Efficient Answering of Historical What-if Queries

Felix S. Campbell
Illinois Institute of Technology
USA, Chicago
fcampbell@hawk.iit.edu

Bahareh Sadat Arab
Intuit
USA, San Diego
barab@hawk.iit.edu

Boris Glavic
Illinois Institute of Technology
USA, Chicago
bglavic@iit.edu

ABSTRACT

We introduce *historical what-if queries*, a novel type of what-if analysis that determines the effect of a hypothetical change to the transactional history of a database. For example, *"how would revenue be affected if we would have charged an additional $6 for shipping?"* We develop efficient techniques for answering historical what-if queries, i.e., determining how a modified history affects the current database state. Our techniques are based on *reenactment*, a replay technique for transactional histories. We optimize this process using program and data slicing techniques that determine which updates and what data can be excluded from reenactment without affecting the result. Using an implementation of our techniques in *Mahif* (a Middleware for Answering Historical what-IF queries) we demonstrate their effectiveness experimentally.

CCS CONCEPTS

• **Information systems → Data provenance**.

KEYWORDS

what-if queries, updates, provenance, program slicing

ACM Reference Format:
Felix S. Campbell, Bahareh Sadat Arab, and Boris Glavic. 2022. Efficient Answering of Historical What-if Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526138

1 INTRODUCTION

What-if analysis [6, 19] 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 company's 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 insights about 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* (*HWQ*), a novel type of what-if queries where the user postulates a hypothetical change to the transactional history of the database.

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	5	o_1
12	Alex	UK	50	5	o_2
13	Jack	US	60	3	o_3
14	Mark	US	30	4	o_4

Figure 1: Running example database instance.

U	SQL
u_1	`UPDATE Order SET ShippingFee=0 WHERE Price>=50;`
$u_1{}'$	`UPDATE Order SET ShippingFee=0 WHERE Price>=60;`
u_2	`UPDATE Order SET ShippingFee=ShippingFee+5 WHERE Country='UK'AND Price <=100;`
u_3	`UPDATE Order SET ShippingFee=ShippingFee-2 WHERE Price <=30 AND ShippingFee>=10;`

Figure 2: History H implementing the shipping fee policy and a hypothetical change of the policy (update $u_1{}'$ replaces u_1 to raise the price for waiving shipping fees to $60).

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	8	o_5
12	Alex	UK	50	5	o_6
13	Jack	US	60	0	o_7
14	Mark	US	30	4	o_8

Figure 3: Result of executing the original history H.

Order

ID	Customer	Country	Price	ShippingFee	
11	Susan	UK	20	8	o_5
12	Alex	UK	50	10	o_6'
13	Jack	US	60	0	o_7
14	Mark	US	30	4	o_8

Figure 4: Result of executing the hypothetical history $H[\mathcal{M}]$.

Example 1. *Consider an online retailer that has developed a new shipping fees policy. An example database instance is shown in Fig. 1. Fig. 2 shows a transactional history with three updates u_1, u_2 and u_3 that implement this policy which resulted in the database state shown in Fig. 3. For example, u_1 waives shipping fees for orders of at least $50. Bob, an analyst, wants to understand how a larger order price threshold for waiving shipping fees, say $60, would have affected revenue. Bob's request can be expressed as a* historical what-if query *which replaces the update u_1 with update $u_1{}'$ (highlighted in red in Fig. 2). Fig. 4 shows the new state of the database after executing the modified transactional history over the database from Fig. 1. The hypothetical change results in an increase of the shipping fee for the record with ID 12 (highlighted in red). By evaluating the effect of changing a past action (an update) instead of changing the current state of the database as in classical what-if analysis, the answer to a historical what-if query can inform future actions. For example, if revenue is increased significantly by using a $60 cutoff for waiving shipping fees, then we may apply this higher threshold in the future.*

In this paper, we study how to efficiently answer historical what-if queries (HWQs) such as the one from Ex. 1. A HWQ $\mathcal{H}$ is a triple $(H, D, \mathcal{M})$ where H is a transactional history (a sequence of insert/update/delete statements), D is the state of the database before the execution of the transactional history H, and $\mathcal{M}$ is a set of modifications to the history, i.e., it replaces some updates

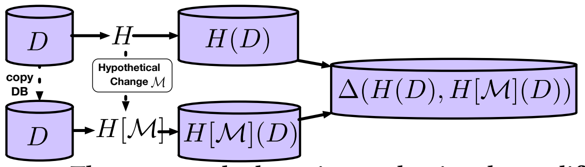

D
H
$H(D)$
copy DB
Hypothetical Change $\mathcal{M}$
$\Delta(H(D), H[\mathcal{M}](D))$
D
$H[\mathcal{M}]$
$H[\mathcal{M}](D)$

Figure 5: The naïve method requires evaluating the modified history over a copy of the original database.

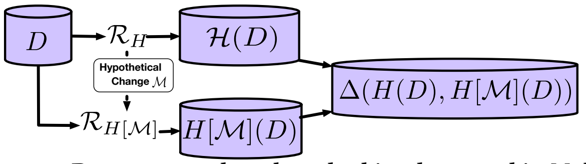

D
$\mathcal{R}_H$
$\mathcal{H}(D)$
Hypothetical Change $\mathcal{M}$
$\Delta(H(D), H[\mathcal{M}](D))$
$\mathcal{R}_{H[\mathcal{M}]}$
$H[\mathcal{M}](D)$

Figure 6: Reenactment-based method implemented in Mahif

from H with hypothetical updates (or inserts new / deletes existing update statements). We use $H[\mathcal{M}]$ to denote the history that is the result of applying $\mathcal{M}$ to H. The result of $\mathcal{H}$ is the symmetric difference (Δ) of the database instances produced by evaluating H ($H[\mathcal{M}]$) over database D, i.e., the set of tuples in the result of the history that are affected by the modification. For our running example, the symmetric difference would contain the two versions of the tuple with ID 12 produced by the original and modified history. We focus on deterministic updates (given the same input, multiple executions of an update are guaranteed to return the same result). The existence of an update in a transactional history is often dependent on the existence of other updates in the history and/or on external events (e.g., user interactions) which are not observed by the DBMS. For instance, if we delete a statement that inserted a customer, then this customer could have never submitted any orders. Consequently, all insert statements corresponding to orders by this customer should be removed. While dealing with such causal relationships is important for helping users to formulate realistic hypothetical scenarios, it is orthogonal to the problem we study in this work: how to efficiently answer HWQs. Learning such causal relationships between the updates of a history and then using them to augment a user-provided HWQ is an interesting and challenging problem that we leave to future work.

A **naïve approach** for answering a HWQ is shown in Fig. 5. This method creates a copy of the database as it was before the execution of the first update that has been modified by $\mathcal{M}$, and then executes the modified history on this copy. It then computes the symmetric difference between the current database state (which is the result of evaluating the original transactional history H over D) and the database state that is the result of evaluating the modified history $H[\mathcal{M}]$ over the copy of database D. Note that this requires access to a past database state D before the execution of the first update of the history, e.g., we can use a DBMS with support for *time travel* to access D (e.g., Oracle, SQLsever, DB2). The naïve method requires additional storage to store the copy of D and the evaluation of the modified history results in a large amount of write I/O. However, an even larger concern is that the modifications $\mathcal{M}$ may only affect a small fraction of the data and many updates in the history may be irrelevant for computing the symmetric difference.

Our **proposed method** is shown in Fig. 6. In order to overcome the limitations of the naïve method, we propose Mahif as a system that answers HWQs using reenactment [3–5], a declarative technique for replaying transactional histories using queries. Our approach also uses time travel to access D, the state of the database just before the time the first modified update was executed. In contrast to the naïve method, the database does not need to be copied. Instead, the modified history is reenacted over D by running a query $\mathcal{R}_{H[\mathcal{M}]}$. Thus, reenactment has the advantage of not incurring write I/O. The result of query $\mathcal{R}_{H[\mathcal{M}]}$ is equal to the result of executing $H[\mathcal{M}]$ over D. We then compute the symmetric difference between the result of the modified history (returned by $\mathcal{R}_{H[\mathcal{M}]}$) and the current database state ($H(D)$) computed by reenacting H over D. Reenacting H, while seemingly redundant, allows us to develop novel optimizations which exclude irrelevant updates from the history and irrelevant data from reenactment.

Program Slicing. To be able to identify updates that can safely be excluded from the evaluation of an HWQ, we introduce the notion of a *slice*. A slice for a HWQ $\mathcal{H}$ is a subset of the updates from H and $H[\mathcal{M}]$ that is sufficient for computing the result of $\mathcal{H}$. We identify a property called tuple-independence which holds for a large class of updates (corresponding to SQL update and delete statements without joins and subqueries, and `INSERT ... VALUES ...` statements). Tuple independence ensures that we can determine whether a subset of updates is a slice by testing for each individual tuple from the database whether the subset produces the same result for $\mathcal{H}$ than for the full histories. To improve the efficiency of slicing, we compress D into a set of constraints that compactly over-approximate the database. Inspired by program slicing and symbolic execution techniques [9, 27], and ideas from incomplete databases [1, 24], we develop a technique that evaluates updates from a history over a single tuple symbolic instance (a tuple with variables as attribute values) subject to the constraints from the compressed database. The result of symbolic evaluation is a single tuple symbolic instance that encodes all possible tuples in the result of the history for any input tuple fulfilling the compressed database constraints. We then use a constraint solver to determine whether a candidate slice produces the same result for $\mathcal{H}$ as the full histories for every possible input tuple. If that is the case, then it is safe to use the slice instead of H and $H[\mathcal{M}]$ to answer $\mathcal{H}$. The cost of program slicing only depends on the number of updates in the history and the size of the constraints encoding the data distribution of the database.

Data Slicing. We also propose *data slicing* to prune data that we can prove is irrelevant for computing the answer to a HWQ. Based on the observation that any tuple in the symmetric difference has to be affected by at least one statement that was modified by $\mathcal{M}$, we filter the input of reenactment to remove tuples which are guaranteed to not be affected by any update modified by $\mathcal{M}$. In addition to the class of queries supported by program slicing, data slicing is also applicable to insert statements with queries (`INSERT ... SELECT` in SQL). The main contributions of this paper are:

- We formalize historical what-if queries and present a novel method for answering such queries based on reenactment.
- We present two optimization techniques, *program slicing* and *data slicing*, which determine which updates and what data can be safely excluded when answering a HWQ.

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

$$\phi := \mathbf{e}\{=,\neq,<,\leq,>,\geq\}\mathbf{e}|\phi\{\wedge,\vee\}\phi|\mathbf{e}\ \textbf{isnull}|\neg\phi|\textbf{true}|\textbf{false}$$

Figure 7: Syntax of expressions e and conditions ϕ

- We demonstrate experimentally that our approach outperforms the naïve approach and that our optimizations result in significant additional performance improvements.

2 BACKGROUND AND NOTATION

Given a universal value domain $\mathbb{D}$, a relation R (instance) of arity n is a subset of $\mathbb{D}^n$. A database instance (or database for short) D is a set of relations R_1 to R_n. We use $\textsc{Sch}(R)$ to denote the schema of relation R. We consider three type of update operations: updates, inserts, and deletes. In the following, we will use the term *update statement*, or statement for short, as an umbrella term for updates, deletes, and inserts. We view statements as functions that take a relation R (or database in the case of inserts with a query) as input and return an updated version of R. We use u to denote any such statement and use $u(R)$ (and sometimes abusing notation also $u(D)$) to denote the result of applying statement u to relation R. An insert $\mathcal{I}_t(R)$ inserts tuple t with the same arity as R into relation R. An insert $\mathcal{I}_Q(R)$ inserts the result of the query Q evaluated over database D into R. A delete $\mathcal{D}_\theta(R)$ removes all tuples from R that do not fulfill condition θ. Finally, an update $\mathcal{U}_{Set,\theta}(R)$ updates the values of each tuple t that fulfills condition θ based on a list of expressions *Set* and returns all other input tuples unmodified. *Set* is a list of expressions $(e_1, ..., e_n)$ with the same arity as R. Each such expression is over the schema of R. We will sometimes use $(A_{i_1} \leftarrow e_1, ..., A_{i_m} \leftarrow e_m)$ as a notional shortcut assuming that the expression for each attribute that is not explicitly mentioned is the identity. For instance, $Set = (B \leftarrow B + 3)$ over schema (A, B, C) denotes $(A, B + 3, C)$. For an update or delete u we use θ_u to denote the update's (delete's) condition. Similarly, Set_u for an update u denotes the update's list of *Set* expressions.

