Equivalence-Invariant Algebraic Provenance for Hyperplane Update Queries

Pierre Bourhis, Daniel Deutch, Yuval Moskovitch

Equivalence-Invariant Algebraic Provenance for Hyperplane Update Queries

Pierre Bourhis
CNRS, UMR 9189 - CRIStAL
Lille, France

Daniel Deutch
Tel Aviv University
Tel-Aviv, Israel

Yuval Moskovitch
Tel Aviv University
Tel-Aviv, Israel

ABSTRACT

The algebraic approach for provenance tracking, originating in the semiring model of Green et. al, has proven useful as an abstract way of handling metadata. Commutative Semirings were shown to be the "correct" algebraic structure for Union of Conjunctive Queries, in the sense that its use allows provenance to be invariant under certain expected query equivalence axioms.

In this paper we present the first (to our knowledge) algebraic provenance model, for a fragment of update queries, that is invariant under set equivalence. The fragment that we focus on is that of hyperplane queries, previously studied in multiple lines of work. Our algebraic provenance structure and corresponding provenance-aware semantics are based on the sound and complete axiomatization of Karabeg and Vianu. We demonstrate that our construction can guide the design of concrete provenance model instances for different applications. We further study the efficient generation and storage of provenance for hyperplane update queries. We show that a naive algorithm can lead to an exponentially large provenance expression, but remedy this by presenting a normal form which we show may be efficiently computed alongside query evaluation. We experimentally study the performance of our solution and demonstrate its scalability and usefulness, and in particular the effectiveness of our normal form representation.

KEYWORDS

Provenance, Transactions

1 INTRODUCTION

The tracking of *provenance* for database queries has been extensively studied in the past years (see e.g. [5, 10, 11, 21]). In a nutshell, data provenance captures details of the computation that took place and resulted in the generation of each output data item. Multiple models for data provenance have been proposed, for multiple query languages such as the (positive) relational algebra, datalog (see [21]), data-intensive workflows (e.g., [12, 31]) data mining [19], and data-centric applications [14]. Provenance has been proven useful for managing access control, trust, hypothetical reasoning, view maintenance and debugging (see [8, 13, 17, 20, 21]).

The approach advocated by [21] is based on designing *algebraic* provenance structures whose equivalence axioms are based on equivalences in the formalism for which provenance is designed to be tracked. This guarantees that by design, equivalent queries/programs in the formalism of interest will have equivalent provenance for their output. In a sense, this means that provenance captures the "essence of computation" that has been performed. The commutative semiring model of [21] achieves this property for the positive relational algebra; several extensions have been studied [5, 6, 24] for different query languages.

In this paper we focus on a fragment of *update queries* and sequences thereof (which we refer to as "transactions"), and propose a novel algebraic provenance model. The fragment of update queries that we focus on is that of *hyperplane queries*, introduced in [3] as simple building blocks of transactions. The provenance annotations in our model are initially assigned to both queries and tuples; those assigned to queries are propagated to the tuples that these queries affect, so that the result of applying an annotated transaction is an annotated database. Then, in a similar vein to the commutative semiring model mimicking the equivalence axioms of positive relational algebra, our model is based on the sound and complete axiomatization for set equivalence of transactions in [23]. Namely, we start with a most generic structure that uses abstract operations to capture the effect of each type of update query, and then introduce, for each of the axioms in [23], a corresponding axiom in our algebraic structure. As we will show, this leads to a provenance framework that has the following favourable property: two transactions are "provenance-equivalent", i.e. their application on every input database yields the same annotated database, if and only if they are set-equivalent. This means that provenance in our framework is independent of the particular way that the transaction is executed and of any optimizations that may take place. To our knowledge, ours is the first provenance model to satisfy this property for transactions (see discussion of previously proposed models in Sections 3.3 and 7).

To demonstrate the usefulness of our generic model, we give examples of concrete interpretations for the algebraic operators that satisfy the axioms and allow provenance tracking in multiple domains. We also give examples of interpretations that *do not* work: semantics that do not satisfy the

axioms, and thus lead to different annotations being assigned by equivalent transactions. In particular, we show that this phenomenon occurs for a previously proposed model called MV-semirings [6].

Importantly, while our generic structure is quite complex, we provide a "prescription" for building instances of it. This is achieved by establishing a connection with the commutative semiring model: we show that for a simple-to-define class of commutative semirings (see Theorem 4.2 for details), their operators can be easily extended to define operators for our model that do satisfy the axioms. We use this method to show that our model generalizes the boolean provenance model of [8], as well as an example for propagating access control credentials through transactions (see Section 4.1).

We then turn to the problem of efficient provenance generation and storage. The model definition already entails an algorithm for provenance generation, but we show that it may lead to an exponential blowup of the provenance size with respect to the transaction length (number of queries). We show that this blowup may be avoided: we propose a "normal form" structure for provenance, and show that every provenance expression obtained by applying a sequence of hyperplane updates may be transformed to this structure. The expressions that we obtain in this structure are far more compact: they are in fact linear in the size of the transaction and input database. Furthermore, we show that we can generate expressions in this structure on-the-fly during query evaluation, avoiding a detour through the exponentially large representation.

Finally, we present an implementation of our provenance framework and algorithms, and experimentally study its properties using the TPC benchmark as well as a synthetic dataset. We measure the time and space overhead that provenance tracking incurs, and the time it takes to use provenance for deletion propagation, by assigning truth values to its variables. Our experimental results (1) validate the usefulness of provenance in this context: deletion propagation through the provenance is much faster than the alternative of rerunning the transaction over a modified database; (2) show that using our construction: the axioms and the rewrite of provenance into its normal form significantly reduces the provenance size, and is thereby beneficial both for provenance maintenance and for its use in the context of deletion propagation.

The rest of this paper is organized as follows. We overview the hyperplane update queries model that we focus on in Section 2. We introduce our provenance model and axiomatization in Section 3 and show concrete applications, as well as connection to previously propsoed formalisms, in Section 4. We study efficient provenance generation and storage in Section 5. We overview related work in Section 7 and conclude in Section 8.

2 PRELIMINARIES

Our goal is to define an algebraic provenance model for updates. In this paper, we focus on the class of "domain-based" updates defined in [3]. This class is a standard model that was studied e.g., in [23, 28, 29]. Importantly, [23] has proposed a sound and complete axiomatization for this fragment, which will serve as a basis for our algebraic provenance model. We describe the class of "domain-based" transactions in a datalog-like language, similar to the one in [8].

Relational Databases. A *relational schema* is defined over a set of relational names. A *relation* has a relation name and a function associating with this name a set of attributes denoted by $\text{att}(R)$. Let $\mathcal{V}$ be an infinite set of values. A tuple t of relation R is a function associating with each attribute of R, a value of $\mathcal{V}$. An *instance* I of a relation R is a set of tuples. A database D of a relational schema associates with each relation name R in the schema an instance, denoted by $R(D)$.

Hyperplane Updates queries. We next recall the definition of update queries from [3] for the class of "domain-based" transactions, where the selection of tuples only involves the inspection of individual attribute values for each tuple. We restrict the updates queries of [8] to those equivalent to a member of this class.

To this end, we use the notation $R(\mathbf{u})$ where $\mathbf{u}$ is a tuple with the same arity as R, that may contain constants and variables. A variable A in $\mathbf{u}$ may further be associated with a disequality expression $[A \neq a]$, restricting assignments so that the attribute in the corresponding position may not be assigned the value a. We say that a tuple $t \in R$ satisfies $\mathbf{u}$ and write $t \vDash \mathbf{u}$ if t corresponds to an instantiation of the variables of $\mathbf{u}$ that satisfy the conditions.

Example 2.1. Figure 1a shows a fragment of a *products* table in an E-commerce application. It includes information about the products in stock, their categories and price (ignore the annotations next to tuples for now). The following is an hyperplane query used to describe all products in the Sport category except for the "Kids mountain bike":

$$products([p \neq \text{“Kids mnt bike”}], \text{“Sport"}, c)\text{:-}$$

The tuple $products(\text{“Tennis Racket"}, \text{“Sport"}, \$70)$ satisfies the conditions specified in the query.

Insertion. An insertion query Q is an expression $R^{+}(\mathbf{u})$:- where $\mathbf{u}$ is a tuple of constants with the same arity as R. The effect of Q applied to a database D, denoted by $Q(D)$, is the insertion of $\mathbf{u}$ to $R(D)$.

Example 2.2. Consider again the database given in Figure 1a. The query

$$Products^{+}(\text{“Lego bricks"}, \text{“Kids"}, \$90)\text{:-}$$

Product	Category	Price	
Kids mnt bike	Sport	\$120	p_1
Tennis Racket	Sport	\$70	p_2
Kids mnt bike	Kids	\$120	p_3
Children sneakers	Fashion	\$40	p_4

(a) Initial Table

Product	Category	Price	
Kids mnt bike	Bicycles	\$120	$(p_1 + p_3) \cdot_M p$
Tennis Racket	Sport	\$70	p_2
Lego bricks	Kids	\$90	p

(b) Updated Table

Figure 1: Products Table

is an example of an insertion query, adding the tuple ("Lego bricks", "Kids", \$90) to the *Products* table.

