Answering Historical What-if Queries with Provenance, Reenactment, and Symbolic Execution

Bahareh Sadat Arab, Boris Glavic

Illinois Institute of Technology

barab@hawk.iit.edu, bglavic@iit.edu

Abstract

What-if queries predict how the results of an analysis would change based on hypothetical changes to a database. While a what-if query determines the effect of a hypothetical change on a query's result, it is often unclear how such a change could have been achieved limiting the practical applicability of such queries. We propose an alternative model for what-if queries where the user proposes a hypothetical change to past update operations. Answering such a query amounts to determining the effect of a hypothetical change to past operations on the current database state (or a query's result). We argue that such *historical what-if* queries are often easier to formulate for a user and lead to more actionable insights. In this paper, we take a first stab at answering historical what-if queries. We use *reenactment*, a declarative replay technique for transactional histories, to evaluate the effect of a modified history on the current database state. Furthermore, we statically analyze the provenance dependencies of a history to limit reenactment to transactions and data affected by a hypothetical change.

Employee

ID	Name	Calls	Sales	Bonus	
101	David	18	15	50	e_1
102	Mark	40	20	100	e_2
103	Susan	80	50	200	e_3
104	Robert	120	80	300	e_4

Figure 1: Running example database instance

T	U	SQL	Time
T_1	u_1	`UPDATE Employee SET Bonus=Bonus-50 WHERE Calls<30`	20
T_1	${u_1}'$	`UPDATE Employee SET Bonus=Bonus-50 WHERE Calls<50;`	20
T_1	c_1	`COMMIT;`	21
T_2	u_2	`UPDATE Employee SET Bonus=Bonus+50 WHERE Calls>100;`	22
T_2	c_2	`COMMIT;`	23
T_3	u_3	`UPDATE Employee SET Bonus=Bonus+100 WHERE Calls>30 AND Bonus<100`	24
T_3	c_3	`COMMIT;`	25

Figure 2: Transactional history implementing the new bonus policy and a hypothetical change the policy (update ${u_1}'$ replaces u_1)

Employee

ID	Name	Calls	Sales	Bonus	
101	David	18	15	0	${e_1}'$
102	Mark	40	20	100	e_2
103	Susan	80	50	200	e_3
104	Robert	120	80	350	${e_4}'$

Figure 3: Database state after executing the original history

Employee

ID	Name	Calls	Sales	Bonus	
101	David	18	15	0	${e_1}'$
102	Mark	40	20	150	${e_2}''$
103	Susan	80	50	200	e_3
104	Robert	120	80	350	${e_4}'$

Figure 4: Database state based on the historical what-if query

1. Introduction

What-if queries [4, 8] predict how a query's result would change based on hypothetical changes to a database. An example for a What-if query is: *"How would a 10% increase in sales affect my companys revenue this year?"* While a what-if query provides insight into how a change to data would affect the result of a query, it does not answer the important question of how such a change could have been achieved. For instance, how to increase sales by 10% is potentially a much harder question to answer than determining the effect of this increase on revenue. In fact, it may be much easier for the user to formulate what-if questions as changes to past actions (the history of the database) instead of changes to the current database state. For example: *"How would revenue be affected if we would have charged 5% interest for account overdrafts instead of 10%?"* Such a question can be interpreted as hypothetical changes to update operations or database transactions executed in the past, e.g., changing all past updates that applied interest to a user's account to use an interest rate of %5. Importantly, the answer to such a question is more likely to lead to actionable insights. If the hypothetical change to overdraft interest would result in a significant increase in revenue, then it is immediately clear how to implement this change — by decreasing the overdraft interest. This example illustrates the fundamental difference between regular what-if queries that propose a change to the data and *historical what-if queries*, the new type of predictive queries that we propose in this work, which postulate a change to past database operations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

TaPP 2017, June 22-23, 2017, Seattle, Washington.

Copyright remains with the owner/author(s).

