnaswamy, Z. H. Liu, and B. Glavic. Debugging transactions and tracking their provenance with reenactment. *Proceedings of the VLDB Endowment*, 10(12), 2017 (to appear).

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

[63] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *Proceedings of the VLDB Endowment*, 11(6):719–732, 2018.

[64] S. Riddle, S. Köhler, and B. Ludäscher. Towards constraint provenance games. In *TaPP*, 2014.

[65] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat. Provsql: Provenance and probability management in postgresql. *Proceedings of the VLDB Endowment*, 11(12):2034–2037, 2018.

[66] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997.

[67] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *International Journal of Web Services Research (IJWSR)*, 5(2):1–22, 2008.

[68] D. Srivastava and Y. Velegrakis. Using queries to associate metadata with data. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1451–1453. IEEE, 2007.

[69] S. Vansummeren and J. Cheney. Recording Provenance for SQL Queries and Updates. *Data Eng. Bull*, 30(4):29–37, 2007.

[70] X. Wang, A. Meliou, and E. Wu. Qfix: Diagnosing errors through query histories. In *SIGMOD*, pages 1369–1384, 2017.

[71] M. Weiser. Program slicing. *ICSE*, pages 439–449, 1981.

[72] J. Xu, W. Zhang, A. Alawini, and V. Tannen. Provenance analysis for missing answers and integrity repairs. *IEEE Data Eng. Bull.*, 41(1):39–50, 2018.

[73] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: An on-demand approach to etl. *Proceedings of the VLDB Endowment*, 8(12):1578–1589, 2015.

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

[75] 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.

[76] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *SIGMOD Record*, 24(2):316–327, 1995.