Note that each insertion query inserts a single tuple, as in [23]; we will consider transactions as means for inserting a bulk of tuples.

Deletion. A deletion query Q is an expression $R^-(\mathbf{u})$:-, where $\mathbf{u}$ is a tuple with the same arity as R, that may contain constants and variables, possibly associated with disequalities. $Q(D)$ is the resulting database obtained by from D by deleting all tuples of its relation R that satisfy $\mathbf{u}$.

Example 2.3. Reconsider the database fragment presented in Figure 1a. The query

$$Products^-(a, \text{"Fashion"}, b)\text{:-}$$

deletes all tuples in the fashion category.

Modification. A modification query Q is an expression $R^M(\mathbf{u_1}, \mathbf{u_2})$:- , where $\mathbf{u_1} = (u_0^1, \ldots, u_n^1)$ and $\mathbf{u_2} = (u_0^2, \ldots, u_n^2)$ have the same arity as R and may contain variables and constants such that either $u_i^1 = u_i^2$ (and then the value for this attribute remains intact) or u_i^2 is a constant (and then the value is changed to u_i^2). I.e. the constants present in $\mathbf{u_2}$ which are different from the corresponding variables/constants in $\mathbf{u_1}$ indicate how instantiations of $\mathbf{u_1}$ are modified. The result of applying Q to a database D is defined as follows: for each valid assignment to $\mathbf{u_1}$ and $\mathbf{u_2}$, the tuple t of R whose values correspond to the instantiation of $\mathbf{u_1}$ is deleted; the tuple t' whose values correspond to the instantiation of $\mathbf{u_2}$ is inserted. We use $t \rightsquigarrow t'$ to denote that t was updated to t'.

Example 2.4. The query

$$Products^M(\text{"Kids mnt bike"}, a, b, \text{"Kids mnt bike"}, \text{"Bicycles"}, b)\text{:-}$$

is a modification query. Applying the query to the database fragment shown in Figure 1a results in an update of the category (second attribute) of the product "Kids mnt bike" to "Bicycles". Namely, we have that ("Kids mnt bike", "Sport", \$120)$\rightsquigarrow$ ("Kids mnt bike", "Bicycles", \$120) and ("Kids mnt bike", "Kids", \$120)$\rightsquigarrow$ ("Kids mnt bike", "Bicycles", \$120).

A *transaction* is a sequence of update queries. Its semantics with respect to a given database is that the update queries are applied sequentially, with each query in the sequence being applied to the result of the transaction prefix that preceded it.

The result of applying the update queries from Examples 2.2, 2.3 and 2.4 as a sequence to our example relation, is shown in Figure 1b.

3 PROVENANCE MODEL

We define an algebraic provenance model for transactions whose design follows the following principle: introduce the most general model that is still insensitive to rewriting under (set) equivalence. We will define the structure, and then the propagation of provenance in the structure through update queries. We will show that our provenance model captures the essence of computation defined by the queries, rather than the query structure.

3.1 Algebraic Structure

In a similar vein to the semiring construction of [21], we start with a basic set of annotations X. These could be thought of as identifiers, which in our case will be associated not only with tuples but also with individual queries, and propagated to the tuples they "touch". We then introduce a structure called $UP[X]$ ("UP" standing for updates) as follows. As a most general structure, we will use six algebraic operations: $+_I$ and $+_D$ which will serve as abstract operations to capture provenance for *insertion* and *deletion* respectively; $-_M$ that will be used in the context of *modification*, to capture the original tuple (before modification); and $+_M$ and $\cdot_M$ that will be used for the tuple after modification. Last, we will use $+$ (and Σ for summation over a set), to capture disjunction originated in the query.

We also introduce a unique element denoted as 0, that intuitively will be used to denote an absent tuple, when used as tuple annotation, or the fact that an updated query has not taken place, used as query annotation. Expressions in $UP[X]$ are then comprised of any combination of elements in $X \cup \{0\}$ using these five operations; we will sometime refer to such expressions as *formulas*.

We still keep these operations abstract, in that we do not impose any concrete semantics or further equivalences; we will do both later.

Annotated Relations. Let R be a (standard) relation schema and let $tup(R)$ be the set of all tuples conforming to R. Given a set of annotations X, we use the term $UP[X]$-relation R to denote a function from $tup(R)$ to $UP[X]$; we thus use $R(t)$ to denote the annotation of a tuple t in R. The set of all tuples not mapped to 0 is called the support of R (we will also say that they are "in" R). A set of $UP[X]$-relations (associated with a schema, in the standard sense) is an $UP[X]$-database.

Annotated Update Queries. We include an *annotation* as part of update query specifications. Intuitively, this annotation may stand for an identifier of the query, or any other meta-data associated with it. We fix a set P of symbols to be used as query *annotations* and attach them to the heads of queries. For instance, the head of a provenance-aware insertion query has the form $R^{+,p}(\mathbf{u})$:-, where $p \in P$ is the annotation; similarly for deletion and modification. For example, the query $Products^{+,p}$("Lego bricks", "Kids", \$90):- is an annotated insertion query with the annotation being p.

Provenance for hyperplanes queries. We are now ready to define provenance for queries. In what follows, let R be an $UP[X]$-relation. We consider different types of update queries Q, and use R' for the $UP[X]$-relation that is the result of applying Q to R. The resulting $UP[X]$-relation is as follows.

- If $Q \equiv R^{+,p}(t)$:- then we define $R'(t) = R(t) +_I p$, and for each $t' \neq t$ we define $R'(t') = R(t')$.
- If $Q \equiv R^{-,p}(\mathbf{u})$:-, then $R'(t) = R(t) +_D p$ for each tuple $t \in R,\ t \vDash \mathbf{u}$ and $R'(t') = R(t')$ otherwise.
- If $Q \equiv R^{M,p}(\mathbf{u_1}, \mathbf{u_2})$:-, then $R'(t_1) = R(t_1) -_M p$ for each $t_1 \in R\ t_1 \vDash \mathbf{u_1}$ and $R'(t_2) = R(t_2) +_M ((\sum_{t_1 \rightsquigarrow t_2} R(t_1)) \cdot_M p)$, for each t_2 s.t. $\exists t_1 \vDash \mathbf{u_1}\ t_1 \rightsquigarrow t_2$. The operator $\sum$ here stands for a disjunctive operator associated to the query. In particular, it is different than $+_M$ and $+_I$. Otherwise $R'(t) = R(t)$.

If $t \notin R$ then $R(t) = 0$. Thus we define $\forall a \in X$

- $0\ op\ a = 0$ if $op \in \{-_M, +_D\}$
- $0\ op\ a = a$ if $op \in \{+_M, +_I\}$
- $a\ op\ 0 = a$ for $op \in \{+_I, +_M, -_M, +_D\}$
- $a \cdot_M 0 = 0 \cdot_M a = 0$

Intuitively, if $t \notin R$, then deleting or modifying t does not change R, and t remains absent from R, thus $0\ op\ a = 0$ if $op \in \{-_M, +_D\}$. The existence of an inserted tuple $t \notin R$ by a query annotated with a depends only on a and thus $0 +_I a = a$. Similarly for an updated tuple $t \rightsquigarrow t'$ for $t' \notin R$. In a way, the righthand element of the operators $+_I, +_M, -_M$ and $+_D$ may be interpreted as a condition for the update, i.e., if the condition is 0, the update did not take place. Therefore $a\ op\ 0 = a$ for $op \in \{+_I, +_M, -_M, +_D\}$. Finally, the expression $a \cdot_M b$ is used to capture the fact that a tuple annotated by a is updated by a query annotated by b to produce an updated tuple. If $a = 0$, the tuple is not in the database; if $b = 0$ the query has not taken place. In both cases the updated tuple was not generated, thus $a \cdot_M 0 = 0 \cdot_M a = 0$.

Example 3.1. Reconsider the database fragment shown in Figure 1a, and the annotated update query:

$Products^{M,p}$("Kids mnt bike", a, b,
"Kids mnt bike", "Bicycles", b):-

By applying the query, the tuple ("Kids mnt bike", Sport, \$120), annotated by p_1 and the tuple ("Kids mnt bike", Kids, \$120), annotated by p_3 are updated to ("Kids mnt bike", Bicycles, \$120), which is not in the database, thus annotated by 0. As a result the new tuples annotations are $p_1 -_M p$, $p_3 -_M p$ and $0 +_M (p_1 +_M p_3) \cdot_M p = (p_1 +_M p_3) \cdot_M p$ respectively.

Provenance of a transaction. For a given transaction, we annotate it – i.e. all of its update queries – with an annotation p (a single annotation is used per transaction, reflecting the grouping of queries to a transaction). We apply the queries in the transaction one by one, using the above definitions to compute the provenance of tuples they "touch": the i'th update is applied on the annotated database obtained from applying the first $i-1$ updates.