A condition θ (as used in updates and deletions) is a Boolean expression over comparisons between scalar expressions containing variables and constants. The grammar defining the syntax of *Set* and θ expressions is shown in Fig. 7. For any expression e, e', and e'' we use $e[e' \leftarrow e'']$ to denote the result of substituting each occurrence of e' in e with e''. We write $Set(t)$ to denote the tuple produced by evaluating the expressions from *Set* over input tuple t (required to be of the same arty as *Set*). For example, for a relation $R(A, B, C)$, tuple $t = (1, 1, 1)$, and $Set = (A, A + B, 20)$ we get $Set(t) = (1, 2, 20)$. Sometimes, we will use $u(t)$ to denote the tuple that is the result of applying a statement u to a single tuple t. We formally define the semantics of evaluating statements over a database D below. Note that the update statements we define here correspond to SQL update and delete statements without nested subqueries and joins and to `INSERT INTO ... VALUES ...` and `INSERT INTO ... SELECT ...`.

$$\mathcal{U}_{Set,\theta}(R) = \{Set(t) \mid t \in R \wedge \theta(t)\} \cup \{t \mid t \in R \wedge \neg\theta(t)\} \quad (1)$$

$$\mathcal{D}_\theta(R) = \{t \mid t \in R \wedge \neg\theta(t)\} \quad (2)$$

$$\mathcal{I}_t(R) = R \cup \{t\} \quad (3)$$

$$\mathcal{I}_Q(R) = R \cup Q(D) \quad (4)$$

A **history** $H = u_1, \ldots, u_n$ over a database D is a sequence of updates over D. Given a history $H = u_1, \ldots, u_n$, we use $H_{i,j}$ for $i \leq j \in [1, n]$ to denote $u_i, u_{i+1}, \ldots, u_j$. Similarly, H_i, called a prefix of H, denotes $H_{1,i}$. Furthermore, for a set of indices $\mathcal{I} = \{i_1, \ldots, i_m\}$ such that $i_j < i_k$ if $j < k$ and $i_j, i_k \in [1, n]$, we use $H_\mathcal{I}$ to denote $(u_{i_1}, \ldots, u_{i_m})$. We use $H(D)$ to denote the result of evaluating the history H over a database instance D (recursively defined below using the fact that $H_n = H$) and will use D_i to denote $H_i(D)$.

$$D_1 = u_1(D) \qquad D_i = u_i(D_{i-1}) \qquad (\text{for } 1 < i \leq n)$$

Our program slicing technique relies on a property we call *tuple independence.* Intuitively, statements that fulfill this property process each input tuple individually.

Definition 1 (Tuple independence). *A statement u is* tuple independent *if for every database D, we have $u(D) = \bigcup_{t \in D} u(\{t\})$*

In SQL, all updates and deletes without nested subqueries or joins and inserts without queries are tuple independent. Thus, all of our statements with the exception of $\mathcal{I}_Q$ are tuple independent.

Lemma 1 (Tuple independent statements). *All updates $\mathcal{U}_{Set,\theta}$, deletes $\mathcal{D}_\theta$, and inserts $\mathcal{I}_t$ are tuple independent.*

Proof Sketch. Proven by unfolding of definitions and using the fact that comprehension distributes over union if the conditions in the comprehension are only over the element that is returned. That is, for any set S and condition ψ that only depends on e, the following equivalence holds: $\{e \mid e \in S \wedge \psi\} = \bigcup_{e \in S}\{e \mid \psi\}$. For the full proof please see [12] □

3 HISTORICAL WHAT-IF QUERIES

We now formally define historical what-if queries. Let H be a history containing an update u. Historical what-if queries are based on **modifications** $m = u \leftarrow u'$ that replace the statement u in H with another statement u', delete the statement u at position i ($m = \mathbf{del}(i)$), or insert a new statement u at position i ($m = \mathbf{ins}_i(u)$). We use $\mathcal{M}$ to denote a sequence of modifications and $H[\mathcal{M}]$ to denote the result of applying the modifications $\mathcal{M}$ to the history H. For example, for a history $H = u_1, u_2, u_3$ and $\mathcal{M} = (u_1 \leftarrow u'_1, \mathbf{del}(3))$ we get $H[\mathcal{M}] = u'_1, u_2$. Replacing a statement u with a statement u' of a different type, e.g., replacing an update with a delete, can be achieved by deleting u and then inserting u'.

To answer a historical what-if query, we need to compute the difference between the current state of the database, i.e., $H(D)$ and the database produced by evaluating the modified history, i.e., $H[\mathcal{M}](D)$. For that we introduce the notion of a database delta. A *database delta* $\Delta(D, D')$ contains all tuples that only occur in D or in D'. Tuples that exclusively are in D' are annotated with a + and tuples that exclusively appear in D are annotated with −.

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

We define a historical what-if query and an answer to such query based on the delta of $H(D)$ and $H[\mathcal{M}](D)$.

Definition 2 (Historical What-If Queries). *A* ***historical what-if query*** *$\mathcal{H}$ is a tuple $(H, D, \mathcal{M})$ where H is a history executed*

Algorithm 1 Naïve HWQ Algorithm

```
1: procedure NAIVE-WHATIF($H$, $D$, $D_{cur}$, $\mathcal{M}$)
2:     $D' \leftarrow$ COPY($D$)
3:     $D_{mod} \leftarrow H[\mathcal{M}](D')$
4:     return $\Delta(D_{cur}, D_{mod})$
```

over database instance D, and $\mathcal{M}$ denotes a sequence of modifications to H as introduced above. The answer to $\mathcal{H}$ is defined as:

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

EXAMPLE 2. *Let D and H be the database shown in Fig. 1 and history shown in Fig. 3, respectively. Consider the modification $\mathcal{M}_1 = (u_1 \leftarrow u_1{}')$ where u_1 and $u_1{}'$ are the updates shown in Fig. 2. $\mathcal{M}_1$ increases the minimum price for waving shipping fees. Bob's historical what-if query from this example can be written as $\mathcal{H}_{Bob} = (H, D, \mathcal{M}_1)$ in our framework. Evaluating $H[\mathcal{M}_1]$ results in the modified database instance shown in Fig. 4. For convenience, we have highlighted modified tuple values. The answer of the HWQ $\mathcal{H}_{Bob}$ is*

$$\Delta(H(D), H[\mathcal{M}_1](D)) = \{-o_6, +o_6'\}$$

That is, the shipping fee for Alex's order is increased by \$5 because it is no longer eligible for free shipping under the new policy ($u_1{}'$).

4 NAÏVE ALGORITHM

Before giving an overview of our approach, we briefly revisit the naïve algorithm (Algorithm 1) in more detail. WLOG assume that $\mathcal{M}$ modifies the first update in the history (and possibly others). If this is not the case, then we can simply ignore the prefix of the history before the first modified statement and use the state of the database before that statement instead of the database before first statement in the history. The input to the algorithm is the history H, the database state before the first statement of H was executed (D), the current state of the database D_{cur} which is assumed to be equal to $H(D)$, and the modifications $\mathcal{M}$ of the historical what-if query $\mathcal{H}$. We assume that D can be accessed using time travel. The algorithm first creates a copy of D' of D. Note that we only need to copy relations that are accessed by the history. The state of any relation not accessed by H will be the same in $H(D)$ and $H[\mathcal{M}](D')$. We rename the relations in D' to avoid name clashes. We then execute $H[\mathcal{M}]$ over the copy D' resulting in $D_{mod} = H[\mathcal{M}](D')$ (Line 3). In the last step (Line 3), the delta of D_{cur} and D_{mod} is computed. The delta computation is implemented as a single query for each relation of D accessed by H. For instance, a relational algebra query computing the delta for a relation R with schema $\text{SCH}(R) = (A, B)$ is shown below. Note that + and − are constants, i.e., the projections add an additional column storing the annotation of a tuple.

$$\Pi_{A,B,-}(R_{cur} - R_{mod}) \cup \Pi_{A,B,+}(R_{mod} - R_{cur})$$

5 OVERVIEW OF OUR APPROACH

We now give a high-level overview of our Algorithm 2 for answering a HWQ $\mathcal{H} = (H, D, \mathcal{M})$. To answer a historical what-if query, we need to compute $H(D)$ and $H[\mathcal{M}](D)$, and compute the delta of $H(D)$ and $H[\mathcal{M}](D)$. As mentioned earlier, we utilize a technique called reenactment for this purpose. In the following we first give an overview of reenactment and then discuss how it is applied by our approach.

Algorithm 2 Optimized, Reenactment-based HWQ Algorithm

```
1: procedure WHATIF($H$, $D$, $\mathcal{M}$)
2:     $\mathcal{I} \leftarrow$ PROGRAMSLICING($H, H[\mathcal{M}]$)      ▷ Compute Slice $\mathcal{I}$
3:     $\mathcal{R}_{H_\mathcal{I}} \leftarrow$ GENREENACTMENTQUERY($H_\mathcal{I}$)
4:     $\mathcal{R}^{DS}_{H_\mathcal{I}} \leftarrow$ DATASLICING($H, \mathcal{M}, \mathcal{R}_{H_\mathcal{I}}$)
5:     $\mathcal{R}_{H[\mathcal{M}]_\mathcal{I}} \leftarrow$ GENREENACTMENTQUERY($H[\mathcal{M}]_\mathcal{I}$)
6:     $\mathcal{R}^{DS}_{H[\mathcal{M}]_\mathcal{I}} \leftarrow$ DATASLICING($H, \mathcal{M}, H[\mathcal{M}]_\mathcal{I}$)
7:     return $\Delta(\mathcal{R}^{DS}_{H_\mathcal{I}}, \mathcal{R}^{DS}_{H[\mathcal{M}]_\mathcal{I}})$
```

5.1 Reenactment

Reenactment [4, 5] is a technique for simulating a transactional history through queries. For simplicity we limit the discussion to a history H over a single relation R even though our approach supports histories over multiple relations. Using reenactment, we can construct a query $\mathcal{R}_H$ such that $H(R) = \mathcal{R}_H(R)$. Reenactment was originally developed for capturing provenance for transactional workloads under multiversioning concurrency control protocols. For our purpose, we only need reenactment for set semantics and introduce a simplified translation for this case. We use $\mathcal{R}_u$ ($\mathcal{R}_H$) to denote the reenactment query for a single statement u (history H).