Example 1. *The employee database shown in Figure 1 stores for each employee their* ID *and* Name. *In addition we store the number of sales* Calls *and* Sales *that each employee has made. The* Bonus *column records the amount of bonus an employee should receive. A new policy is introduced for updating an employee's bonus: the bonus for employees with less than 30 calls will be reduced by \$50, employees with more than 100 calls will receive a bonus increase of \$50, and employees with a bonus less than \$100 that have made more than 30 calls will receive an additional \$100. Three transactions T_1, T_2 and T_3 that implement this policy change are shown in Figure 2. For simplicity assume that these transactions were executed sequentially resulting in the database state shown in Figure 3. Manager Alice requests a report on how raising the minimum number of calls an employee has to make to avoid a bonus cut (Transaction T_1) would impact the total bonus payments made by the company. The bonus expenditure can be computed using a query Q_{total}:* `SELECT SUM(Bonus) FROM Employee` *(\$650 in our example). Alice's request can be modelled as a historical what-if query that changes the update u_1 in Transaction T_1 to the alterative ${u_1}'$ highlighted in red in Figure 2. Figure 4 shows the result of evaluating the what-if query by executing the modified transactional history over the database from Figure 1. The updated result of query Q_{total} would be \$700. Based on the predicted change Alice can decide whether to change the policy or not.*

In this paper, we introduce historical what-if queries and present an approach for evaluating such queries over a database backend. Our approach relies on reenactment [2, 3], a technique we have developed for replaying a transactional history (no matter whether real or hypothetical) using temporal queries. Such queries can be executed using any DBMS with support for *time travel* (e.g., Oracle, DB2, SQLsever). The naive solution to implement historical what-if queries is to reevaluate the modified history to produce an updated database state. Afterwards, we evaluate the user provided query over both the current database state and the updated database state and then determine what has changed. However, this is not feasible for large histories where replaying the whole history would be prohibitively expensive. Thus, we investigate optimizations of this approach. Specifically, we present a method that statically analyzes potential provenance dependencies among statements in a history using a method which borrows ideas from programming languages, i.e., symbolic execution [6, 14]. The proposed optimizations enables us exclude data from the replay (*data slicing*) and to avoid replaying parts of the history (*program slicing*).

2. Historical What-if Queries

We now introduce some basic notation and then define historical what-if queries. A transaction $T = (u_1, \ldots, u_n, c)$ is a sequence of updates (u_i) followed by a commit (c). The execution order of operations within a transaction is given by the order of these operations in the sequence. That is, if $i < j$ then u_i is executed before u_j. A transactional history is a pair $H = (\{T_1, \ldots, T_m\}, \leq_H)$ where each T_i is a transaction and $\leq_H$ is a total order over the operations of H's transactions that complies with the order of operations within each transaction, i.e., for each transaction $T_i = (u_1, \ldots, u_n, c)$ and $1 \leq i < j \leq n$ we have $u_i <_H u_j$ (ignoring the special case for commit operations). Given a database instance $\mathcal{D}$, we use $H(\mathcal{D})$ to denote the result of evaluating the history H over $\mathcal{D}$. Obviously, the precise result of this operation depends on the concurrency control protocol that is applied. For now we focus on serial histories, i.e., the history can be treated as a sequence of updates. For historical what-if queries we want to modify such histories, e.g., by replacing transactions. We use $m = T \leftarrow T'$, called a *modification*, to denote replacing transaction T with transaction T'. Note that for non-serial histories we would have to also provide an update to $\leq_H$ that removes all operations of T and declares how the operations of T' are interleaved with operations of other transactions from H. We use $H[m]$ to denote the history that is the result of applying modification m to history H. Similarly, the result of applying a sequence of modifications $\mathcal{M}$ to history H is denoted as $H[\mathcal{M}]$. We use $T.u \leftarrow u'$ to denote replacing update u of transaction T with u'. Typically, a user will be interested in how the change to the history affects a query Q's result. Keeping this in mind, instead of showing to the user the full query result over the database state produced by the modified history, we want to only show how $Q(H[\mathcal{M}](\mathcal{D}))$ differs from $Q(H(\mathcal{D}))$. One common method used in incremental query processing is to represent such a difference between two database states $\mathcal{D}$ and $\mathcal{D}'$ is as a set of delta tuples, i.e., tuples annotated with $+$ or $-$. An annotated tuple $+t$ denotes that t exists in $\mathcal{D}'$, but not in $\mathcal{D}$, and $-t$ denotes that t is in $\mathcal{D}$, but not in $\mathcal{D}'$. More formally, given two instances $\mathcal{D}$ and $\mathcal{D}'$ their *delta* $\Delta(\mathcal{D}, \mathcal{D}')$ is defined as:

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

Furthermore, we use $\Delta(Q, \mathcal{D}, \mathcal{D}')$ as a notational short cut for $\Delta(Q(\mathcal{D}), Q(\mathcal{D}'))$. Having defined modifications to histories and query result deltas, we are ready to define historical what-if queries.

Definition 1 (Historical What-if Query)**.** *A historical what-if query $\mathcal{H}$ is a tuple $(H, \mathcal{D}, \mathcal{M}, Q)$ where H is a transactional history run over a database instance $\mathcal{D}$, $\mathcal{M}$ denotes a sequence of modifications to H as introduced above, and Q is a query over the schema of $\mathcal{D}$. The answer to $\mathcal{H}$ is defined as:*

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

Example 2. *Let $\mathcal{D}$ and H be database shown in Figure 1 and history shown in Figure 3, respectively. Consider the modification $m_1 = T_1.u_1 \leftarrow u_1'$ using u_1 and u_1' as shown in Figure 2 which implements the policy change of increasing the minimum number of calls for avoiding a bonus cut from 30 to 50. Alice's historical what-if query from this example can be expressed as $\mathcal{H} = (H, \mathcal{D}, m_1, Q_{total})$ in our framework. Evaluating $H[m_1]$ results in a modified database instance as shown in Figure 4. For convenience we have highlighted modified tuple values. Employee Mark Smith receives a \$50 higher bonus under this new policy, because the bonus cut ($u_1{}'$) is applied (-\$50) which drops his bonus below \$100 and, thus, he becomes eligible to receive the additional bonus implemented by u_3. Evaluating the query Q_{total} and computing the delta as the answer for the what-if query we get:*

$$\{-(650), +(700)\}$$

Therefore, the total bonus payments would be \$50 dollar higher if the minimum number of calls would have been 50. Alice can now decide whether to change the bonus policy based on this outcome.

3. Answering Historical What-if Queries

To answer a historical what-if query, we have to compute $H[\mathcal{M}](\mathcal{D})$ and $H(\mathcal{D})$, compute the answer of query Q over these database versions, and then compute the symmetric difference between $Q(H[\mathcal{M}](\mathcal{D}))$ and $Q(H(\mathcal{D}))$ to determine the delta $\Delta(Q, H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$. Note that to formulate and evaluate historical what-if queries we have to have access to the transactional history of a database. For databases that support this feature (e.g., Oracle and DB2), we can use an audit log that stores a history of SQL commands executed in the past and for each command when it was executed and as part of which transaction. Otherwise, we can use triggers to capture SQL commands or exploit other logging mechanisms if available. Since $H(\mathcal{D})$ is the current state of the database, $Q(H(\mathcal{D}))$ can be computed by evaluating Q. The naive solution for answering a historical what-if query with modification $\mathcal{M}$ is

1. Use time travel to create a copy of $\mathcal{D}$ as of the start time of H.
2. Evaluate the modified history $H[\mathcal{M}]$ over the copy. If $H[\mathcal{M}]$ contains concurrent transactions, then we need to ensure that statements are grouped together as transactions and executed in the order given in the history. This can be achieved by opening multiple client connections (one per transaction) and executing statements in the order given by the history using the client connection associated with a transaction T to execute statements of transaction T.
3. Evaluate Q over both $H(\mathcal{D})$ and $H[\mathcal{M}](\mathcal{D})$ and compute their symmetric difference using SQL. Ignoring the $+$ and $-$ annotations, we compute $\Delta(R, R')$ as $(R - R') \cup (R' - R)$.