Example 3.2. The following is an example of a transaction over the database fragment given in Figure 1a:

$Products^{M,p}$("Kids mnt bike", "Kids", c,
"Kids mnt bike", "Sport", c):-

$Products^{M,p}$("Kids mnt bike", "Sport", c,
"Kids mnt bike", "Bicycles", c):-

The resulting database from the transaction contains the tuple $Products$("Kids mnt bike", "Kids", \$120) with the provenance annotations $p_3 -_M p$ due to the first query. The annotation of the tuple $Products$("Kids mnt bike", "Sport", \$120) is $(p_1 +_M (p_3 \cdot_M p)) -_M p$, where the part in the parentheses is the result of the first query. Finally, the annotation of the tuple $Products$(Kids mnt bike, Bicycles, \$120) is $0 +_M ((p_1 +_M (p_3 \cdot_M p)) \cdot p)$, where the sub-expression $p_1 +_M (p_3 \cdot_M p)$ comes from the provenance annotation of the tuple $Products$("Kids mnt bike", "Sport", \$120) after the execution of the first query.

3.2 Algebraic Axiomatization

The operations we have introduced so far lead to a very abstract notion of provenance tracking which essentially requires full tracking of the operation of the update queries that took place, without allowing for any simplifications.

A fundamental question is what simplifications can take place, while still capturing the "essence" of updates that took

place? To this end, we note that [23] has introduced a sound and complete axiomatization of set equivalence for update queries. Combining this axiomatization with our basic provenance definition, we obtain a set of equivalence axioms over expressions in $UP[X]$. We next exemplify the derivation of axioms in our structure based on [23]:

Example 3.3. Based on [23], the following transactions are equivalent

$$\begin{array}{ll} R^{M,p}(\mathbf{u_1},\mathbf{u_2})\text{:-} & \\ R^{-,p}(\mathbf{u_2})\text{:-} & \end{array} \quad \sim \quad \begin{array}{l} R^{-,p}(\mathbf{u_1})\text{:-} \\ R^{-,p}(\mathbf{u_2})\text{:-} \end{array}$$

Note that $\forall t_1 \vDash \mathbf{u_1}$ the provenance expression after the transaction on the left is $R(t_1) +_M p$ and after the transaction on the right, $R(t_1) -_D p$, thus $a +_D b = a -_M b$, i.e., $-_M$ and $+_D$ are equivalent, and therefore, from now on we use $-$ to denote both. Furthermore, $\forall t_2 \vDash \mathbf{u}_2$, from the left transaction we obtain the expression $\left(R(t_2) +_M \left(\left(\sum_{\substack{t_1 \vDash \mathbf{u_1} \\ t_1 \rightsquigarrow t_2}} R(t_1)\right) \cdot_M p\right)\right) +_D p$, and from the right transaction the expression $R(t_2) +_D p$. Since the sum in the former expression can contain a single element, we obtain that $\left(a +_M (b \cdot_M c)\right) - c = a - c$ for all a, b and c.

We simplified the axioms and removed redundancies to obtain the following set of equivalence axioms:

$$\left(a +_M (b \cdot_M c)\right) +_M (d \cdot_M c) = \left(a +_M (d \cdot_M c)\right) +_M (b \cdot_M c) \quad (1)$$

$$\left(a +_M (b \cdot_M c)\right) - c = a - c \quad (2)$$

For $\bigcap_{i \in I} J_i = \emptyset, \bigcup_{s \in S}\{e_s\} = \bigcup_{i \in I}\bigcup j \in J_i\{b_j\}$:

$$a +_M \left(\left(\sum_{i=1}^{n}(b_i +_M (\sum_{j \in S_i} c_j) \cdot_M d)\right) \cdot_M d\right) =$$

$$\left(a +_M ((\sum_{i \in I} c_i) \cdot_M d)\right) +_M \left((\sum_{i=1}^{n} b_i) \cdot_M d\right) \quad (3)$$

$$(a - b) - b = a - b \quad (4)$$

$$a +_M \left(\left(\sum_i (b_i - c)\right) \cdot_M c\right) = a \quad (5)$$

$$\left(a +_M (b \cdot_M c)\right) +_I c = (a +_I c) +_M (b \cdot_M c) \quad (6)$$

$$(a +_I b) - b = a - b \quad (7)$$

$$a +_M \left((b +_I c) \cdot_M c\right) = (a +_I c) +_M (b \cdot_M c) \quad (8)$$

$$\left(a +_M (b \cdot_M c)\right) +_I c = a +_I c \quad (9)$$

$$(a - b) +_I b = a +_I b \quad (10)$$

$$a +_M (\sum_i b + \sum_j d_j) \cdot_M c) =$$

$$(a +_M (\sum_i b_i \cdot_M c)) +_M (\sum_j d_j \cdot_M c) \quad (11)$$

$$(a-b) +_M (c \cdot_M b) = (a-b) +_M \left(\left((d-b) +_M (c \cdot_M b)\right) \cdot_M b\right) \quad (12)$$

Intuitively, axiom 1 states that if two different tuples were updated to the same tuple, the order of updates does not matter. Axiom 2 shows that deleting an updated tuple is similar to deleting the tuple without updating it. Axioms 4 and 7 show the same for a sequence of insertions or deletions of a tuple followed by a deletion. The fact that modifying tuples that satisfy $\mathbf{u_1}$ to tuples that satisfy $\mathbf{u_2}$ and then to tuples that satisfies $\mathbf{u_3}$, is equivalent to applying the modification of tuples that satisfy $\mathbf{u_1}$ to $\mathbf{u_3}$ and $\mathbf{u_2}$ to $\mathbf{u_3}$ is described by axiom 3. Axiom 5 states that updating a deleted tuple is the same as simply deleting it. From axioms 6 and 8 we see that insertion and modification can commute, and axioms 9 and 10 show that inserting an updated or deleted tuple is equivalent to simply inserting the tuple. Finally, axiom 11 states that modifying a set of tuples equivalent ot splitting it into disjoint sets and applying the modification on both, and axiom 12 shows that the order in which we apply the modification when modifying tuples in a cycle of length two does not affect the result of the computation.

A formula can be *rewritten* into another formula by applying a sequence of axioms. This rewriting is bidirectional, thus forming an equivalence relation: two formulas ϕ_1 and ϕ_2 of our update algebraic structure are equivalent if and only if there is a sequence axioms such ϕ_1 can rewritten into ϕ_2 by using the axioms. We denote it by $\phi_1 \equiv_{UP[X]} \phi_2$.

3.3 Preserving Provenance Under Set Equivalence

We next state the main property of our construction: two transactions yield the same provenance-aware result if and only if they are set-equivalent. We first define equivalence of transaction under our provenance-aware semantics:

Definition 3.4. We say that two UP[X]-relations R, R' are $UP[X]$-equivalent, and denote $R \equiv_{UP[X]} R'$, if for every tuple t we have that $R(t) \equiv_{UP[X]} R'(t)$[1]. We further say that two UP[X]-databases D, D' are $UP[X]$-equivalent (denote $D \equiv_{UP[X]} D'$) if there is a isomorphism between the relation names in D and D' so that matching relations are $UP[X]$-equivalent.

Finally, we say that two annotated transactions T_1^p and T_2^p are $UP[X]$-equivalent, and denote $T_1^p \equiv_{UP[X]} T_2^p$ if for every $UP[X]$-database D, we have that $T_1^p(D) \equiv_{UP[X]} T_2^p(D)$.

We further say that for non-annotated transactions T_1, T_2, they are set-equivalent, and denote by $T_1 \equiv_B T_2$, if for every database D, we have that $T_1(D) \equiv T_2(D)$, where "$\equiv$" now stands for standard isomorphism between the databases.

We are now ready to state the following result:

[1]Note that in particular, $UP[X]$ equivalence implies that the two relations include the same set of tuples.

Proposition 3.5. *For every two transactions T_1, T_2 we have that $T_1 \equiv_B T_2$ if and only if $T_1^p \equiv_{UP[X]} T_2^p$.*

We next outline the main ideas of the proof, omitting details for readability.

Proof. (sketch) Note that $UP[X]$-equivalence is defined to capture both set-equivalence and provenance equivalence, thus one direction of the proposition is trivial (by definition, $UP[X]$-equivalence implies set-equivalence).

In the other direction, assume there exists a sequence of axioms $A_1, \ldots A_n$ (axioms of [23]) such that T_2 can be obtain from T_1 by applying the axioms. We prove by induction on n that $T_1^p \equiv_{UP[X]} T_2^p$.