DEFINITION 3 (REENACTMENT QUERIES). *Let be a statement u (update $\mathcal{U}_{Set,\theta}$, delete $\mathcal{D}_\theta$, insert $\mathcal{I}_t$, or insert $\mathcal{I}_Q$) over a relation R with schema $(A_1, \ldots, A_n)$ and let $Set = (e_1, \ldots, e_n)$. The reenactment query $\mathcal{R}_u$ for u is defined as shown below:*

$$\mathcal{R}_{\mathcal{U}_{Set,\theta}} := \Pi_{\textbf{if } \theta \textbf{ then } e_1 \textbf{ else } A_1, \ldots, \textbf{if } \theta \textbf{ then } e_n \textbf{ else } A_n}(R)$$

$$\mathcal{R}_{\mathcal{D}_\theta} := \sigma_{\neg\,\theta}(R) \qquad \mathcal{R}_{\mathcal{I}_t} := R \cup \{t\} \qquad \mathcal{R}_{\mathcal{I}_Q} := R \cup Q$$

Let $H = (u_1, \ldots, u_n)$ be a history. The reenactment query $\mathcal{R}_H$ for H is constructed from the reenactment queries for u_i for $i \in \{1, \ldots, n\}$ by substituting the reference to relation R in $\mathcal{R}_{u_i}$ with $\mathcal{R}_{u_{i-1}}$.

An insert is reenacted as the union between the current state of relation R and the inserted tuple ($\mathcal{I}_t$) or the result of query Q (for $\mathcal{I}_Q$). For a delete $\mathcal{D}_\theta$, we have to remove all tuples fulfilling the condition of the delete. This is achieved by using a selection to only retain tuples that do not fulfill this condition, i.e., we filter based on $\neg\theta$. To reenact an update, we have to update the attribute values of all tuples fulfilling the condition θ using the expressions *Set*. All other tuples are just copied from the input. For that, we project on conditional expressions that for each attribute A_i return e_i if the tuple fulfills θ and A_i otherwise. For a history H which accesses multiple relations, a separate query, $\mathcal{R}^R_H$, is constructed for each relation R based on all statements from history H that access R.

EXAMPLE 3. *Consider Ex. 1 and let $I, U, C, P,$ and F denote attributes* ID, Customer, Country, Price, *and* ShippingFee *of relation* Order *(abbreviated as* O*). The reenactment query $\mathcal{R}^O_H$ for the history H from Fig. 2 is:*

$$\mathcal{R}^O_H = \Pi_{I,U,C,P,\textbf{if } P\leq 30 \wedge F\geq 10 \textbf{ then } F-2 \textbf{ else } F}(
\Pi_{I,U,C,P,\textbf{if } U=UK \wedge P\leq 100 \textbf{ then } F+5 \textbf{ else } F}(
\Pi_{I,U,C,P,\textbf{if } P\geq 50 \textbf{ then } 0 \textbf{ else } F}(O)))$$

Recall that $H[\mathcal{M}]$ differs from H in that $u_1{}'$ replaces u_1 and that the condition of $u_1{}'$ is $P \geq 60$. Thus, $\mathcal{R}^O_{H[\mathcal{M}]}$ differs from $\mathcal{R}^O_H$ in that condition $P \geq 50$ in the first selection is replaced with $P \geq 60$.

5.2 Reenacting Historical What-if Queries

As shown above, we use reenactment to simulate the evaluation of histories. Given the reenactment queries for H and $H[\mathcal{M}]$, what remains to be done is to compute their delta. Continuing with our example from above, the result $\Delta(\mathcal{R}^O_H(D), \mathcal{R}^O_{H[\mathcal{M}]}(D))$ of $\mathcal{H}$ is computed as shown below.

$$\Delta(\mathcal{R}^O_H(D), \mathcal{R}^O_{H[\mathcal{M}]}(D)) = \Pi_{I,U,C,P,F,-}(\mathcal{R}^O_H(D) - \mathcal{R}^O_{H[\mathcal{M}]}(D)) \\ \cup \Pi_{I,U,C,P,F,+}(\mathcal{R}^O_{H[\mathcal{M}]}(D) - \mathcal{R}^O_H(D))$$

We use Algorithm 2 to answer historical what-if queries. This algorithm applies two novel optimizations that significantly improve performance. Program slicing (Line 2, discussed in Sec. 7) determines subsets of histories (encoded as a set of positions $\mathcal{I}$ called a *slice*) which are sufficient for computing the answer to the what-if query $\mathcal{H}$. We then generate reenactment queries (Lines 3 and 5) for the slices of H and $H[\mathcal{M}]$ according to $\mathcal{I}$. Recall that $H_\mathcal{I}$ denotes the history generated from H by removing all statements not in $\mathcal{I}$. Afterwards (Lines 4 and 6), we apply our second optimization, data slicing (discussed in Sec. 6). Data slicing injects selection conditions into the reenactment query that filter out data that is irrelevant for computing the result of the HWQ. The result of data and program slicing is an optimized version of a reenactment query that has to process significantly less data and avoids reenacting updates that are irrelevant for $\mathcal{H}$. We then calculate the delta of these two queries and return it as the answer for $\mathcal{H}$ (Line 7).

6 DATA SLICING

In this section, we present *data slicing*, a technique which excludes data from reenactment for a HWQ $\mathcal{H}$ without affecting the result. Our technique is based on the observation that any difference between $H(D)$ and $H[\mathcal{M}](D)$ has to be caused by a difference between H and $H[\mathcal{M}]$. Thus, any tuple that is in the result of $\mathcal{H}$ has to be derived from a tuple that was affected (e.g., fulfills the condition of an update) by a statement affected by $\mathcal{M}$ in either the original history, the modified history, or both (but in different ways).

For example, in our running example from Fig. 2 the original update u_1 and modified update $u_1{}'$ only modify tuples for which either $Price \geq 50$ or $Price \geq 60$. For instance, the tuple with ID 11 does not fulfill any of these two conditions. Even though this tuple is modified by both histories, the same modifications are applied and, thus, the final result is the same (see Fig. 3 and Fig. 4): the shipping fee of this order was changed to \$8. Our data slicing technique determines selection conditions that filter out such tuples. For instance, for our running example we can apply the condition shown below (checking that either u_1 or $u_1{}'$ may modify the tuple):

$$(Price \geq 50) \vee (Price \geq 60)$$

Initially, we will limit the discussion to data slicing for a single modification $m = u \leftarrow u'$ where u and u' are of the same type (e.g., both are updates). We will show how to construct conditions $\theta^{DS}_H(m)$ and $\theta^{DS}_{H[\mathcal{M}]}(m)$ that we apply to filter irrelevant tuples from the inputs of $\mathcal{R}_H$ and $\mathcal{R}_{H[\mathcal{M}]}$. As explained above, for a single modification $u \leftarrow u'$ we can assume WLOG that u is the first update in H, because any update before u can be ignored for reenactment. Afterwards, we extend the technique for multiple modifications and modifications that insert or delete statements (which also covers modifications that replace a statement with a statement of a different type). In the following, we will use Q^{DS}_H to denote $\sigma_{\theta^{DS}_H(m)}(R)$ and $Q^{DS}_{H[\mathcal{M}]}$ to denote $\sigma_{\theta^{DS}_{H[\mathcal{M}]}(m)}(R)$.

Updates. First, consider a modification $m = u \leftarrow u'$ where both u and u' are updates. Since only tuples that match the condition of an update operation (the operation's **WHERE** clause) can be affected by the operation, 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 tuples in D from which such a tuple is derived have to either match the condition of u (θ_u) or the condition of u' ($\theta_{u'}$). This means we can filter the input to the reenactment queries using:

$$\theta^{DS}_H(m) = \theta^{DS}_{H[\mathcal{M}]}(m) = \theta_u \vee \theta_{u'} \tag{5}$$

Deletes. Let us now consider a single modification $u \leftarrow u'$ which replaces a delete $u = \mathcal{D}_\theta$ with a delete $u' = \mathcal{D}_{\theta'}$. For a tuple $t \in R$ to contribute to $\Delta(\mathcal{R}_H(R), \mathcal{R}_{H[\mathcal{M}]}(R))$, it has to be deleted by either u or u', but not by both (such tuples do not contribute to any result of $\mathcal{R}_H(R)$ or $\mathcal{R}_{H[\mathcal{M}]}(R)$). Thus, we can filter from R all tuples that do not fulfill the condition

$$\theta^{DS}_H(m) = \theta^{DS}_{H[\mathcal{M}]}(m) = (\theta \wedge \neg\, \theta') \vee (\neg\, \theta \wedge \theta') \tag{6}$$

Inserts with Queries. Recall that an insert $\mathcal{I}_Q$ is reenacted using the query $R \cup Q$. Only tuples that are returned by the query Q need to be considered. Thus, if $\mathcal{I}_Q$ is the only statement that is modified, then it is sufficient to replace $R \cup Q$ in the reenactment query with Q. However, for multiple modifications, tuples from the LHS of the union of the reenactment query for a statement $\mathcal{I}_Q$ may be affected by downstream updates modified by $\mathcal{M}$. Thus, we cannot simply replace $R \cup Q$ with Q if $\mathcal{I}_Q$ is not the first and only statement in the history that got modified by $\mathcal{M}$. To deal with this case, we need a condition that selects tuples which may contribute to the result of Q. We can achieve this by pushing the selection conditions of Q down to the relations accessed by Q. For that we apply standard selection move-around techniques from query optimization. The final result is a selection condition for each input relation of the query. For instance, for $\mathcal{I}_{\sigma_{A=5}(R \bowtie_{A=C} S)}(R)$ over relations $R(A, B)$ and $S(C, D)$, the selection can be pushed to both inputs of the join resulting in a condition $A = 5$ for R and $C = 5$ for S. We formally define the rules for pushing conditions through queries in [12].

Multiple modifications. Data slicing can also be applied to HWQs with more than one modification. For a tuple to be in the result of the what-if query, it has to be affected by at least one statement u such that there exists one modification $m \in \mathcal{M}$ with either $m = u \leftarrow u'$ or $m = u' \leftarrow u$ for some statement u'. However, we cannot simply use the disjunction of the data slicing conditions $\theta^{DS}_H(m)$ and $\theta^{DS}_{H[\mathcal{M}]}(m)$ we have developed for single modifications to filter the input. To see why this is the case, consider a modification $m = u \leftarrow u'$ where u is the i^{th} update in H. The input of u (u') over which the condition of the update is evaluated is the result of H_{i-1} (or $H[\mathcal{M}]_{i-1}$). To be able to derive a selection condition that can be applied to R, we have to "push" the condition for u down to determine a condition that returns the set of tuples from R that contribute to tuples in H_{i-1} fulfilling condition $\theta_H^{DS}(m)$ (or $\theta_{H[\mathcal{M}]}^{DS}(m)$). For that, we iteratively substitute references to attributes in $\theta_H^{DS}(m)$ (or $\theta_{H[\mathcal{M}]}^{DS}(m)$) with the expressions from the previous statement in H that defines them. For instance, consider a history $H = (u_1 = \mathcal{U}_{A \leftarrow 3, C=5}, u_2 = \mathcal{U}_{B \leftarrow B+1, A<4})$ and modification $m = u_2 \leftarrow u_2'$ with $u_2' = \mathcal{U}_{B \leftarrow B+1, A<5}$. To push the condition $A < 4$ of u_2, we substitute A with **if** $C = 5$ **then** 3 **else** A and get (**if** $C = 5$ **then** 3 **else** A) < 4.