The naive method that creates a copy of the database is expensive as it requires additional storage and evaluating the modified history results in a large amount of write I/O. In the following we introduce three optimizations to the naive approach:

- **Reenactment and Incremental Maintenance** - we describe how to use incremental view maintenance techniques to speed up the process and how to avoid copying the database and running updates using *reenactment* [3].
- **Data Slicing** - we exclude parts of the database from replay that are not affected by $\mathcal{M}$. For that we need a provenance model that tracks dependencies of data on operations.

- **Program Slicing** - as long as the input to a statement u is the same in H and $H[\mathcal{M}]$ we avoid replaying u since its output will be the same in $H[\mathcal{M}]$ and H. Determining the subset of updates to replay is akin to a technique from programming language research called *program slicing* [17] that determines which part of a program affects the value of a set of variables at a certain position of a program (e.g., an output).

4. Reenactment and Incremental Maintenance

The naive approach evaluates the query Q of a historical what-if query $\mathcal{H}$ over the current database state $H(\mathcal{D})$ and over $H[\mathcal{M}](\mathcal{D})$ and then computes the delta of the results. Alternatively, we can treat the computation of $\Delta(Q, H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$ as an incremental view maintenance problem: we compute $Q(H(\mathcal{D}))$ and incrementally maintain this result based on $\Delta(H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$ using any incremental view maintenance technique (e.g., [12]). To further improve performance, we focus on the computation of $\Delta(H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$. We apply *reenactment*, a declarative replay technique for transactions that we have introduced in previous work [3], to compute $H[\mathcal{M}](\mathcal{D})$. Given a transactional history H, reenactment generates a *reenactment query* $\mathbb{R}(H)$ which is equivalent to the history under standard bag and set semantics and was proven to have the same provenance. Importantly, reenactment queries can be expressed in standard SQL using time travel and can be used to replay only a part of a history. Thus, we can run the query $\mathbb{R}(H[\mathcal{M}])$ over $\mathcal{D}$ to generate $H[\mathcal{M}](\mathcal{D})$. Compared to the naive approach this has the advantage that there is no need to copy the database and we avoid the logging overhead caused by running update operations. More importantly, it enables the data slicing and program slicing optimizations that we discuss next.

5. Data Slicing

Intuitively, any difference between $H(\mathcal{D})$ and $H[\mathcal{M}](\mathcal{D})$ has to be caused by a difference between H and $H[\mathcal{M}]$. For simplicity of exposition consider a single modification $m = T.u \leftarrow u'$. For a tuple t to appear in $\Delta(H(\mathcal{D}), H[m](\mathcal{D}))$, it has to be affected by either u or u' or both (but in different ways). Given an expressive enough provenance model that allows us to determine based on the provenance of a tuple t which updates of a history affected it (e.g., our MV-semiring model [3]), we can filter the inputs to the delta computation based on their provenance - only retaining tuples that have either u or u' in their provenance. However, this would require us to compute both $H(\mathcal{D})$ and $H[m]$ with provenance tracking and the overhead associated with that may easily outweigh the advantage of reducing the input size of the delta computation. Since only tuples that match the condition of an update operation (the update's `WHERE` clause) can be affected by the update, a conservative overestimation of the set of tuples in $\Delta(H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$ is the set of tuples that are derived from tuples affected by u in the original history or u' in the modified history. Thus, the original version of these tuples in $\mathcal{D}$ have to either match the condition of u or of u'. Let $\theta(u)$ denote the condition of an update u. To compute the delta, we can reenact both $H(\mathcal{D})$ and $H[\mathcal{M}](\mathcal{D})$ and filter the input to these reenactment queries using a condition $\theta(u) \vee \theta(u')$.