The base case is when T_2 is obtained from T_1 by applying a single axiom A. We can show that for every database D, $t \in T_1^p(D) \iff t \in T_2^p(D)$ since $T_1 \equiv_B T_2$ for each axiom A. For instance, if A is modification axiom 1 of [23] then T_1 contains the sequence $R^M(\mathbf{u_1}, \mathbf{u_2})$:-,$R^M(\mathbf{u_3}, \mathbf{u_4})$:- and T_2 is obtained by replacing this sequence with $R^M(\mathbf{u_3}, \mathbf{u_4})$:-,$R^M(\mathbf{u_1}, \mathbf{u_2})$:-. Thus T_1^p contains the sequence $R^{M,p}(\mathbf{u_1}, \mathbf{u_2})$:-,$R^{M,p}(\mathbf{u_3}, \mathbf{u_4})$:- and T_2^p is obtained by replacing this sequence with $R^{M,p}(\mathbf{u_3}, \mathbf{u_4})$:-,$R^{M,p}(\mathbf{u_1}, \mathbf{u_2})$:-. For every tuple $t \in T_1^p(D) \setminus \{t \mid t \vDash \mathbf{u_2} \vee t \vDash \mathbf{u_4}$, $T_1^p(D)(t) = T_2^p(D)(t)$. If $\mathbf{u_2} \neq \mathbf{u_4}$, then $T_1^p(D)(t) = T_2^p(D)(t)$ $\forall t \in \{t \mid t \vDash \mathbf{u_2} \vee t \vDash \mathbf{u_4}\}$. Otherwise, by using the provenance axiom 1 we obtain that $\forall t \vDash \mathbf{u_2}$ $T_1^p(D)(t_2) = T_2^p(D)(t_2)$.

A similar argument can be used for each of the axioms in [23]. Then, in the inductive step, we assume that the proposition holds when T_2 can be obtained from T_1 by a sequence for $n-1$ axioms, and show that it holds if T_2 can be obtain from T_1 by a sequence for n axioms $A_1, \ldots, A_n$ as follows. Let T_3 be the transaction obtain from T_1 bu applying the axioms $A_1, \ldots, A_{n-1}$. By the induction hypothesis $T_1^p \equiv_{Up[X]} T_3^p$. We then get that $T_3^p \equiv_{Up[X]} T_2^p$ and thus $T_1^p \equiv_{Up[X]} T_2^p$ □

Example 3.6. Reconsider the database fragment given in Figure 1a. According to [23], by modification axiom 2, the following transaction is set-equivalent to the transaction from Example 3.2.

$Products^{M,p}$(“Kids mnt bike", “Kids", c,
“Kids mnt bike", “Bicycles", c):-

$Products^{M,p}$(“Kids mnt bike", “Sport", c,
“Kids mnt bike", “Bicycles", c):-

The effect of both is that the tuples $Products^M$(“Kids mnt bike", “Kids", \$120) and $Products^M$(“Kids mnt bike", “Sport", \$120) are updated into a single tuple $Products^M$(“Kids mnt bike", “Bicycles", \$120). Indeed, the provenance expressions obtained by both are equivalent. The provenance of the tuple $Products$(“Kids mnt bike", “Kids", \$120) is $p_3 - p$, in both cases. The annotation of the tuple $Products$(“Kids mnt bike", “Sport", \$120) using the latter transaction is $p_1 - p$ and is equivalent to $(p_1 +_M (p_3 \cdot_M p)) - p$ by axiom 2. Last, the annotation of the tuple $Products$(Kids mnt bike, Bicycles, \$120) in the database obtained by this transaction is $(0 +_M (p_1 \cdot_M p)) +_M (p_3 \cdot_M p)$. By axiom 4 (and using $a = 0$) it is equivalent to the provenance of this tuple obtained in Example 3.2, and thus the databases resulting by the two transaction are equivalent.

Comparison with [6]. There exists a previously proposed algebraic provenance model for update queries, called MV-semirings [6]. This model is an extension of the semiring framework, in the sense that for every semiring K, the corresponding MV-semiring K^ν is introduced. The elements of such a semiring are symbolic expressions over elements from K, version annotations, and semiring operations where the structure of an expression encodes the derivation history of a tuple. For instance, $\mathbb{N}[X]^\nu$ is the MV-semiring corresponding to the provenance polynomials semiring $\mathbb{N}[X]$. Using the $\mathbb{N}[X]^\nu$ MV-semiring, each tuple is annotated by a provenance expression consisting of variables which represent identifiers of freshly inserted tuples, and version annotations that encode the sequence of updates that were applied to the tuple. The version annotation $X^{id}_{T,\nu}(k)$ denotes that operation X (X may be one of U, I, D, or C, which stand for update, insert, delete or commit respectively) was executed at time $\nu - 1$ by transaction T, where k is the annotation of the tuple before the update and id is the identifier of the operation.

The following example shows that unlike our construction, MV-semirings do not capture transaction equivalences, namely two equivalent transactions may yield different provenance-aware results [2]:

Example 3.7. Consider the equivalent transactions from Examples 3.2 and 3.6. Using the MV-semiring model, applying the two transactions to the database given in Figure 1a results in different provenance expressions. For instance, if the provenance annotations satisfy $p_3 = I^1_{T,2}(x_1)$, then the provenance of the tuple $Products$(Kids mnt bike, Bicycles, \$120) after applying the first transaction contains an expression of the form $U^1_{T,4}(U^1_{T,3}(I^1_{T,2}(x_1)))$ while the provenance annotation after applying the second transaction contains an expression of the form $U^1_{T,3}(I^1_{T,2}(x_1))$.

4 APPLICATIONS

We next demonstrate the usefulness of the introduced structure through a concrete semantics assigned to the operators.

[2] It should be noted that [6] does not claim invariance under equivalence for their model.

As we shall illustrate, the general axioms that we have derived above can guide the design of such semantics so that provenance is preserved through rewritings of transactions.

Each concrete semantics is represented by tuple $(\mathcal{K}, +_M^{\mathcal{K}}, \cdot_M^{\mathcal{K}}, -^{\mathcal{K}}, +_I^{\mathcal{K}}, +^{\mathcal{K}}, 0^{\mathcal{K}})$ where $\mathcal{K}$ is a set of provenance annotations, and $+_M^{\mathcal{K}}$, $\cdot_M^{\mathcal{K}}$, $-^{\mathcal{K}}$ and $+_I^{\mathcal{K}}$ are concrete operation over the values in $\mathcal{K}$. We call such tuple Update-Structure.

An important principle underlying the semiring-based provenance framework is that one can compute an "abstract" provenance representation and then "specialize" it in any domain. This "specialization" is formalized through the use of semiring homomorphism. To allow for a similar use of provenance in our setting, we extend the notion of homomorphism to Update-Structures.

Definition 4.1. Let $S_1 = (\mathcal{K}_1, +_M^{\mathcal{K}_1}, \cdot_M^{\mathcal{K}_1}, -^{\mathcal{K}_1}, +_I^{\mathcal{K}_1}, +^{\mathcal{K}_1}, 0^{\mathcal{K}_1})$ and $S_2 = (\mathcal{K}_2, +_M^{\mathcal{K}_2}, \cdot_M^{\mathcal{K}_2}, -^{\mathcal{K}_2}, +_I^{\mathcal{K}_2}, +^{\mathcal{K}_2}, 0^{\mathcal{K}_2})$ be two Update-Structures. A homomorphism is a mapping $h : S_1 \mapsto S_2$ such that

$$\begin{aligned}
h(a +_M^{\mathcal{K}_1} b) &= h(a) +_M^{\mathcal{K}_\epsilon} h(b) \\
h(a \cdot_M^{\mathcal{K}_1} b) &= h(a) \cdot_M^{\mathcal{K}_2} h(b) \\
h(a -^{\mathcal{K}_1} b) &= h(a) -^{\mathcal{K}_2} h(b) \\
h(a +_I^{\mathcal{K}_1} b) &= h(a) +_I^{\mathcal{K}_2} h(b) \\
h(a +^{\mathcal{K}_1} b) &= h(a) +^{\mathcal{K}_2} h(b) \\
h(0^{\mathcal{K}_1}) &= 0^{\mathcal{K}_2}
\end{aligned}$$

Crucially, we may show that provenance propagation commutes with homomorphisms.

Proposition 4.2. *Let S_1 and S_2 be two Update Structures such that there exists an homomorphism from S_1 to S_2. Let D be a database instance, T a transaction and t a tuple in $T(D)$. Let $\phi_1(t)$ (respectively $\phi_2(t)$) be the provenance expression of t by T over S_1 (respectively S_2). We have that $h(\phi_1(t)) = \phi_2(t)$.*

This property allows us to support applications as exemplified next.

4.1 Example Semantics

Consider an application that supports different products and prices for different countries (e.g., based on the the different shipping costs and taxes). Each tuple is annotated with a set of country names, such that a user from country c can see a tuple t only if t's annotation contains c. Similarly, update queries are also annotated by sets of countries, so that the query annotation defines the set of countries that are affected by the update. For instance, if a deletion query q deletes the tuple t and q's annotation contains the country c, then after the deletion the tuple t is no longer available for users from the country c.

Product	Price	
Kids mnt bike	\$120	t_1
Kids mnt bike	\$110	t_2

Figure 2: Deals Table

We can define the following semantics for the provenance operations:

$$\begin{aligned}
a +_M b &:= a \cup b \\
a \cdot_M b &:= a \cap b \\
a - b &:= a \setminus b \\
a +_I b &:= a \cup b \\
a + b &:= a \cup b
\end{aligned}$$

Where a and b are sets.