More formally, consider a modification $m = u_i \leftarrow u_i{}'$ for a history $H = (u_1, \ldots, u_n)$. Let us first consider how to push $\theta_H^{DS}(m)$ (the case for $\theta_{H[\mathcal{M}]}^{DS}(m)$ is symmetric). We construct $\theta_H^{DS}(m) \downarrow^j$, the version of $\theta_H^{DS}(m)$ pushed down through $j < i$ updates as shown below. We use $\theta_H^{DS}(m) \downarrow^*$ to denote $\theta_H^{DS}(m) \downarrow^{i-1}$, i.e., pushing the condition through all updates of the history before u. Furthermore, we use an operator $(\theta) \downarrow^Q$ to push a condition θ through a query Q. See [12] for the formal definition of this operator.

$$\theta_H^{DS}(m) \downarrow^0 = \theta_H^{DS}(m)$$

$$\theta_H^{DS}(m) \downarrow^{j+1} = \begin{cases} \theta_H^{DS}(m) \downarrow^j [\vec{A} \leftarrow \vec{e}] & \textbf{if } u_{i-j} = \mathcal{U}_{Set,\theta} \\ \theta_H^{DS}(m) \downarrow^j \vee (\theta_H^{DS}(m) \downarrow^j) \downarrow^Q & \textbf{if } u_{i-j} = \mathcal{I}_Q \\ \theta_H^{DS}(m) \downarrow^j & \textbf{otherwise} \end{cases}$$

In the above equation, $\vec{A}$ denotes $(A_1, \ldots, A_n)$ and $\vec{e}$ denotes

$$(\textbf{if } \theta \textbf{ then } Set(A_1) \textbf{ else } A_1, \ldots, \textbf{if } \theta \textbf{ then } Set(A_n) \textbf{ else } A_n)$$

Furthermore, $e[\vec{A} \leftarrow \vec{e}]$ denotes the result of substituting each reference to A_i in e with e_i (for all $i \in [1, n]$).

Example 4. *Consider our running example history and a modification that replaces u_3 (reducing shipping fee by \$2 if the shipping fee is at least \$10 and the order price is at most \$30) with $u_3{}'$ which applies to orders of $\leq$ \$40: $u_3{}' = \mathcal{U}_{F \leftarrow F-2, P \leq 40 \wedge F \geq 10}$. The data slicing condition for u_3 and $u_3{}'$ is $(P \leq 30 \wedge F \geq 10) \vee (P \leq 40 \wedge F \geq 10)$ which can be simplified to $(P \leq 40 \wedge F \geq 10)$. To push this condition through u_2, we have to substitute F (the shipping fee) with the conditional update of the shipping fee corresponding to u_2 and get $(P \leq 40 \wedge F'' \geq 10)$ for $F'' =$* **if** *$C = UK \wedge P \leq 100$* **then** *$F + 5$* **else** *F. We then have to push this condition through u_1. For that we substitute F again, this time with $F' =$* **if** *$P \geq 50$* **then** *0* **else** *F. The final data slicing condition for both H and $H[\mathcal{M}]$ and our modification $m = u_3 \leftarrow u_3{}'$ is:*

$$\theta_H^{DS}(m) \downarrow^* = \theta_{H[\mathcal{M}]}^{DS}(m) \downarrow^* = (P \leq 40 \wedge F'' \geq 10)$$

$$F'' = \textbf{if } C = UK \wedge P \leq 100 \textbf{ then } F' + 5 \textbf{ else } F'$$

$$F' = \textbf{if } P \geq 50 \textbf{ then } 0 \textbf{ else } F$$

Evaluating this condition over the database from Fig. 1, only the tuple with ID 11 has a sufficiently low price $P \leq 40$ and fulfills the condition $F'' \geq 10$ ($F = F' = 5$ and $F'' = F' + 5 = 10$). Thus, using this slicing condition we can exclude tuples 12, 13, and 14 from reenactment.

Modifications that insert or delete statements. Recall that we also allow modifications that insert a new statement at position i ($\mathbf{ins}_i(u)$) or delete the statement at position i ($\mathbf{del}(i)$). Note that it is possible to insert new statements into a history without changing its semantics as long as these statements do not modify any data, e.g., a delete $\mathcal{D}_{\mathbf{false}}$ that does not delete any tuples. We refer to such operations as *no-ops*. Using no-ops, we can pad the original history at position i for every insert $\mathbf{ins}_i(u)$. We then can rewrite $\mathbf{ins}_i(u)$ in $\mathcal{M}$ into a modification $u_i \leftarrow u$ where u_i is a no-op. A deletion $\mathbf{del}(i)$ is rewritten into a modification $u_i \leftarrow u_i{}'$ where $u_i{}'$ is a no-op. Thus, the data slicing method explained above is already sufficient for dealing with inserts $\mathbf{ins}_i(u)$ and deletes $\mathbf{del}(i)$.

Theorem 2 (Data Slicing). *Consider a history H, a relation R, and a sequence of modifications $\mathcal{M} = (m_1, \ldots, m_n)$. Let $Q_H^{DS} = \sigma_{\bigvee_{i=1}^n \theta_H^{DS}(m_i) \downarrow^*}(R)$ and $Q_{H[\mathcal{M}]}^{DS} = \sigma_{\bigvee_{i=1}^n \theta_{H[\mathcal{M}]}^{DS}(m_i) \downarrow^*}(R)$. Then,*

$$\Delta(\mathcal{R}_H(R), \mathcal{R}_{H[\mathcal{M}]}(R)) = \Delta(\mathcal{R}_H(Q_H^{DS}(R)), \mathcal{R}_{H[\mathcal{M}]}(Q_{H[\mathcal{M}]}^{DS}(R)))$$

Proof Sketch. We prove this theorem by induction over the number of modifications in $\mathcal{M}$. For the base case we prove the claim by case analysis (update and deletes). We show that any tuple not fulfilling θ and θ' does not contribute to the delta and, therefore can be excluded. For the inductive step, we prove by induction over the number of steps a conditions has to be "pushed-down", that the pushed-down condition ($\theta_H^{DS}(m_i) \downarrow^*$ or $\theta_{H[\mathcal{M}]}^{DS}(m_i) \downarrow^*$) excludes only irrelevant tuples. For the full proof see [12]. □

7 PROGRAM SLICING

In addition to data slicing, we also optimize the process of answering a historical what-if query $\mathcal{H} = (H, D, \mathcal{M})$ by excluding statements from reenactment if their existence has provably no effect on the answer of $\mathcal{H}$. This is akin to *program slicing* [13, 35] 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. 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. Recall that the result of a historical what-if query is computed as the delta (symmetric difference) between the result of the original and the modified history. That is, only tuples in the delta are relevant for determining slices.

Definition 4 (History Slices). *Let $\mathcal{H} = (H, D, \mathcal{M})$ be a historical what-if query over a history $H = (u_1, \ldots, u_n)$. Furthermore, let $\mathcal{I} = \{i_1, \ldots, i_m\}$ be a set of indexes from $[1, n]$ such that $i_j < i_k$ for $j < k$. We call $(H_\mathcal{I}, H[\mathcal{M}]_\mathcal{I})$ a* slice *for $\mathcal{H}$ if*

$$\Delta(H(D), H[\mathcal{M}](D)) = \Delta(H_\mathcal{I}(D_{i_1}), H[\mathcal{M}]_\mathcal{I}(D_{i_1}))$$

History slices allow us to optimize the evaluation of a historical what-if query by excluding statements from reenactment. Thus, ideally, we would like slices to be *minimal*, i.e., the result of removing any statement from $H_\mathcal{I}$ or $H[\mathcal{M}]_\mathcal{I}$ is not a slice. A naïve method for testing whether $\mathcal{I}$ is a slice, is to compute $\Delta(H(D), H[\mathcal{M}](D))$ and compare it against $\Delta(H_\mathcal{I}(D_{i_1}), H[\mathcal{M}]_\mathcal{I}(D_{i_1}))$. However, this is more expensive then just directly evaluating $\Delta(H(D), H[\mathcal{M}](D))$ which we wanted to optimize. Instead we give up minimality and restrict program slicing to tuple independent statements (Def. 1) which enables us to check that the slice and full histories produce the same result one tuple at a time. Furthermore, we design a method that (lossily) compresses the database D_C and checks this condition (same result for each input tuple) over the compressed database. Since the compression is lossy, a compressed database D_C represents all databases D such that compressing D yields D_C. To ensure that our method produces a slice that is valid for each such D, we adapt techniques from incomplete databases [24, 37].

8 SLICING WITH SYMBOLIC EXECUTION

We adapt concepts from incomplete databases [24] to reason about the behavior of updates over a set of possible databases represented by a compressed database. This is akin to *symbolic execution* [11, 26] which is used in software testing to determine inputs that would lead to a particular execution path in the program. We use *Virtual C-tables* [25, 37] (*VC-tables*) as a compact representation of the set of possible worlds represented by a compressed database (to be discussed in Sec. 8.3.1) and demonstrate how to evaluate updates with possible worlds semantics over such representations. That is, the result of a history over a VC-table instance encodes all possible results of the history over every possible world represented by the VC-table. Using a constraint solver, we can then prove existential or universal statements over these possible results. Specifically, we will check that a candidate slice and the full histories produce the same result for a HWQ $\mathcal{H}$.

8.1 Incomplete Databases 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 an incomplete database $\mathcal{D}$ are evaluated using possible world semantics where the result of the query (statement) is the set of possible worlds derived by applying the query (statement) to every possible world from $\mathcal{D}$:

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

For our purpose, it will be sufficient to use an incomplete database consisting of possible worlds containing a single tuple, because we restrict program slicing to tuple independent statements which process every input tuple independent of every other input tuple. This incomplete database contains one world for any such singleton relation. We then evaluate updates from the original and modified history and their slices over this incomplete database and search for worlds where the delta for the full histories is different from the delta for the slice. For efficiency we need a compact representation of an incomplete database. We employ Virtual C-tables [25, 37] which extend C-tables [24] to support scalar operations over values.

A VC-table $\mathbf{R}$ is a relation with tuples whose values are symbolic expressions over a countable set of variables Σ and where each tuple $\mathbf{t}$ (we use boldface to indicate tuples with symbolic values) is associated with a condition $\phi(\mathbf{t})$ (the so-called *local condition*). The grammar shown in Fig. 7 defines the syntax of valid expressions. A VC-database $\mathbf{D}$ is a set of VC-tables paired with a condition Φ, called a global condition. Let $\mathbb{D}$ denote a universal domain of values. A VC-db $\mathbf{D}$ encodes an incomplete database which consists of all possible worlds that can be generated by assigning a value to each variable in Σ, evaluating the symbolic expressions for each tuple in $\mathbf{D}$ and including tuples in the possible world whose local condition $\phi(\mathbf{t})$ evaluates to true. Only assignments for which the global condition Φ evaluates to true are part of the incomplete database represented by $\mathbf{D}$. We use $Mod(\mathbf{D})$ to denote the set of worlds encoded by the VC-database $\mathbf{D}$ (and apply the same notation for VC-tables). For ease of presentation, we will limit the discussion to databases with a single relation and for convenience associate a global condition with this single relation (instead of with a VC-database). However, our method is not subject to this restriction.

Definition 5 ($Mod(\mathbf{D})$). *Let* $\mathbf{D}$ *be a VC-db and let* Λ *be the set of all assignments* $\Sigma \to \mathbb{D}$.

$$Mod(\mathbf{D}) = \{D \mid \exists \lambda \in \Lambda : \lambda(\mathbf{D}) = D \wedge \lambda(\Phi)\}$$

Abusing notation, we apply λ *to VC-dbs, tuples, and symbolic expressions* e *using the semantics defined below.*

$$\lambda(\mathbf{D}) = \{\lambda(\mathbf{t}) \mid \mathbf{t} \in \mathbf{D} \wedge \lambda(\phi(\mathbf{t}))\}$$

$$\lambda(e_1 \diamond e_2) = \lambda(e_1) \diamond \lambda(e_2) \;\; \textit{for} \diamond \in \{+, -, \cdot, \div, =, \neq, <, \leq, >, \geq, \wedge, \vee\}$$

$$\lambda(\diamond e_1) = \diamond \lambda(e_1) \;\; \textit{for} \diamond \in \{\neg, \textbf{isnull}\}$$

$$\lambda((e_1, \cdots, e_n)) = (\lambda(e_1), \cdots, \lambda(e_n)) \qquad \lambda(\textbf{true}) = \textbf{true}$$

$$\lambda(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \begin{cases} \lambda(e_2) & \textbf{if } \lambda(e_1) \\ \lambda(e_3) & \textbf{otherwise} \end{cases} \qquad \lambda(\textbf{false}) = \textbf{false}$$

Example 5. *Consider relation* `Order` *from Ex. 1. In this example, we just consider the 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 Fig. 8. This VC-table contains a single tuple with three variables* $x_{Country}$, x_{Price}, *and* $x_{ShippingFee}$ *and 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* $\{(UK, 10, 0)\}$.