Example 3. *To compute $\Delta(Q, H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$ for the Example 2, we first construct reenactment queries for updates u_1, u_2, u_3 and u_1', u_2, u_3. Based on the conditions of u_1 and u_1', we add a condition* `Calls<30 OR Calls<50` *to the inputs of these queries (adding a* `WHERE` *clause). In our example, this would exclude tuples e_3 and e_4 from reenactment since any tuple in $H(\mathcal{D})$ and $H[\mathcal{M}](\mathcal{D})$ derived from these tuples will occur in the result of both histories and, thus, not be part of the delta.*

A similar optimization can be applied for deletes. We leave the study of how this optimization can be extended to support inserts with queries (i.e., `INSERT INTO ... SELECT`) for future work.

6. Program Slicing

In addition to data slicing, we can also exclude updates from reenactment if their output is the same in H and $H[\mathcal{M}]$. Here we only consider deterministic updates, thus, if the input of an update is the same in both histories then so is its output. Intuitively, the input of an update u can only differ if it contains one or more tuples affected by an update modified by the historical what-if query. Given a history H, modification $m = T.u_{orig} \leftarrow u_{new}$, and database instance $\mathcal{D}$, we say an update $u \in H$ *depends* on m according to $\mathcal{D}$ if there exists a tuple affected by u that is also affected by either u_{orig} or u_{new}. Dependency can be extended to a set of modifications $\mathcal{M}$ in the obvious way. The notion "affected by" can be made more precise using our MV-semiring provenance model that records which updates did affect a tuple t or one of the tuples in t's provenance. However, for reasons of space we do not present the full formalization here. Intuitively, only updates that depend on a modification have to be reenacted to compute the delta between $H(\mathcal{D})$ and $H[\mathcal{M}](\mathcal{D})$. We propose to statically analyze the history to find a set of updates that can be safely excluded from reenactment. The result of this analysis is a conservative overestimation of the real set of dependent updates. To this end we use symbolic execution [11] to determine whether an update may depend on a modification. We start from a symbolic instance that consists of a single tuple of variables, i.e., we treat the input as a c-table [10]. The effect of an update on a symbolic instance are encoded as constraints over the variables of the instance. To determine whether an update u depends on a modification we test satisfiability of such constraints using a constraint solver. For reasons of space we only illustrate this approach using an example.

Example 4. *Consider symbolic execution for the running example. We start with a symbolic table with a single tuple of variables*

ID	Name	Calls	Sales	Bonus
x_i	x_n	x_c	x_s	x_b

Update u_1 reduces an employee's bonus by \$50 if the number of calls is less than 30. To apply this update we replace the value of attribute Bonus *with a symbolic expression that encodes the conditional update. Here $\otimes$ denotes a pairing of a value with a condition (similar to the tensor construction of [1]).*

ID	Name	Calls	Sales	Bonus
x_i	x_n	x_c	x_s	$x_b + (-50 \otimes (x_c < 30))$

To determine whether an update u depends on another update u' we have to check whether conditions of the two updates are mutually satisfiable, i.e., there may exist a tuple that is updated by both updates. For that we replace references to attributes in conditions with symbolic expressions that model the effect updates between u and u'. For instance, the symbolic input for update u_3 computed by the process outlined above would be

ID	Name	Calls	Sales	Bonus
x_i	x_n	x_c	x_s	$x_b + (-50 \otimes (x_c < 30)) + (50 \otimes (x_c > 100))$

To check whether u_3 may depend on u_1 we have to check whether $(Calls < 30) \wedge (Calls > 30 \wedge Bonus < 100)$ is satisfiable. Substituting the expressions from the symbolic instances we get:

$$(x_c < 30) \wedge (x_c > 30) \wedge ((x_b + (-50 \otimes (x_c < 30)) + (50 \otimes (x_c > 100))) < 100)$$

This formula is not satisfiable (since $x_c < 30 \wedge x_c > 30$ is unsatisfiable) and, thus, we know that u_3 is not dependent on u_1.