Proposition 4.3. *The above structure satisfies the axioms from Section 3.2.*

The following example presents a different semantics of the provenance operators. It serves to demonstrate that a-priori plausible interpretations of the operators that nevertheless violate the axioms, may consequently yield different provenance result for equivalent transactions.

When the data is uncertain we may annotate each tuple in the Database with trust value, e.g. a number $x \in [0, 1]$, where tuples that are annotated with 1 are always in the database and tuples annotated with 0 are not. We can then define the following semantics for the provenance operations:

$$\begin{aligned}
a +_M b &\equiv \max(a, b) \\
a \cdot_M b &\equiv a \cdot b \\
a - b &\equiv \begin{cases} 0 & \text{if } a \leq b \\ a & otherwise \end{cases} \\
a +_I b &\equiv \max(a, b) \\
a + b &\equiv \max(a, b)
\end{aligned}$$

The annotation in the case of (deletion/modification) query depends on the query annotation. Intuitively, when deleting or modifying a tuple, if the query annotation is greater than the provenance of the tuple before the update we apply the deletion/modification and otherwise we do not apply it.

Consider the Deals table shown in Figure 2 and the following transactions.

$Deals^{M,p}$("Kids mnt bike", \$120, "Kids mnt bike", \$110):-

$Deals^{M,p}$("Kids mnt bike", \$110, "Kids mnt bike", \$100):-

and

$Deals^{M,p}$("Kids mnt bike", \$120, "Kids mnt bike", \$100):-

$Deals^{M,p}$("Kids mnt bike", \$110, "Kids mnt bike", \$100):-

From the axioms of [23] (using modification axiom 2) the two transactions are equivalent. However, since the above semantics does not satisfy some of the axioms (e.g., axiom 1), the provenance of *Deals*("Kids mnt bike", \$110) may be different in some cases. For instance, consider the case where $p = 0.9$, $t_1 = 0.8$, and $t_2 = 0.5$. The annotation resulting from the first transaction is $0 +_M \Big(\big(t_2 +_M (t_1 \cdot_M p)\big) \cdot_M p\Big) = \big(t_2 +_M (t_1 \cdot_M p)\big) \cdot_M p$ and the annotation resulting from the second transition is $\big(0 +_M (t_1 \cdot_M p)\big) +_M (t_2 \cdot_M p) = (t_1 \cdot_M p) +_M (t_2 \cdot_M p)$
Using the above semantics and valuation we obtain for the first transaction
$\big(t_2 +_M (t_1 \cdot_M p)\big) \cdot_M p = \max(0.5, (0.8 \cdot 0.9)) \cdot_M 0.9 = 0.648$
But for the second transaction
$(t_1 \cdot_M p) +_M (t_2 \cdot_M p) = \max((0.8 \cdot 0.9), (0.5 \cdot 0.9)) = 0.72$

4.2 From Semiring to $UP[X]$-operators

As discussed above, it is commonplace to define algebraic provenance through semirings. We next show how to transform a commutative semiring of interest – given that it satisfies some natural constraints – into an $UP[X]$ structure that can be used for provenance in the presence of update queries.

Theorem 4.4. *Let $(K, +_K, \cdot_K, 0, 1)$ be a commutative semiring that satisfies $a +_K 1 = 1$ and $a \cdot_K a = a$, then the set of elements $X = K$, with the operators $+_M, +_I, \cdot_M$ defined as follows: $\forall a, b \in X$:*

$$\begin{aligned} a +_M b &= a +_K b \\ a +_I b &= a +_K b \\ a \cdot_M b &= a \cdot_K b \end{aligned}$$

and any $-$ operator that satisfies the axioms 2, 4, 5, 7, 10 and 12 from Section 3.2 with respect to the semiring $+$ and $\cdot$ operators, is an $UP[X]$ structure.

Proof. We show that all the axioms are satisfied by the defined operators. We use $+$ to denote $+_M$ and $+_I$ and $\cdot$ to denote $\cdot_M$.

- Axioms 1 and 6 satisfied for every commutative semiring due to commutativity and associativity of $+$.
- Axiom 3:

$$\big(a + ((\sum_{i\in I} c_i) \cdot d)\big) + \big((\sum_{i=1}^{n} b_i) \cdot d\big) \overset{\cap S_i=\emptyset, \cup S_i=I}{=}$$
$$(a + (\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot d) + \big((\sum_{i=1}^{n} b_i) \cdot d\big) \overset{\text{associativity of } +}{=}$$
$$a + \Big((\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot d + \big((\sum_{i=1}^{n} b_i) \cdot d\big)\Big) \overset{d\cdot d=d}{=}$$
$$a + \Big((\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot (d \cdot d) + \big((\sum_{i=1}^{n} b_i) \cdot d\big)\Big) \overset{d\cdot d=d}{=}$$
$$a + \Big((\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot (d \cdot d) + \big((\sum_{i=1}^{n} b_i) \cdot d\big)\Big) \overset{\text{associativity and distributivity of } \cdot}{=}$$
$$a + \Big(\big((\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot d\big) + (\sum_{i=1}^{n} b_i)\Big) \cdot d\Big) \overset{\text{commotativity of } +}{=}$$
$$a + \Big(\big((\sum_{i=1}^{n} b_i) + \big((\sum_{i=1}^{n}\sum_{j\in S_i} c_j) \cdot d\big)\Big) \cdot d\Big) \overset{\text{distributivity of } \cdot}{=}$$
$$a + \Big(\big((\sum_{i=1}^{n} b_i) + \big(\sum_{i=1}^{n}((\sum_{j\in S_i} c_j) \cdot d)\big) \cdot d\big)\Big) \overset{\text{associativity of } +}{=}$$
$$a + \Big(\big(\sum_{i=1}^{n} \big(b_i + (\sum_{j\in S_i} c_j) \cdot d\big)\Big) \cdot d\Big)$$

- Axiom 8:

$$a + \big((b + c) \cdot c\big) \overset{\text{distributivity of } \cdot}{=} a + \big((b \cdot c) + (c \cdot c)\big) \overset{c\cdot c=c}{=}$$
$$a + \big((b \cdot c) + c\big) \overset{\text{associativity and commotativity of } +}{=} (a + c) + (b \cdot c)$$

- Axiom 9:

$$\big(a + (b \cdot c)\big) + c \overset{\text{associativity of } +}{=} a + \big((b \cdot c) + c\big) \overset{\text{distributivity of } \cdot}{=}$$
$$a + \big((b + 1) \cdot c\big) \overset{b+1=1}{=} a + \big(1 \cdot c\big) \overset{c\cdot 1=c}{=} a + c$$

- Axiom 11 is satisfied for every commutative semiring due to distributivity of $\cdot$ and associativity of $+$.
- Axioms 2, 4, 5, 7, 10 and 12 are satisfied by definition.

□

Example 4.5. Recall the example from Section 4.1. The corresponding semiring is $(\mathcal{P}(C), \cup, \cap, \emptyset, C)$ where C is the set of all countries and $\mathcal{P}(C)$ is the power set of C. Note that this is a commutative semiring that satisfies $\forall a \in \mathcal{P}(C)\ a \cup C = C$ and $a \cap a = a$. Furthermore, by defining the $-$ operator as set-difference we obtain a structure that satisfies the axioms (following Proposition 4.3).

As another example, consider the PosBool semiring $(\mathbb{N}[\mathbb{B}], \vee, \wedge, \bot, \top)$. By defining $a - b = a \wedge (\neg b)$ we obtain a structure that satisfies the axioms.

5 EFFICIENT PROVENANCE COMPUTATION

We have proposed a provenance structure UP[X], showed that it is faithful to the semantics of update queries in the sense that it is preserved under query equivalence, and demonstrated useful instances of it.

A next natural question is that of complexity: how large may the provenance be? Can it be efficiently computed during query evaluation?

5.1 Naive Construction

A first attempt is to generate provenance by directly using the definitions. That is, starting from the initial instance, we apply sequentially the update queries. We compute the provenance of each tuple after each update using the definitions of Section 3. Unfortunately, the following result shows that this approach incurs an exponential blowup in the transaction length.

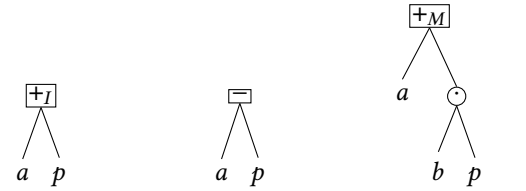
Proposition 5.1. *There exists a transaction T and a database D with only two tuples t_1 and t_2 such that the provenance of t_1 and the provenance of t_2 after applying T to D is at least exponential in the number of queries.*

Fortunately, we introduce a normal form for our provenance expression which is linear in the database size and the transaction length. Moreover, we prove that this normal form is computable in polynomial time in the size of the database and the transaction.