We will show in Sec. 8.3.1 how to encode information about the data distribution of the database of a HWQ as part of the global condition of an VC-database.

8.2 Updates on VC-Tables

Prior work on updating incomplete databases (e.g., [1, 21, 36]) does not support VC-tables. For our purpose, we need to be able to evaluate statements over VC-tables with possible world semantics. That is, the possible worlds of the result of applying a statement to a VC-table are derived by computing the statement over every possible world of the input. For an insert $\mathcal{I}_t$ we just add the concrete tuple t with a local condition $\phi(t) = \textbf{true}$ to the input VC-table $\mathbf{R}$. For a delete $\mathcal{D}_\theta$, for some assignment λ, the concrete tuple $\lambda(\mathbf{t})$ derived from a symbolic tuple $\mathbf{t} \in \mathbf{R}$ is deleted by the statement if the tuple's local condition evaluates to true $\lambda(\phi(\mathbf{t})) \models \textbf{true}$ and the tuple does not fulfill condition θ ($\lambda(\mathbf{t}) \not\models \theta$). We can achieve this behavior by setting the local condition of every tuple $\mathbf{t}$ to $\phi(\mathbf{t}) \wedge \neg\theta(\mathbf{t})$. The symbolic expression $\theta(\mathbf{t})$ is computed by substituting any reference to attribute A in θ with the symbolic value $\mathbf{t}.A$. An update $\mathcal{U}_{Set,\theta}$ can affect a tuple $\mathbf{t}$ in a VC-table in one of two ways in each possible world (λ): (i) either the update's condition evaluates to false and the values of $\lambda(\mathbf{t})$ are not modified or (ii) the update's condition evaluates to true on $\lambda(\mathbf{t})$ and *Set* is applied to the values of $\lambda(\mathbf{t})$. We have to provision for both cases.

One option is to encode each case as a separate output tuple, but this would result in an exponential blow-up of the number of tuples since each update in a history would double the number of tuples (because it produces two output tuples for each input tuple). We can avoid this exponential blow-up by introducing tuples with fresh variables to represent the updated versions of tuples and by assigning values to these new variables using the global condition. We show these semantics for statements below. To ensure that there are no name clashes between variables, we generate fresh variables $\{x_{\mathbf{t},A_i}\}$ to represent the value of attribute A_i of the tuple produced by applying the statement to tuple $\mathbf{t}$ from the input VC-table. We use $\phi(\mathbf{R},\mathbf{t})$ to denote the local condition of tuple $\mathbf{t}$ in relation $\mathbf{R}$ and for convenience define $\phi(\mathbf{R},\mathbf{t}) = \mathbf{false}$ for any $\mathbf{t} \notin \mathbf{R}$.

Definition 6 (Updates over VC-tables). *Let* $\mathbf{R}$ *be a VC-table and* $\textsc{Sch}(\mathbf{R}) = (A_1, \ldots, A_n)$. *Update statements over VC-tables are defined as shown below. Let* $Set = (e_1, \ldots, e_n)$. *Given a tuple* $\mathbf{t}$, *we use* $\mathbf{t_{new}}$ *to denote* $(x_{\mathbf{t},A_1}, \ldots, x_{\mathbf{t},A_n})$.

$$\mathcal{U}_{Set,\theta}(\mathbf{R}) = \{\mathbf{t_{new}} \mid \mathbf{t} \in \mathbf{R}\} \qquad \phi(\mathcal{U}_{Set,\theta}(\mathbf{R}), \mathbf{t_{new}}) = \phi(\mathbf{R}, \mathbf{t})$$

$$\Phi(\mathcal{U}_{Set,\theta}(\mathbf{R})) = \Phi(\mathbf{R}) \wedge \bigwedge_{\mathbf{t}\in\mathbf{R}} \bigwedge_{i=1}^{n} x_{\mathbf{t},A_i} = \mathbf{if}\ \theta(\mathbf{t})\ \mathbf{then}\ e_i(\mathbf{t})\ \mathbf{else}\ \mathbf{t}.A_i$$

$$\mathcal{D}_\theta(\mathbf{R}) = \{\mathbf{t} \mid \mathbf{t} \in \mathbf{R}\} \qquad \phi(\mathcal{D}_\theta(\mathbf{R}), \mathbf{t}) = \phi(\mathbf{R}, \mathbf{t}) \wedge \neg\theta(\mathbf{t})$$

$$\mathcal{I}_t(\mathbf{R}) = \mathbf{R} \cup \{t\} \quad \phi(\mathcal{I}_t(\mathbf{R}), t) = \mathbf{true} \quad \phi(\mathcal{I}_t(\mathbf{R}), \mathbf{t}) = \phi(\mathbf{R}, \mathbf{t}) \quad (\text{for } \mathbf{t} \neq t)$$

$$\Phi(\mathcal{I}_t(\mathbf{R})) = \Phi(\mathcal{D}_\theta(\mathbf{R})) = \Phi(\mathbf{R})$$

Using these semantics, the result of a sequence of n statements 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 of the statements (θ or *Set*). For our use case we execute a sequence of statements over an instance with a single tuple. Thus, it will be convenient to use a different naming schema for variables. We use $x_{A,i}$ to denote the value of attribute A of the version of this single input tuple after the i^{th} update.

Example 6. *Continuing with Ex. 5, consider the first two updates from Fig. 2 (we abbreviate attribute names as in previous examples):* $u_1 = \mathcal{U}_{F\leftarrow 0, P\geq 50}$ *and* $u_2 = \mathcal{U}_{F\leftarrow F+5, C=UK\wedge P\leq 100}$. *After execution of* u_1 *and* u_2 *over* $\mathbf{D_0}$ *shown in Fig. 8a, we get an instance with a single tuple. Since both updates only modify attribute* ShippingFee, *all other attributes can reuse the same variable as in the input. The value of attribute* ShippingFee *is a new variable* $x_{ShippingFee,2}$ *which is constrained by the global condition that ensures that it is equal to the previous value of this attribute* ($x_{ShippinFee,1}$) *if the condition of* u_2 *does not hold and otherwise is the result of applying* Set_{u_2} *to* $x_{ShippingFee,1}$. *Furthermore,* $x_{ShippingFee,1}$ *is related to the value of attribute* ShippingFee *in the input in the same way using a conditional expression based on* u_1*'s condition and update expression (setting the shipping fee to* 0 *if the price is at least* 50*).*

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

Theorem 3. *Let* $\mathbf{D}$ *be a VC-database and* u *a statement. We have:*

$$Mod(u(\mathbf{D})) = u(Mod(\mathbf{D}))$$

Proof Sketch. Using the definitions of $Mod(\mathbf{D})$ and $\lambda(\mathbf{D})$, we demonstrate equivalence over updates, inserts, and deletes. Note that λ factors through expressions (conditions θ and *Set* for updates) which we exploit to prove that any assignment λ that corresponds to a possible world of the input can be extended into an assignment λ' over the update's result such that: (i) λ' fulfills the global condition for the updated database and (ii) λ' assigns through the global condition to each variable appearing in tuples in this database the values from the tuple(s) produced by evaluating the update over the possible world $\lambda(\mathbf{D})$. The full proof is shown in [12]. □

Note that by induction, Thm. 3 implies that evaluating a history H over a VC-database also has possible world semantics.

8.3 Computing Slices with Symbolic Execution

To compute a slice for a historical what-if query $\mathcal{H} = (H, \mathcal{M}, D)$ where H consists of tuple independent statements only, we create a VC-database $\mathbf{D_0}$ with a single tuple with fresh variables for each relation in the database's schema. Even though they are tuple independent, we do not consider inserts of the form $\mathcal{I}_t$ here, because, as we will show in Sec. 9, we can split a reenactment query for a history with such inserts into a union of two queries — one that is the reenactment query for the history restricted to updates and deletes and a second one that only operates on tuples inserted by inserts $\mathcal{I}_t$ from the history. Since the second query only operates on an instance of size at most $|H|$, its cost is too low to warrant spending time on slicing it.

8.3.1 Compressing the Input Database. Optionally, we compress the input database D into a set of range constraints that restrict the variables of the single tuple in $\mathbf{D_0}$. For that, we decide on a number of groups and for each table select an attribute to group on. We then compute the minimum and maximum values of each non-group-by attribute A for each group and use them to constrain the values of A in this group. For each group we then generate a conjunction of these range constraints for each attribute. The disjunction of the constraints generated for the groups, which we denote as Φ_D, is then added to the global condition. Note that every tuple from a table of the database D corresponds to an assignment of the variables from $\mathbf{D_0}$ to the constants of the tuple that fulfills the condition. For attributes with unordered data types, we just omit the range condition for this attribute.

Example 7 (Compressing Databases). *Let us compress the instance from Fig. 1 into two tuples by grouping on* Country. *We get the following constraint that we add to the global condition to constrain the worlds of* $\mathbf{D_0}$. *Here, we omit the constraint for the name attribute and abbreviate attribute names as in previous examples.*

$$\begin{aligned}\Phi_D :=& (x_C = UK \wedge x_{ID} \in \{11, 12\} \wedge x_P \in [20, 50] \wedge x_F = 5)\\ &\vee (x_C = US \wedge x_{ID} \in \{13, 14\} \wedge x_P \in [30, 60] \wedge x_F \in [3, 4])\end{aligned}$$

For instance, the first two tuples (group UK*) get compressed into one conjunction of range constraints. Since the smallest (greatest) price in this group is* 20 *(*50*), the range constraint for* x_P *is* $x_P \in [20, 50]$.

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

Φ = true

(a) Initial VC-database $\mathbf{D_0}$.

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

$\Phi := \Phi_1 \wedge \Phi_2$

$\Phi_1 := (x_{ShippingFee,1} = \textbf{if } x_{Price} \geq 50 \textbf{ then } 0 \textbf{ else } x_{ShippingFee})$

$\Phi_2 := (x_{ShippingFee,2} = \textbf{if } (x_{Country} = UK \wedge x_{Price} \leq 100) \textbf{ then } x_{ShippingFee,1} + 5 \textbf{ else } x_{ShippingFee,1})$

(b) VC-table after evaluating $H = (u_1, u_2)$.

Figure 8: Running example for evaluating updates over VC-Tables.

Country	Price	ShippingFee	
x_C	x_P	$x_{F,2'}$	true

$\Phi' = \Phi_{1'} \wedge \Phi_{2'} \wedge \Phi_D$ $\quad \Phi_{1'} = (x_{F,1'} = \textbf{if } x_P \geq 60 \textbf{ then } 0 \textbf{ else } x_F)$

$\Phi_{2'} = (x_{F,2'} = \textbf{if } x_C = UK \wedge x_P \leq 100 \textbf{ then } x_{F,1'} + 5 \textbf{ else } x_{F,1'})$

(a) VC-database $H[\mathcal{M}](\mathbf{D_0})$

Country	Price	ShippingFee	
x_C	x_P	$x_{F,1''}$	true

$\Phi'' = \Phi_D \wedge x_{F,1''} = \textbf{if } x_P \geq 50 \textbf{ then } 0 \textbf{ else } x_F$

(b) VC-database $H_{\{1\}}(\mathbf{D_0})$

Country	Price	ShippingFee	
x_C	x_P	$x_{F,2'}$	true

$\Phi''' = \Phi_D \wedge x_{F,1'''} = \textbf{if } x_P \geq 60 \textbf{ then } 0 \textbf{ else } x_F$

(c) VC-database $H[\mathcal{M}]_{\{1\}}$

Figure 9: VC-database instances for our slicing example. Attributes names are abbreviated as: (C)ounty, (P)rice, Shipping(F)ee.