7. What-if Algorithm

We now give a high-level overview of our algorithm for answering a historical what-if query $\mathcal{H} = (H, \mathcal{D}, \mathcal{M}, Q)$ that employs the

Algorithm 1 Answering Historical What-if Query

```
1: procedure WhatIf($H, \mathcal{D}, Q, \mathcal{M}$)
2:     $dep \leftarrow$ ComputeDependencies($H, \mathcal{M}$)
3:     $\mathbb{R}(H) \leftarrow$ GenReenactmentQuery($H, dep$)
4:     $\mathbb{R}(H)^* \leftarrow$ ApplyDataSlicing($\mathcal{M}, \mathbb{R}(H)$)
5:     $\mathbb{R}(H[\mathcal{M}]) \leftarrow$ GenReenactmentQuery($H[\mathcal{M}], dep$)
6:     $\mathbb{R}(H[\mathcal{M}])^* \leftarrow$ ApplyDataSlicing($\mathcal{M}, \mathbb{R}(H[\mathcal{M}])$)
7:     return SymmetricDiff($Q, \mathbb{R}(H)^*, \mathbb{R}(H[\mathcal{M}])^*$)
8: end procedure
```

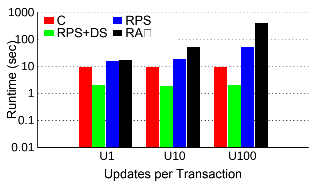

Figure 5: Updates/Transaction

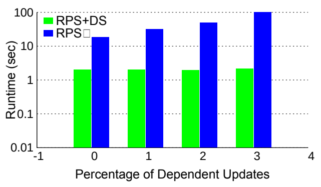

Figure 6: Dependent Updates

optimizations introduced in the previous sections. We first compute dep, an overestimate of the set of dependent updates using the approach described in Section 6. We then generate reenactment queries for H and $H[\mathcal{M}]$ restricted to dep and add selection conditions to implement the data slicing optimization (Section 5). Finally, function `SymmetricDiff` takes a query Q and the two reenactment queries as input, computes $\Delta(H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$ using the reenactment queries, and incrementally maintains $Q(H(\mathcal{D}))$ to compute the answer to $\mathcal{H}$.

8. Related Work

What-if queries determine the effect of a hypothetical change to an input database on the results of query. What-if queries [4, 8, 18] are often realized using incremental view maintenance to avoid having to reevaluate the query over the full input including the hypothetical changes. The how-to queries of Tiresias [15] determine how to translate a change to a query result into modifications of the input data. The QFix system [16] 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 relation with a single symbolic tuple, i.e., the number of constraints we produce is independent of the database instance size. The connection of provenance and program slicing was first observed in [7]. We present a method that statically analyzes potential provenance dependencies among statements in the history using a method which borrows ideas from symbolic execution [6, 11, 14], constraint databases [9, 13], program slicing [17], and expressive provenance models [1].

9. Experiments

We have conducted preliminary experiments using a proof-of-concept implementation to evaluate the performance of our approach, study the effectiveness of the proposed optimizations, and compare against a naive approach that evaluates the modified history over a copy of the database.

Datasets and Workload. We use a single relation with five numeric columns with 1M tuples. We consider histories of 10 transactions where parameter U is the number of update statements per transaction. We consider what-if queries that modify a single update of the first transaction of a history. Each update statement affects 10 tuples which are chosen randomly based on a range condition over the primary key attribute (e.g., `WHERE ID>30 AND ID<41`). All experiments were executed 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 commercial DBMS X (name omitted due to licensing restrictions) executing transactions using the snapshot isolation concurrency control protocol [5]. Each experiment was repeated 100 times, we report average runtimes.