5.2 Normal Form

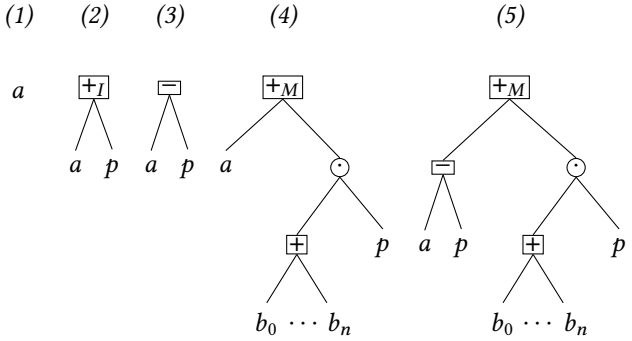
For presentation purposes, we represent our provenance expressions as trees in a classical manner. Figure 3 depicts the basic tree representation for each one of the provenance operations. Any provenance expression obtained by the construction for the class of "domain-based" transactions, when applied to an X-database (i.e. a database whose tuple annotations are just identifiers), can be represented as a composition of the basic trees.

Theorem 5.2. *Given a transaction T^p, an X-database D, and $t \in T^p(D)$ with the provenance expression ϕ. Then, there exists an equivalent provenance expression $\phi' \sim \phi$ such that the tree representation of ϕ' has one of the following forms:*



(a) $a +_I p$ **(b)** $a - p$ **(c)** $a +_M (b \cdot_M p)$

Figure 3: Tree representation of provenance expression



Computing $\phi'(t)$ may be performed in polynomial time in the size of in D and T. Moreover, $\phi'(t)$ can be computed incrementally for each update of the transaction.

To prove the previous theorem, we introduce a new set of rules that are used to compute this normal form; the rules are described in Figure 4. In Rule 1 and Rule 2, a is the annotation associated to the tuple on which the update is applied. Intuitively, applying an insertion or a deletion overrides the previous updates. Rules 3 and 8 intuitively state that an update based on an deleted tuple has no effect and Rule 4 states that an update based on an inserted tuple is equivalent to inserting the current tuple. Rules 5, 6 and 7 intuitively allow to "factorize" successive updates into a single update.

Proof. (Sketch) Let D be a X-database. Let T^p be an annotated transaction. We prove the theorem by induction on the transaction structure.

The three kinds of updates are handled as follows:

Insertion By Rule 1, we simply replace the provenance expression by one of size 3.

Deletion By Rule 2, we again simply replace the provenance expression by one of size 3.

Modification Let $t_{i_1}, \ldots, t_{i_m}$ be the tuples modified to be the tuple t. For each such tuple t_{i_j}, its provenance is equal to $\phi(t_{i_j}) - p$. By Rule 2, each minimized provenance expression $\phi'(t_{i_j})$ is equal to a tree of the form of Tree (b) in Figure 3. Then, note that $\phi'(t)$ is equal to $\phi(t) +_M \sum(\phi(t_{i_j})) \cdot_M p$. By induction, trees for the provenance expressions of $\phi(t)$ and $\phi(t_{i_j})$ are of the

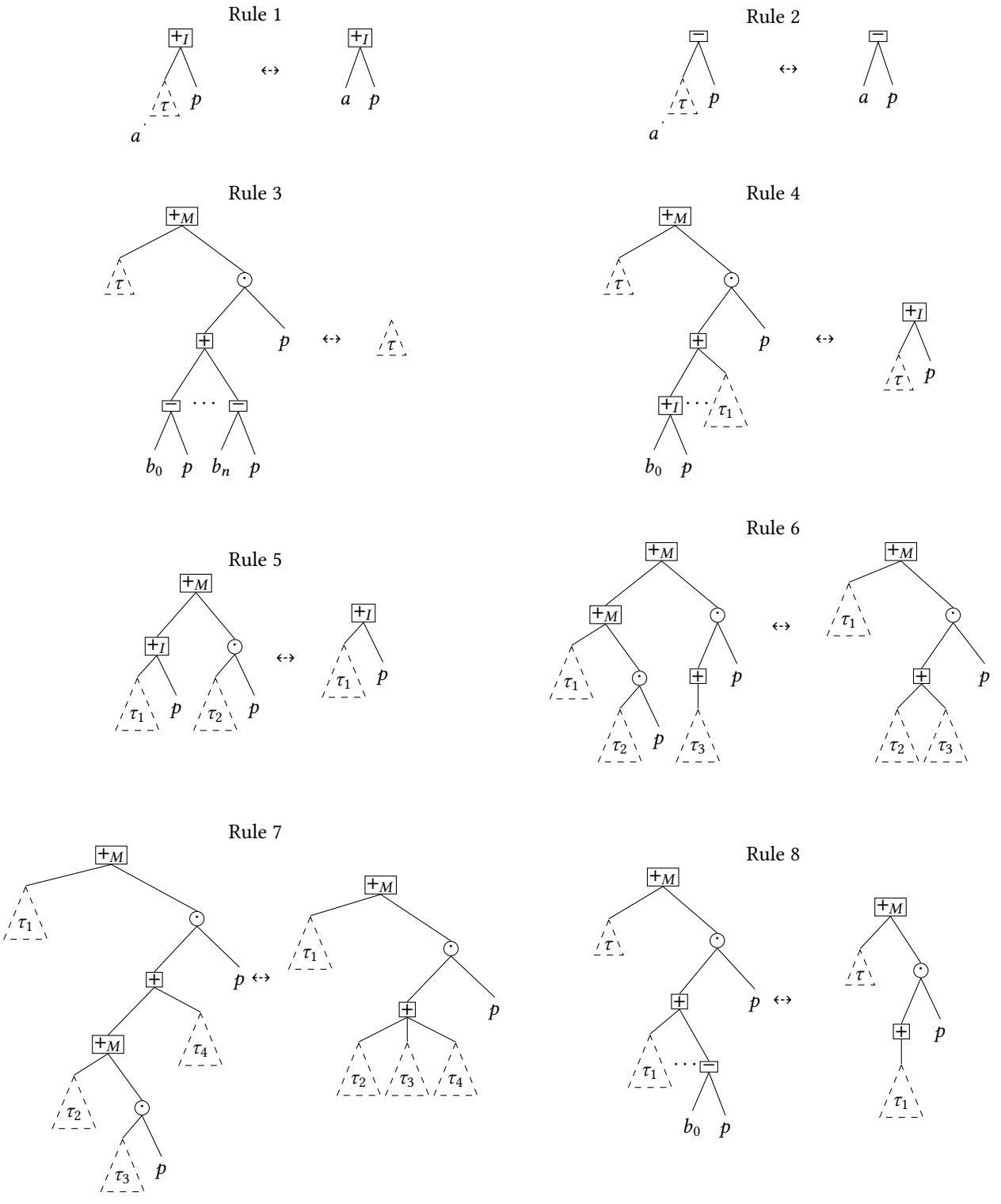


Figure 4: Rules for computing the normal form

form presented in Figure 3. First, we distinguish different cases following the root of $\phi(t)$:

- The root of $\phi(t)$ is the $+_I$ operator: by using Rule 5, the provenance expression $\phi'(t) = \phi(t)$.
- The root of $\phi(t)$ is the $-$ operator: the subtree representing $\phi(t)$ is not changed.
- The root of $\phi(t)$ is the $+_M$ operator: by using Rule 6 and the form of $\phi(t)$, we can rewrite $\phi'(t)$ such that the new lefthand tree under the root labeled $+_M$ is of the form of Tree (a) or (b) of Figure 3.

We next explain how to modify the righthand subtree τ under a root labeled $+_M$ in the representation of $\phi'(t)$. This subtree represents the provenance expression $(\sum_i \phi(t_{i_j})) \cdot_M p$. We consider different cases:

- If there exists a tuple t_{i_j} such that the root of $\phi(t_{i_j})$ is the $+_I$ operator, then, by Rule 4, the provenance expression $\phi'(t)$ is equal to $\phi(t) +_I p$ which is of the form of Tree (a) of Figure 3.
- If each tuple t_{i_j} such that the root of $\phi(t_{i_j})$ is the $-$ operator, then, by Rule 3, $\phi'(t)$ is equal to $\phi(t)$.
- If there exists at least one tuple t_{i_j} such that $\phi(t_{i_j})$ is equal to $a - p$, then, by Rule 8, it is possible to remove each subtree representing $\phi(t_{i_j})$ if its roots is labeled by $-$.
- If there exists a t_{i_j} such that $\phi(t_{i_j})$ is equal to $a +_M ((\sum_s b_s) \cdot_M p)$, then, by Rule 7, it is possible to obtain the desire form of the tree.

We observe that after each step, we compute only a linear size formula and that the number of operations performed on this formula is polynomial in the database and the size of the (prefix of the) transaction. □

This normal form is in general not guaranteed to be minimal; this is due to the possible existence of 0 in the formula, and not removed by our rules. This can be fixed by applying the axioms related to 0 to the normal form; we show that the result is a minimal expression, and that further it is unique.