8.3.2 *Computing Slices.*

To determine whether a given set of indices $\mathcal{I}$ is a slice for $\mathcal{H}$, we have to test whether:

$$\Delta(H(D), H[\mathcal{M}](D)) = \Delta(H_{\mathcal{I}}(D), H[\mathcal{M}]_{\mathcal{I}}(D)) \tag{7}$$

Recall that we restrict program slicing to tuple independent statements (Def. 1). That is, the result produced by such a statement for an input tuple only depends on the values of this tuple and is independent of what other tuples exist in the input. Thus, if both deltas return the same result for every input tuple, then the two deltas are guaranteed to be equal. Thus, $\mathcal{I}$ is a slice if for all input tuples from D, both deltas return the same result (see Eq. (8) below). Note that this is only a sufficient, but not necessary condition. To see why this is the case, consider two input tuples t_1 and t_2 and assume that the delta of the results of the full histories returns s_1 for t_1 and s_2 for t_2, but the delta of the results of the sliced histories returns s_2 for t_1 and s_1 for t_2. The final result is the same, even though the results for the individual input tuples is different.

$$\begin{aligned}\forall t \in D : \; & \Delta(H(\{t\}), H[\mathcal{M}](\{t\})) \\ =& \Delta(H_{\mathcal{I}}(\{t\}), H[\mathcal{M}]_{\mathcal{I}}(\{t\}))\end{aligned} \tag{8}$$

For each $t \in D$, by construction of $\mathbf{D_0}$ (the VC-database we use as input for program slicing), there exists a world $D_t \in Mod(\mathbf{D_0})$ such that $D_t = \{t\}$. Note that since $\mathbf{D_0}$ is generated by compressing the input database into a set of range constraints, some worlds may not correspond to a tuple from D. However, our argument only requires that for each $t \in D$ there exists a world in $\mathbf{D_0}$ which implies that if the condition from Eq. (9) evaluates to true for every such D_t, then Eq. (8) holds. Thus, the formula shown below is a sufficient condition for $\mathcal{I}$ to be a slice.

$$\begin{aligned}\forall D_t \in Mod(\mathbf{D_0}) : \quad & \Delta(H(D_t), H[\mathcal{M}](D_t)) \\ =& \; \Delta(H_{\mathcal{I}}(D_t), H[\mathcal{M}]_{\mathcal{I}}(D_t))\end{aligned} \tag{9}$$

For an input tuple t, based on the definition of symmetric difference, $\Delta(H(D_t), H[\mathcal{M}](D_t))$ is equal to $\Delta(H_{\mathcal{I}}(D_t), H[\mathcal{M}]_{\mathcal{I}}(D_t))$ if either (i) $H(D_t) = H[\mathcal{M}](D_t)$ and $H_{\mathcal{I}}(D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t)$ which means that both deltas return the empty set for D_t or (ii) both deltas return the same set of tuples over D_t which is the case when $H(D_t) \neq H[\mathcal{M}](D_t)$ and one of the conditions shown below holds.

- (a) $H(D_t) = H_{\mathcal{I}}(D_t) \wedge H[\mathcal{M}](D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t)$
- (b) $H(D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t) \wedge H[\mathcal{M}](D_t) = H_{\mathcal{I}}(D_t)$

Thus, Eq. (9) is equivalent to:

$$\begin{aligned}&\forall D_t \in Mod(\mathbf{D_0}) : \\ &\big(H(D_t) = H[\mathcal{M}](D_t) \wedge H_{\mathcal{I}}(D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t)\big) \\ &\vee (H(D_t) \neq H[\mathcal{M}](D_t) \wedge \\ &\qquad (H(D_t) = H_{\mathcal{I}}(D_t) \wedge H[\mathcal{M}](D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t) \\ &\qquad \vee H(D_t) = H[\mathcal{M}]_{\mathcal{I}}(D_t) \wedge H[\mathcal{M}](D_t) = H_{\mathcal{I}}(D_t)))\end{aligned} \tag{10}$$

Based on the semantics of updates over VC-tables, the result of a history over a single tuple instance $\mathbf{D_0}$ is an instance with a single tuple whose local condition governs the existence of the tuple in any particular world D_t. For a history H let us denote this tuple as $\mathbf{t}_H$. Consider the valuation λ_t generating D_t. Then for two histories H and H', the condition $H(D_t) = H'(D_t)$ is equivalent to the equation shown below as long as we appropriately rename variables such that the two VC-databases do not share any variables except for the variables from $\mathbf{D_0}$.

$$\begin{aligned}&(\lambda_t(\mathbf{t}_H) = \lambda_t(\mathbf{t}_{H'}) \wedge \phi(\lambda_t(\mathbf{t}_H)) \wedge \phi(\lambda_t(\mathbf{t}_{H'}))) \\ &\vee(\neg\phi(\lambda_t(\mathbf{t}_H)) \wedge \neg\phi(\lambda_t(\mathbf{t}_{H'})))\end{aligned} \tag{11}$$

Intuitively, this condition means that for the two histories to return the same result over D_t, either (i) they both return the same result tuple (equal values and the local condition of the result tuple evaluates to true for both histories) or (ii) they both return the empty set (the local condition evaluates to false for both histories).

If we substitute this equation into Eq. (10), then we get a universally quantified first order sentence (a formula without free variables) over the variables from the VC-database $\mathbf{D_0}$. We will use $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ to denote the resulting formula (recall that Φ_D denotes the constraints encoding the compressed database). We can now use a constraint solver to determine whether $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ is true by checking that its negation is unsatisfiable. We use an MILP-solver for this purpose. The translation rules for transforming a logical condition into an MILP program are mostly well-known rules applied in linear programming and many have been used in related work (e.g., [29]). We refer the interested reader to [12] for the details. We are now ready to state the main result of this section.

Theorem 4 (Slicing Condition). *Let $\mathcal{H} = (H, D, \mathcal{M})$ be a historical what-if query where H is a history with n statements (updates and deletes). If $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ is true, then $\mathcal{I}$ is a slice for $\mathcal{H}$.*

Proof Sketch. We first prove that Eq. (8) implies Eq. (7) for histories consisting of updates and deletes which are both tuple independent. This follows from the definition of tuple independence (Def. 1). Eq. (8) is implied by Eq. (9), because the worlds of $\mathbf{D_0}$ encode a superset of D by construction and Thm. 3 (updates have possible world semantics). The equivalence of Eq. (10) and Eq. (9) follows from the definition of database deltas. Finally, the equivalence of $H(D_t) = H'(D_t)$ and Eq. (11) follows from Thm. 3. □

Example 8 (Testing Slice Candidates). *Consider our running example database (Fig. 1) and the history $H = \{u_1, u_2\}$ from Ex. 6 and let $u_1{}' = \mathcal{U}_{ShippingFee \leftarrow 0, Price \geq 60}$. Let $\mathbf{D_0}$ be as shown in Fig. 8, but with $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_D$ where Φ_D is the database constraint from Ex. 7. Furthermore, consider a HWQ $\mathcal{H} = (H, \mathcal{M})$ for $\mathcal{M} = (u_1 \leftarrow u_1')$ (higher price requirements for waiving shipping fees). To test whether $\mathcal{I} = \{1\}$ is a slice, we first have to construct $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ for which we have to evaluate H, $H[\mathcal{M}]$, $H_{\{1\}}$, and $H[\mathcal{M}]_{\{1\}}$ over $\mathbf{D_0}$. The results are shown in Fig. 9. We use Φ to denote the global condition of $H(\mathbf{D_0})$, Φ' for $H[\mathcal{M}](\mathbf{D_0})$, Φ'' for $H_{\{1\}}(\mathbf{D_0})$, and Φ''' for $H[\mathcal{M}]_{\{1\}}(\mathbf{D_0})$. Since the local condition of the result tuple is true in the result of both histories and their slices, we do not have to test whether the local condition is true or false as done in Eq. (11) and can instead directly test equality of two history's result tuples to test whether the histories return the same result. Furthermore, observe that all four histories only modify attribute* `ShippingFee`*. Thus, it is sufficient to compare tuples on attribute* `ShippingFee` *to determine whether the result tuples are the same. Applying these simplifications, $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ is equal to:*

$$\begin{aligned}\forall x_I, x_P, x_F :\ & \Phi_D \wedge \Phi \wedge \Phi' \wedge \Phi'' \wedge \Phi''' \wedge \\ & ((x_{F,2} = x_{F,2'} \wedge x_{F,1''} = x_{F,1'''}) \\ & \vee (x_{F,2} \neq x_{F,2'} \wedge ((x_{F,2} = x_{F,1''} \wedge x_{F,2'} = x_{F,1'''}) \\ & \qquad \vee (x_{F,2} = x_{F,1'''} \wedge x_{F,2'} = x_{F,1''}))))\end{aligned}$$

This formula is not true for all possible input tuples. For instance, if the shipping fee is $x_F = 55$ and country is $x_C = UK$, then the final shipping fee for H ($x_{F,2}$) is \$5 and for $H[\mathcal{M}]$ is \$50. Thus, $H(\lambda(\mathbf{D_0})) \neq H[\mathcal{M}](\lambda(\mathbf{D_0}))$. Since the slice candidate $\mathcal{I} = \{1\}$ does not apply the second update, we get \$0 (for $H_{\mathcal{I}}$) and \$45 (for $H[\mathcal{M}]_{\mathcal{I}}$). Thus, the slice candidate may produce a result for this database that is different to the one returned by $\mathcal{H}$ which means that $\mathcal{I}$ is not a valid slice.

8.3.3 *Our Slicing Algorithm.*

Given a set of indexes $\mathcal{I}$, we now have a sound method for testing whether $\mathcal{I}$ is a slice for a historical what-if query $\mathcal{H}$. A brute force approach for computing a slice would be to test all possible subsets of indexes to determine the smallest possible slice. Note that even this method is not guaranteed to return a minimal slice, because the test we have devised is not complete. The disadvantage of the brute force approach is that there is an exponential number of candidates (all subsets of the histories). We propose instead a greedy algorithm that considers a linear number of candidates. The algorithm starts with a trivial slice $\mathcal{I}_0 = [1, n]$ where n is the number of updates in the history (recall from Sec. 6 that we can pad histories such that both H and $H[\mathcal{M}]$ have n statements). It then iterates for n steps. In each iteration, we remove index i from the current slice $\mathcal{I}_i$ and test whether $\mathcal{I}_i - \{i\}$ is still a slice. If yes, then we set $\mathcal{I}_{i+1} = \mathcal{I}_i - \{i\}$. Otherwise, $\mathcal{I}_{i+1} = \mathcal{I}_i$. The final result produced by this algorithm is $\mathcal{I}_n$ which is guaranteed to be a valid slice.

In [12], we present an additional optimized version of the slicing condition $\zeta(\mathcal{H}, \mathcal{I}, \Phi_D)$ that only works for single modifications.

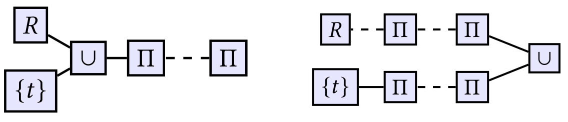
9 OPTIMIZING HISTORIES WITH INSERTS

In Sec. 7 and Sec. 8.2 we have limited the discussion to histories consisting only of update and delete statements. We now introduce an optimization that splits a reenactment query for a history into two parts that can be optimized individually: (i) one part that only simulates update and delete statements over the database at the time of the beginning of the history and (ii) a second part that evaluates the whole history, but only over tuples inserted by insert statements. We use program slicing to optimize (i). The input data size for (ii) is bound by the number of statements in the history and, thus, typically negligible. Note that our symbolic execution technique required by program slicing requires solving a MILP program (an NP-hard problem) whose size is polynomial in the size of the history. Thus, while it may be possible to extend program slicing techniques to deal with inserts, the costs of evaluating (ii) is polynomial in the size of the history and, thus, it is not possible to amortize the cost of program slicing for this part.