Compared Methods. We compare the following methods for evaluating what-if queries. **Copy (C)**: creates a copy of the database as of the start time of the transaction which is modified by the what-if query, replays $H[\mathcal{M}]$ over this copy to compute $H[\mathcal{M}](\mathcal{D})$, and then computes the delta $\Delta(Q, H(\mathcal{D}), H[\mathcal{M}](\mathcal{D}))$. **Reenact All (RA)** creates a reenactment queries for H and $H[\mathcal{M}]$. For both queries we exclude statements that executed before the statement modified by the what-if query. We then use time travel to access the database state as of the time the modified update was executed, run both reenactment queries over this instance, and then compute the delta. **Reenact Program Slicing (RPS)**: same as the previous method except that we only reenact a subset of the histories determined by our program slicing optimization. **Reenact Program Slicing + Data Slicing (RPS+DS)**: same as the previous method except that we also apply the data slicing technique.

Updates/Transaction. In this experiment we vary the number of updates per transaction ($U1$, $U10$, and $U100$). For instance, $U10$ denotes a history consisting of 10 transactions with 10 updates each. The runtimes of the different methods for answering a historical what-if query are shown in Figure 5. Our fully optimized method *RPS+DS* has the best performance outperforming the second best method (C) by a factor of ~ 3. The cost of C is dominated by creating the copy of the input database and computing the delta while reexecuting the history has negligible cost (~ 0.5s for U100). For *RPS+DS* the cost is dominated by the cost of time travel. The large number of update statements in transactions considerably degrades the performance of the *RPS* method that does not use data prefiltering and the *RA* method which has to reenact all updates. While the cost of reenactment itself is tolerable, computing the delta over two large intermediate results is costly.

Dependent Updates/Transaction. In this experiment, we use 10 transactions with 100 updates ($U100$) and vary the number of updates that depend on the modified update from 1% ($D1$) of the total number of updates up to 50% ($D50$). The results are shown in Figure 6. We compare *RPS+DS* and *RPS*, the two methods that exploit dependencies. Both method's runtime increases with increasing number of dependencies. The performance of method *RPS+DS* is dominated by the cost of time travel for this workload and, thus, reducing the number of updates that have to be reenacted has a less pronounced effect.

10. Conclusions

Traditional what-if queries determine how a hypothetical change to data affects a query result. However, it is often completely unclear how such a change could be achieved in the first place. We present our vision for historical what-if queries, a novel type of predictive queries that evaluate the effect of a hypothetical change to a database's history. Historical what-if queries overcome the aforementioned drawback of traditional what-if queries by focusing on "what could I have done differently" instead of "what if the state of the world would be different". We present an implementation of historical what-if queries that exploits our declarative replay technique called reenactment and uses optimizations based on static analysis of potential provenance dependencies to improve performance. Our preliminary experiments demonstrate that this approach is feasible.

References

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

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

[3] B. S. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, pages 841–850, 2016.

[4] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *VLDB*, pages 220–231, 2000.

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

[6] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. *SIGARCH Comput Archit News*, 42 (1):239–254, 2014.

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

[8] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

[9] M. T. Gómez-López and R. M. Gasca. Using constraint programming in selection operators for constraint databases. *Expert Syst Appl*, 41 (15):6773–6785, 2014.

[10] T. Imieliński and W. Lipski Jr. Incomplete Information in Relational Databases. *JACM*, 31(4):761–791, 1984.

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

[12] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014.

[13] G. Kuper, L. Libkin, and J. Paredaens. *Constraint databases*. 2013.

[14] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*, pages 575–586, 2014.

[15] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[16] X. Wang, A. Meliou, and E. Wu. Qfix: Diagnosing errors through query histories. In *SIGMOD*, pages 1369–1384, 2017.

[17] M. Weiser. Program slicing. *ICSE*, pages 439–449, 1981.

[18] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *SIGMOD Record*, 24(2):316–327, 1995.