Proposition 5.3. *Let T^p be an annotated transaction, applied to an X-database D. Let $\phi(t)$ be the normal form provenance expression of a tuple t after applying the transaction T^p to D. Let $\phi'(t)$ be the expression obtained by using the axioms related to 0 to $\phi(t)$ to minimize it. Then $\phi'(t)$ is unique and a minimized formula.*

Proof. (sketch) We observe that by applying the "0 axioms" to a normal form formula, we may obtain either (1) a normal form expression, or (2) 0 or (3) a formula of the form $\Sigma_i(b_i) \cdot_M p$. We can show that none of these expressions is equivalent to any other, and there is no further concise way of representing neither of them. □

6 EXPERIMENTAL EVALUATION

We have conducted experiments whose main goals were examining (1) the scalability of the approach in terms of time and memory overhead, (2) the usefulness of the resulting provenance for applications such as deletion propagation, and (3) the effectiveness of our provenance normal form representation which in turn is based on our provenance equivalence axiomatization.

We used Python 3 to implement our provenance framework for an in-memory database. The experiments were executed on Windows 10, 64-bit, with 8GB of RAM and Intel Core i7-4600U 2.10 GHz processor.

6.1 Benchmarking

We have developed a dedicated benchmark that involves both synthetic and real data as follows.

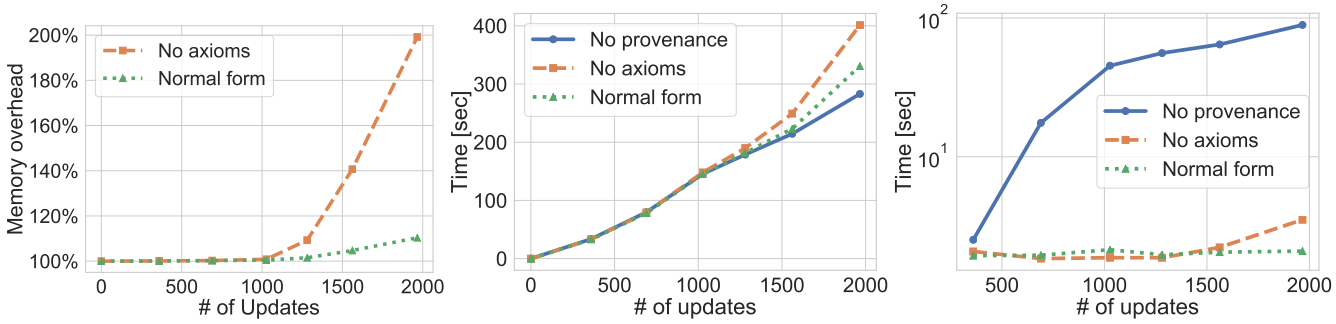
TPC-C benchmark. The TPC Benchmark C (TPC-C)[1] is an on-line transaction processing benchmark. The benchmark includes two read-only and three update-intensive transactions, that simulate the activity of complex on-line transaction processing application environments. The database consists of nine tables and is populated with initial data of about 2.1M tuples. For more details and the implementation rules of the benchmark see [1]. For our experiments, we used the Python open source implementation of the benchmark from [2] to generate transactions log, and executed the log using our in-memory database with provenance support implementation.

Synthetic dataset. We used a database with a single relation with two columns, string and numeric. The table was populated with 1M tuples, with randomly generated values from a fixed domain using a uniform distribution. We generated sequences of update queries of varying length. The type of query (insertion, deletion or update) was randomly selected with uniform distribution; the parameters of the queries (e.g., which tuples are modified and how) were selected at random from a fixed domain, where deletion and modification queries perform a selection over the numeric column.

Compared solutions. We examine the performance of provenance tracking without using the axioms (orange line), of provenance tracking using the normal form representation (green line) and of executing transactions with no provenance tracking (blue line).

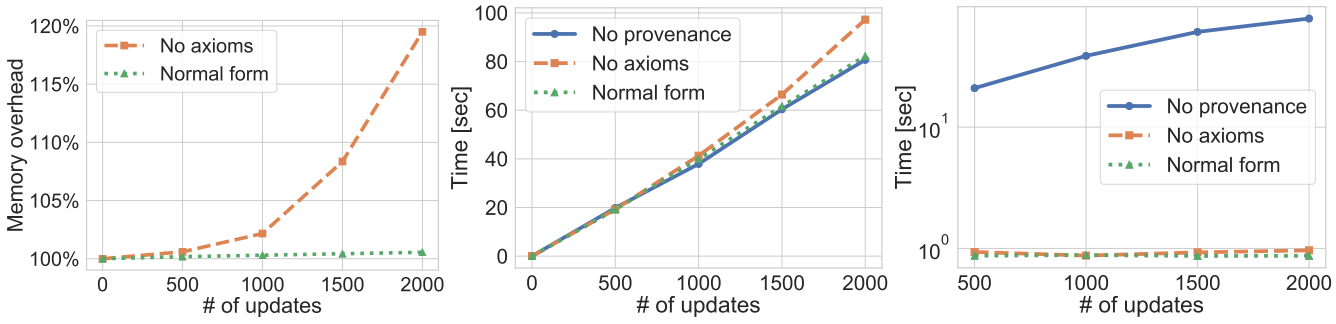
6.2 Overhead and Usage

The first set of experiments aims at assessing the provenance generation time and memory overhead, as well as the usefulness of the approach. To this end we used both the TPC-C



(a) Memory overhead **(b) Runtime** **(c) Usage time (log scale)**

Figure 5: Provenance overhead and usage (TPC-C dataset)



(a) Memory overhead **(b) Runtime** **(c) Usage time (log scale)**

Figure 6: Provenance overhead and usage (synthetic dataset)

benchmark as well as the synthetic dataset. We augmented the database tuples with provenance annotations, varied the number of updates up to 2000, and observed the effect on the runtime, memory consumption, and usage for hypothetical reasoning. In the sequel we report the results obtained using transactions that affect 200 tuples (0.02% of the the database tuples) in the synthetic dataset, which is consistent with observed percentage in TPC-C. We observed similar trends for other numbers of affected tuples.

Memory overhead. Provenance tracking leads to two types of space overhead. First, recall that deleted and modified tuples are in fact not removed from the database in our construction (intuitively so that the operation may be "undone"). Therefore, the database size continuously grow. Second, maintaining the provenance expressions incurs an overhead. We measure the provenance size by the number of nodes in its tree representation. Figures 5a and 6a show the memory overhead compared to executing the transactions with no provenance tracking, as a function of the number of updates, for the TPC-C and synthetic datasets respectively.

The initial database of the TPC-C dataset consist of 2,095,703 tuples. We note that the choice of provenance representation does not affect the number of tuples in the database: provenance tracking with or without the normal form representation leads to the same number of tuples. The largest database obtained after 2000 updates contained 2,144,887 tuples with provenance tracking, an overhead of 2% compared to the initial database size. In contrast to the case with the database size, there is a significant difference in the provenance size: for the largest number of updates, the provenance size using the naive approach (i.e. no application of axioms) was 4,127,127, while using the normal form representation the size of the provenance was only 2,264,798, a difference of over 82%.

Using the synthetic dataset with 1M tuples, the database resulting from the computation with provenance tracking contained 1,001,260 tuples. We observed an overhead of about 100% (i.e., $\times 2$) using the normal form representation, where the provenance size was only $1,004,160$ while the provenance size without applying the axioms was 1,193,640 an overhead of 120% with respect to no provenance tracking.

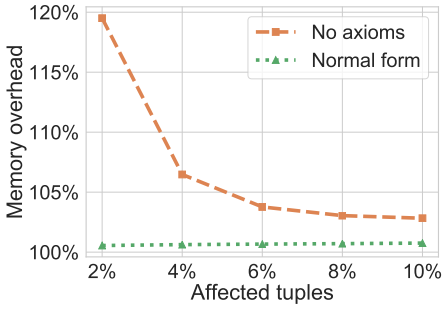
Running time. Figure 5b depicts the running time of the transaction for the TPC-C dataset. Although provenance tracking and maintenance incur overhead in both the database size and additional memory for the provenance information, the overhead is reasonable: the running time without

provenance tracking was 283 seconds for the largest number of updates, and 401 and 330 seconds for the provenance tracking without using the axioms and with the normal form representation respectively. Interestingly, even though using the normal form requires the application of rules for minimization (see Section 5), the running time of the construction with normal form representation is lower than that of the naive approach. This is because the minimization is done incrementally after each update, and as a result the maintained provenance size is significantly smaller than the provenance expression obtained without using the axioms. Note that generating new provenance expression for new or updated tuples uses the existing tuples provenance and requires copying it. Thus large provenance expressions lead also to overhead in the tracking time, which underlines another useful aspect of the normal form.

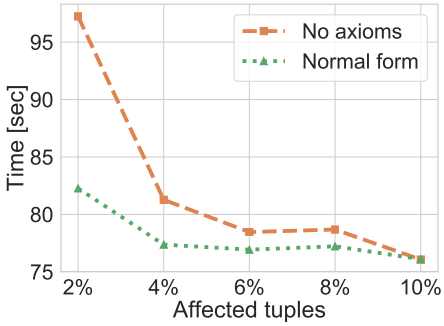
We observed similar trends with the synthetic dataset as shown in Figure 6b. The computation time of the transaction with no provenance tracking was about 77 seconds; provenance tracking without using the axioms incurred an overhead of over 25% (about 97 seconds), whereas using the normal form representation, the running time overhead was less than 3% (only 79 seconds).