We first introduce the idea underlying our optimization, then formally define it, and finally prove its correctness. Recall from Def. 3 that the reenactment query for an insert statement $\mathcal{I}_t(R)$ is a union between the state of the relation before the insert and a singleton relation containing the inserted tuple. Updates are reenacted using projections and deletions using selection. As an example consider a history H consisting of a single insert u_0 followed by n update statements u_1 to u_n. Fig. 10a shows the structure of the reenactment query for this history. Using the standard algebraic equivalences shown below which allow us to pull a union through a projection or selection, we can pull the union up through the projections reenacting the updates of the history. The algebra tree for the resulting query is shown in Fig. 10b. Note that in the rewritten query (i) the right branch only accesses the tuple inserted by the insert statement and (ii) the left input to the union is equal to the reenactment query for a history H_{noIns} that is the result of deleting the insert statement from H.

$$\Pi_A(Q_1 \cup Q_2) \equiv \Pi_A(Q_1) \cup \Pi_{B \rightarrow A}(Q_2)$$
$$\sigma_\theta(Q_1 \cup Q_2) \equiv \sigma_\theta(Q_1) \cup \sigma_\theta(Q_2)$$

Generalizing this example, the algebraic equivalences shown above are sufficient for rewriting the reenactment query of any history H without insert statements with queries into a query $\mathcal{R}_{H_{noIns}} \cup \mathcal{R}_{H/R}$ where $\mathcal{R}_{H/R}$ is derived from $\mathcal{R}_H$ by replacing the subquery (union) corresponding to the first insert u in the history with the singleton relation $\{t\}$ containing the tuple inserted by u. Importantly, we can apply program slicing to optimize H_{noIns}.



(a) Before optimization (b) After optimization

Figure 10: Structure of an example reeactment query for a history with a single insert. The unions can be pulled up to create two separate queries: the left query accesses R and is the same as the reeactment query for the history without inserts while the right one only accesses inserted tuples.

10 RELATED WORK

What-if queries determine the effect of a hypothetical change to an input database on the results of a query. To avoid having to reevaluate the query over the full input including the hypothetical changes, incremental view maintenance is often applied to answer what-if queries [6, 7, 19, 39]. The how-to queries of Tiresias [29] determine how to translate a change to a query result into modifications of the input data. Similar to their approach, our system utilizes Mixed Integer Programming to express a set of possible worlds. The QFix system [34] is essentially a variation on this where the change to the output has to be achieved by a change to a query (update) workload. The query slicing technique of QFix is similar to our program slicing optimization. The main difference is that we apply symbolic execution to a single tuple instance, i.e., the number of constraints we produce is independent of the database size.

Several provenance models for relational queries have been studied such as Why-provenance, minimal Why-provenance [10], and Lineage [16]. The K-relations introduced by Green et al. [23] generalize these models for positive relational algebra. In [3–5] we have introduced MV-semirings [3–5] as an extension of K-relations that supports transactional updates. Furthermore, we did introduce reenactment as a technique for replaying histories using queries. The reenactment query for a history is equivalent to the history under MV-semiring semantics [3–5], i.e., it returns the same database state and provenance. [8] did study extensions of semiring-annotated relations for updates that allow updates to be deleted from a history. However, this approach only supports a limited class of updates.

The connection of provenance and program slicing was first observed in [13]. We present a method that statically analyzes potential provenance dependencies among statements in the history using a method which borrows ideas from symbolic execution [9, 26, 27], incomplete databases [1, 24, 25, 37], program slicing [35], and expressive provenance models [2]. Symbolic execution has been used extensively in software testing [11]. Cosette [14] is an automated prover for checking equivalences of SQL queries which converts input queries to constraints over symbolic relations. [38] use a symbolic representation of a query result to prove two queries to be equivalent using an SMT solver [18]. Rosette [33] is a language for building DSL with built-in support for symbolic evaluation. The transaction repair approach from [17] also detects dependencies between update operations like our optimized program slicing technique for single modifications. However, transaction repair operates on concrete data while we reason about the interactions of updates for a set of inputs encoded compactly as VC-tables.

11 EXPERIMENTS

We have conducted experiments to 1) evaluate the performance of our approach and compare it with the naïve approach, 2) examine the effectiveness of the proposed optimizations, and 3) study how our approach scales in database size and other important factors. All experiments were performed on a machine with 2 x AMD Opteron 4238 CPUs (12 cores total), 128 GB RAM, and 4 x 1TB 7.2K HDs in a hardware RAID 5 configuration. We used PostgreSQL 11.4 as the database backend. Based on preliminary experiments, the variance of runtimes was determined to be low. We repeated each experiment at least 3 times and report average runtimes.

11.1 Datasets and Workloads

Datasets. We use a taxi trips dataset from the Chicago's open data portal [1], as well as the standard TPC-C [2] and YCSB [15] benchmarks to evaluate the performance of our approach. The original taxi dataset has $\sim$ 100M rows and 23 attributes. The dataset contains trip information such as the `Company` (the taxi company), the `Taxi ID`, `Trip Start Timestamp` (when the trip started), `Trip Seconds` (duration of the trip in seconds), `Trip Miles` (distance of the trip in miles), `Pickup Community Area`, `Tips`, `Tolls`, `Extras` (tips, tolls and extra charges for the trip), and `Trip Total` (total cost of the trip). We used samples from these tables amounting to 10% ($5M$) and 50% ($50M$) of the entire taxi dataset in some of the experiments. The TPC-C and YCSB benchmark databases were generated with OLTP-Bench [20]. For TPC-C, we used the `stock` relation consisting of $10M$ rows (scale factor 100). For YCSB database we used scale factor 5000, resulting in a single relation consisting of $5M$ rows. The workloads generated by OLTP-Bench for each benchmark were modified to control the proportion of affected tuples.

Workloads. Unless stated otherwise, we use HWQs with a single modification that modifies the first update in a history over a single relation. We vary the following parameters. U is the number of updates in the history (e.g. $U100$ for 100 updates). M is the number of modifications made to the history. D is the percentage of updates that are dependent on the update(s) modified by the historical what-if query. We use 10% as the default ($D10$). T is the percentage of tuples in the relation that are affected by each dependent update (the default is 10%), where $T0$ means that a small, constant number of tuples was affected. I and X are the percentage of statements in the history that are inserts and deletes, respectively. Non-dependent statements affect a fixed fraction of the data equal to T, though independent from the tuples modified by dependent updates.

11.2 Compared Methods

We compare the following methods in our experiments. **Naïve (N)**: This method creates a copy of the database as of the start time of the history which is modified by the what-if query (`Creation`), executes $\mathcal{M}$ over this copy (`Exe`), and then computes the delta $\Delta(H(D), H[\mathcal{M}](D))$ by running a query over the current database state and the updated copy (`Delta`). **Reenactment Only (R)** creates a reenactment query for H and for $H[\mathcal{M}]$. We use run both reenactment queries over the database, and then compute the delta.

[1] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew (2020-10-13)
[2] TPC-C is an On-Line Transaction Processing Benchmark: http://www.tpc.org/tpcc/

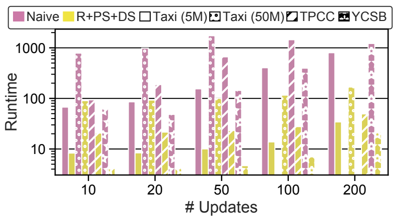


Figure 11: Naïve vs. Mahif (sec)

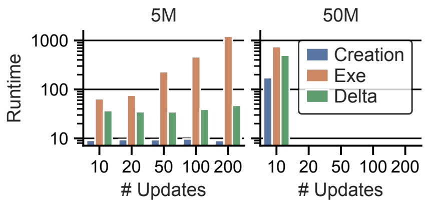


Figure 12: Breakdown Naïve

Size	Method	#Updates				
		10	**20**	**50**	**100**	**200**
5M	PS	0.07	0.18	1.30	8.46	62.13
	Exe	8.08	8.29	9.15	18.76	12.36
	R+PS+DS	8.14	8.47	10.45	27.23	74.49
	R	63.63	81.12	133.29	218.87	400.71
50M	PS	0.07	0.18	1.29	8.46	62.22
	Exe	90.11	90.33	83.32	108.07	132.07
	P+PS+DS	90.18	90.51	84.61	116.53	194.29
	R	722.23	878.70	1414.94	2310.84	4173.17

Figure 13: Breakdown Mahif

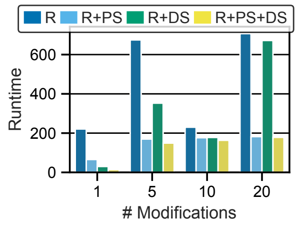


Figure 14: Mult. Modifications

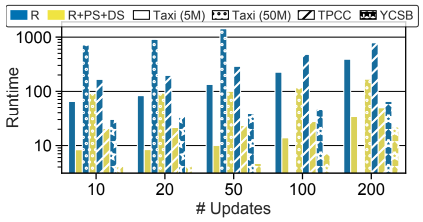


Figure 15: Optimization

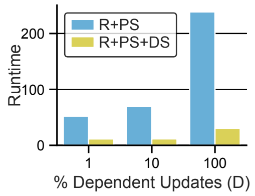


Figure 16: Dependent Updates

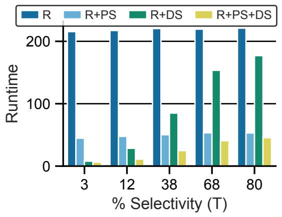


Figure 17: Affected Data

Reenactment with Data Slicing (R+DS): same as *R* except that we restrict reenactment to the part of data that is determined to be relevant by our data slicing optimization. **Reenactment with Program Slicing (R+PS)**: same as the *R* method except that we only reenact the subset of updates from a slice determined by our program slicing optimization. **Reenactment with Program Slicing + Data Slicing (R+PS+DS)**: we apply both optimizations.

Fig. 11 shows the naïve method's performance in comparison to *R+PS+DS*. Fig. 15 shows the runtime of reenactment (*R*) and reenactment with all optimizations enabled (*R+PS+DS*). Fig. 13 breaks down the cost of *R+PS+DS* into PS and Exe, and together they form the runtime of *R+PS+DS* which should be compared to the cost of *R* (`Reenact All`). Given the clear efficiency gains of *R+PS+DS* over *N* and *R*, we omit *N* and *R* from most remaining experiments and focus on evaluating our optimizations.

11.3 Optimization Methods

We now evaluate our optimization techniques varying the parameters introduced in Sec. 11.1 over update-only workloads to observe which workload characteristics benefit which optimization.

Varying Datasets (at D10). We vary number of updates and amount of tuples affected per update ($T0$, $T10$, $T25$). Overall, we see that our approach scales very well in dataset size. As the cost of program slicing is independent of the database size, larger datasets benefit much more from program slicing. For lower selectivity (Fig. 18), we see that *R+DS* is very competitive with *R+PS+DS* for the smaller Taxi dataset (5M) as reenactment of the entire history over a small relation and an even smaller proportion of affected input data is cheaper than the cost of solving the MILP problem for program slicing. Notably, for the YCSB dataset, the MILP cost exceeds the cost of *R+DS*, as data slicing is well-served by the physical correlation of the key used to update the data. For larger proportions of data to be reenacted (Fig. 19 and Fig. 20), *R+PS+DS* consistently outperforms *R+PS* and *R+DS*. The optimal case for *R+PS+DS* is when the size of the affected input data as determined by data slicing is large enough that calculating and reenacting over a slice is worth the execution cost of the generated MILP.

Varying Percentage of Dependent Updates (at T10). Fig. 16 demonstrates the effect the proportion of dependent updates (D) in a given history has on *R+PS*, and how the addition of data slicing (*R+PS+DS*) is an effective way to mitigate these effects. This experiment uses the $5M$ row taxi trip table, with defaults T10 (10% of tuples affected by modified updates) and U100 (100 updates in the history). As the proportion of dependent updates in the history increases, program slicing becomes less effective as more updates have to be included in the slice over the history. At D100 (100% of updates are dependent), program slicing is not beneficial at all, but still incurs the MILP solver cost. However, data slicing mitigates this effect, as the input to reenactment is filtered to include only a fraction of the data, making it more effective than *R+PS* at D100.

Varying Affected Data (at U100, D1). The effect of the percentage of tuples affected by the updates modified by the HWQ (T) is examined in Fig. 17. This experiment is executed for 100 updates on the taxi trips relation with $5M$ rows. For example, $T3$ means 3% of tuples (~ 150K out of 5M) are modified by the HWQ. The result demonstrates that varying T does not change the performance of *R+PS*, as the whole history is evaluated over the full input table in all cases. However, increasing T, increases the runtime of *R+DS* and *R+PS+DS* considerably as data slicing becomes less efficient due to the greater amount of input data that needs to be accessed during reenactment. However, at moderate selectivity (T68), *R+PS+DS* provides enough filtering over the history and data to be more performant than either optimization alone.

11.4 Mixed Workloads with Deletes and Inserts

We now consider workloads that contain deletions, inserts, and updates. Since deletes are handled in a similar fashion to updates, we mainly focus on evaluating the impact of the fraction of inserts on performance. We use the taxi trip tables for this experiment.

As shown in Fig. 22, *R+PS+DS* outperforms the other methods for mixed workloads. When comparing to similar workloads presented in Fig. 18 to 20, we see that introducing deletes and inserts into our workload in lieu of updates reduces the cost of reenactment and our optimizations. Note that delete statements result in fewer

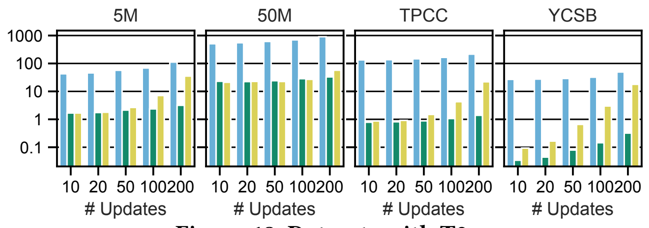


Figure 18: Datasets with T0

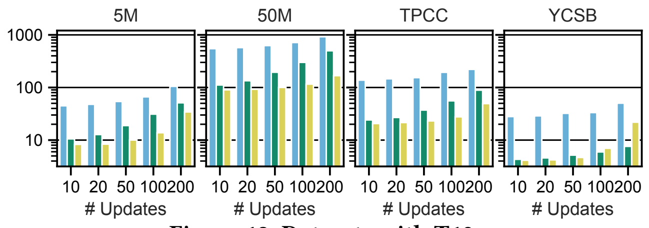


Figure 19: Datasets with T10

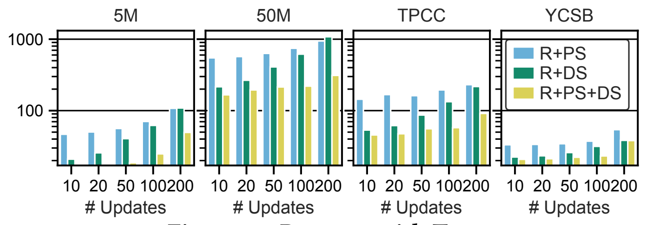


Figure 20: Datasets with T25

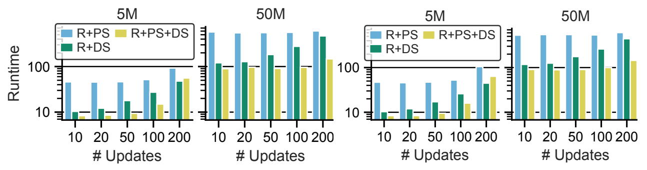


Figure 21: Inserts: I10, T10 Figure 22: Mixed: I10, X10, T10

constraints in the program slicing MILP than update statements. Inserts are much cheaper to process than program slicing an update as we are able to reenact the unsliced prefix of the history on a very small amount of tuples (only the tuples being inserted, typically a very small fraction of a given workload).

11.5 Varying the number of modifications

So far we have evaluated HWQs with a single modification. We now evaluate how multiple modifications affect the performance of Mahif and of the proposed optimizations.

Fig. 14 depicts the effect of changing the number of modifications per HWQ. Program slicing is more expensive than its single modification counterpart, given that we have to test each update by comparing the state of its symbolic tuple not only between H and $H[\mathcal{M}]$, but duplicating these histories while removing the update being tested, in order to not falsely classify an update as independent. This effectively quadruples the individual MILP program size over the single modification case. Data slicing also becomes more expensive as we employ the push down technique described in Sec. 6, which in turn creates a selection operator with a larger, more complicated condition. Recall that such selection conditions basically include some partial reenactment in order to capture every tuple that would be modified by modifications that occur after the first modification in the HWQ. That is, program slicing and data slicing are less efficient for multiple modifications.

The data from Fig. 14 shows a decrease in performance from a single modification to the multiple modification case. The nature of the modification (attributes updated, conditions, selectivity, dependencies across updates) significantly impacts the performance of *R+DS*, as evaluating the data slicing conditions pushed down through a long history is sometimes expensive. *R+PS+DS* remains an effective optimization compared to *R*, though its cost is higher than for single modifications. In part this is due to the effect of slicing the history which also reduces the size of conditions produced by pushing down data slicing conditions. It should be noted that as the amount of modifications grows, the program slicing time goes down as these modifications are inherently dependent. That being said, an inflection point is possible where the gains in program slicing execution speedup results in a slowdown from the longer history the data slicing conditions need to be pushed through.

11.6 Summary

Our approach outperforms the naïve method in most cases despite it not needing any additional storage. Even reenactment without optimizations is already considerably faster than the naïve method. The proposed optimizations are very effective, especially for large number of updates and larger databases. However, for small relations or very low selectivity, the cost of program slicing will outweigh the cost of reenactment or reenactment with data slicing. Data slicing is very effective for single modifications, but less so for multiple modifications, because the size and complexity of the data slicing conditions increases when these conditions are pushed through the updates upstream from a modification. Our experiments also show that our approach scales well with respect to database size.

12 CONCLUSIONS

We propose historical what-if queries, a new type of what-if queries which allow users to explore the effects of hypothetical changes to the transactional history of a database. Our system Mahif efficiently answers such queries using reenactment and two novel optimization techniques (program and data slicing) that exclude irrelevant data and updates from the computation. Our experimental evaluation demonstrates the effectiveness of our approach and of our optimizations. In future work, we will explore how to augment a user's HWQ based on information about unobserved external factors and dependencies between updates, e.g., if a HWQ deletes the statement creating a customer from history, then the statements creating the orders of this customer should be removed too. Furthermore, we will explore novel application of our symbolic evaluation technique such as proving equivalence of histories.

REFERENCES

[1] Serge Abiteboul and Gösta Grahne. 1985. Update semantics for incomplete databases. In *VLDB*. 1–12.
[2] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for Aggregate Queries. In *PODS*. 153–164.

[3] Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2016. Reenactment for Read-Committed Snapshot Isolation. In *CIKM*. 841–850.

[4] Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2018. Using reenactment to retroactively capture provenance for transactions. *TKDE* 30, 3 (2018), 599–612.

[5] Bahareh Sadat Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. 2014. A Generic Provenance Middleware for Database Queries, Updates, and Transactions. In *TaPP*.

[6] Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. 2000. Hypothetical Queries in an OLAP Environment. In *VLDB*. 220–231.

[7] Pierre Bourhis, Daniel Deutch, and Yuval Moskovitch. 2016. Analyzing data-centric applications: Why, what-if, and how-to. In *ICDE*. 779–790.

[8] Pierre Bourhis, Daniel Deutch, and Yuval Moskovitch. 2020. Equivalence-Invariant Algebraic Provenance for Hyperplane Update Queries. In *SIGMOD*. 415–429.

[9] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. *SIGARCH Comput Archit News* 42, 1 (2014), 239–254.

[10] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*. 316–330.

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *CACM* 56, 2 (2013), 82–90.

[12] Felix S. Campbell, Bahareh S. Arab, and Boris Glavic. 2022. Efficient Answering of Historical What-if Queries (extended version). (2022). arXiv:2203.12860 [cs.DB]

[13] James Cheney. 2007. Program slicing and data provenance. *IEEE Data Eng. Bull.* 30, 4 (2007), 22–28.

[14] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.

[16] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *TODS* 25, 2 (2000), 179–227.

[17] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction Repair for Multi-Version Concurrency Control. In *SIGMOD*. 235–250.

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *CACM* 54, 9 (2011), 69–77.

[19] Daniel Deutch, Zachary G Ives, Tova Milo, and Val Tannen. 2013. Caravan: Provisioning for What-If Analysis. In *CIDR*.

[20] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[21] Ronald Fagin, Gabriel M. Kuper, Jeffrey D. Ullman, and Moshe Y. Vardi. 1986. Updating Logical Databases. *Adv. Comput. Res.* 3 (1986), 1–18.

[22] Su Feng, Aaron Huber, Boris Glavic, and Oliver Kennedy. 2021. Efficient Uncertainty Tracking for Complex Queries with Attribute-level Bounds. In *SIGMOD*. 528–540.

[23] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. 31–40.

[24] Tomasz Imieliński and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *JACM* 31, 4 (1984), 761–791.

[25] Oliver Kennedy and Christoph Koch. 2010. PIP: A database system for great and small expectations. In *ICDE*. 157–168.

[26] James C King. 1976. Symbolic execution and program testing. *CACM* 19, 7 (1976), 385–394.

[27] Kasper Luckow, Corina S Păsăreanu, Matthew B Dwyer, Antonio Filieri, and Willem Visser. 2014. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*. 575–586.

[28] A. Meliou, W. Gatterbauer, K.F. Moore, and D. Suciu. 2010. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVLDB* 4, 1 (2010), 34–45.

[29] Alexandra Meliou and Dan Suciu. 2012. Tiresias: The Database Oracle for How-To Queries. In *SIGMOD*. 337–348.

[30] J. Pearl. 2000. *Causality: models, reasoning, and inference*. Cambridge Univ Pr.

[31] Sudeepa Roy and Dan Suciu. 2014. A formal approach to finding explanations for database queries. In *SIGMOD*.

[32] Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey Naughton, and Val Tannen. 2017. m-tables: Representing Missing Data. In *ICDT*, Vol. 68.

[33] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, Vol. 49. 530–541.

[34] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. Qfix: Diagnosing errors through query histories. In *SIGMOD*. 1369–1384.

[35] M. Weiser. 1981. Program slicing. *ICSE* (1981), 439–449.

[36] Marianne Winslett. 1986. Updating Logical Databases Containing Null Values. In *ICDT*, Vol. 243. 421–435.

[37] Ying Yang, Niccolo Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. 2015. Lenses: An on-demand approach to etl. *PVLDB* 8, 12 (2015), 1578–1589.

[38] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *PVLDB* 12, 11 (2019), 1276–1288.

[39] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. 1995. View maintenance in a warehousing environment. *SIGMOD Record* 24, 2 (1995), 316–327.