Provenance Usage. We examine the usefulness of the framework to the application of hypothetical reasoning. In particular, we considered deletion propagation by assigning values to database tuples. We measured the assignment time to the provenance generated by the construction without using the axioms and the normal form representation. We further compared these measurements to a simple baseline approach of directly deleting tuples from the database and re-running the transaction on the resulting modified database. The results are reported in Figures 5c and 6c.

For the two datasets and for both variants of provenance tracking, using the provenance framework significantly outperforms the baseline approach. For the largest number of updates in the TPC-C dataset (Figure 5c), re-running the transaction over the modified database took 89 seconds, while the provenance assignment time was 3.43 seconds (over ×25 faster that the baseline) for the naive construction and 1.94 seconds using the normal form representation (over ×45 faster that the baseline). The gain of using the normal form representation compared with the naive construction was significant: about 78%. For the synthetic dataset (Figure 6c), the re-computation time was 78 seconds, the assignment time for the provenance generated without using the axioms was 0.96 seconds, and for the normal form it was 0.86 seconds. These are over ×81 and ×91 faster than the baseline, respectively.



(a) Memory overhead



(b) Runtime

Figure 7: Naive representation Vs. normal form as a function of number affected tuples

6.3 When do we gain from the Normal Form?

The last set of experiments aims at assessing the usefulness of the normal form representation in synthetic environment where we change the provenance size. As the number of update per tuple increases, the difference between the sizes of the provenance generated without using the axioms and of the provenance represented in the normal form, increases. For a fixed number of updates, as the number of affected tuples increases, the number of update per tuple decreases (since the updated tuples are selected with uniform distribution). Thus, we fixed the transaction length and examined the effect of the number of tuples affected by the transaction on the overhead incurred by provenance tracking with and without the normal form. We varied the number of affected tuples from 200 to 1000 (0.1% of the database size). This is in line with the number of affected tuples in the TPC-C dataset that varies from from 200 to 2000 (0.1% of the database size there). The results for 1M tuples and 2000 update queries are shown in Figure 7.

Figure 7a depicts the memory overhead of provenance tracking as a function of the number of tuples affected by the transaction. Recall that the axioms allow us to compactly represent the provenance expression of a single tuple at a

time. Thus, for large number of updates per tuple, we expect to see a significant difference between the two approaches. Indeed, for small numbers of affected tuples, the provenance size of each tuple is larger. Then, the effect of the axioms on the provenance size is notable, reflecting on the memory overhead. We note that there is a moderate growth in the memory overhead when using the axioms as the number of affected tuples increases.

The running time as a function of the number of tuples that are affected, is presented in Figure 7b. As a result of the changes in the provenance size when the number of updates per tuple increases, the overhead of maintaining the provenance without using the axioms increases. We also observed a moderate growth for the construction that uses the axioms when the number of update per tuple increases. This growth is due to the (relatively small) overhead of minimizing the provenance after each update.

7 RELATED WORK

Data provenance has been studied for different data transformation languages, from relational algebra to Nested Relational Calculus, with different provenance models [7, 10, 11, 15, 16, 19, 25, 34]) and applications [18, 26, 27, 30, 32].

A particular line of research studying the tracking of *how-provenance* (also termed semiring provenance) aims at recording the computational process induced by a given query while accounting for *relevant meta-data*. It has been shown in [21] that the algebraic structure of *semirings* adequately captures provenance for the positive relational algebra: the two semiring operations, addition and multiplication, correspond to the two kinds of data uses found in the context of positive relational algebra, namely those of *alternative* and *joint* use of data; and the equivalence axioms of the semiring structure (associativity of each operation, commutativity of multiplication over addition, etc.) correspond precisely to equivalence axioms of positive relational algebra. Consequent algebraic constructions that follow this pattern have since been proposed for various formalisms including aggregate queries [5], queries with difference [4], Datalog [21] and SPARQL queries [17]. The construction of the model we propose in this paper is inspired by the work of [21], for the highly expressive formalism of update transactions.

A provenance model for SPARQL queries and updates using a provenance graph was presented in [22], and [10] proposes an approach to provenance tracking for data that is copied among databases using a sequence of insert, delete, copy, and paste actions. Provenance for updates was also studied in [9, 33] and in [8], where a boolean provenance model for updates was proposed; however, none of these approaches has proposed an algebraic provenance model. Updates are a form of non-monotone reasoning, and as such are related to the notion of relational difference. Algebraic provenance models for queries with difference have been proposed in [4, 16, 17], but naturally none could be directly applied to update queries; in particular using the “monus" operation of [16, 17] as our minus operation may not result in a structure satisfying the equivalence axioms. Further exploration of the connections between these models and ours is left for future work. Closest to our work is the multi-version semiring model (MV-semiring) [6] discussed above, a provenance model for queries and updates that extends the semiring annotation framework of [21]. This model is designed to support transactions executed using the snapshot isolation concurrency control protocol. While the MV-semiring allows for provenance tracking for rich update queries, beyond the class of “domain-based" transactions that we present here, it is not preserved under transaction equivalences as shown in Section 3. An extension of the semiring model of [21] to account for updates was also studied in [24], however the focus there is on the use of provenance in the context of trust and while the work includes an efficient implementation, it falls short of proposing a generic algebraic construction.

8 CONCLUSION AND LIMITATIONS

We have developed a novel algebraic provenance model for hyperplane update queries and sequences thereof, following the axiomatization of query equivalence in [23]. We have shown that the model captures the “essence of computation" for such queries, i.e. equivalent transactions yield equivalent provenance. We have shown means of instantiating the model, towards applications of provenance in this context. The example instances show the usefulness of the generic model: by following the axioms, we are guaranteed that our provenance construction is independent of transaction rewrites. We have further studied the efficient computation and storage of provenance, and have shown a minimization technique that leads to compact provenance representation via a normal form. This again leverages the axioms, this time in a computational manner. Our experimental evaluation shows the tractability and usefulness of the approach, as well as the benefit of using the normal form.

A main limitation of our solution is that it is confined to hyperplane queries. One may address this challenge towards supporting update queries with conjunctive conditions and beyond; yet such attempt would likely cost in the loss of the property of provenance being preserved under equivalence, since (to our knowledge) no sound and complete axiomatization is known for these more expressive fragments. Further exploration of such endeavours is left for future work.

REFERENCES

[1] Tpc benchmark. http://www.tpc.org/tpcc/.

[2] Tpc implementation. https://github.com/apavlo/py-tpcc.

[3] S. Abiteboul and V. Vianu. Equivalence and optimization of relational transactions. *J. ACM*, 35(1), 1988.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. On the limitations of provenance for queries with difference. In *TaPP*, 2011.

[5] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *Proc. of PODS*, 2011.

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

[7] O. Benjelloun, A. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17, 2008.

[8] P. Bourhis, D. Deutch, and Y. Moskovitch. Analyzing data-centric applications: Why, what-if, and how-to. In *ICDE*, 2016.

[9] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[10] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4), 2008.

[11] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[12] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

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

[14] D. Deutch, Y. Moskovitch, and V. Tannen. Provenance-based analysis of data-centric processes. *VLDB J.*, 24(4), 2015.

[15] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5), 2012.

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

[17] F. Geerts, T. Unger, G. Karvounarakis, I. Fundulaki, and V. Christophides. Algebraic structures for capturing the provenance of SPARQL queries. *J. ACM*, 63(1), 2016.

[18] B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. TRAMP: understanding the behavior of schema mappings through provenance. *PVLDB*, 3(1):1314–1325, 2010.

[19] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. Provenance for data mining. In *Tapp*, 2013.

[20] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007.

[21] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[22] H. Halpin and J. Cheney. Dynamic provenance for sparql updates. In *ISWC*, 2014.

[23] D. Karabeg and V. Vianu. Simplification rules and complete axiomatization for relational update transactions. *ACM Trans. Database Syst.*, 16(3), 1991.

[24] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen. Collaborative data sharing via update exchange and provenance. *ACM Trans. Database Syst.*, 38(3), 2013.

[25] B. Kenig, A. Gal, and O. Strichman. A new class of lineage expressions over probabilistic databases computable in p-time. In *SUM*, pages 219–232, 2013.

[26] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12), 2011.

[27] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, 2012.

[28] D. Montesi and R. Torlone. A rewriting technique for the analysis and the optimization of active databases. In *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, pages 238–251, 1995.

[29] D. Montesi and R. Torlone. Analysis and optimization of active databases. *Data Knowl. Eng.*, 40(3):241–271, 2002.

[30] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, 2014.

[31] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, 5(2), 2008.

[32] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[33] S. Vansummeren and J. Cheney. Recording provenance for SQL queries and updates. *IEEE Data Eng. Bull.*, 30(4), 2007.

[34] Prov-overview, w3c working group note. http://www.w3.org/TR/prov-overview/, 2013.