Analyzing Data-Centric Applications: Why, What-if, and How-to

Pierre Bourhis
CNRS, UMR 9189 - CRISTAL

Daniel Deutch
Tel Aviv University

Yuval Moskovitch
Tel Aviv University

Abstract—We consider in this paper the analysis of complex applications that query and update an underlying database in their operation. We focus on three classes of analytical questions that are important for application owners and users alike: Why was a result generated? What would be the result if the application logic or database is modified in a particular way? How can one interact with the application to achieve a particular goal? Answering these questions efficiently is a fundamental step towards optimizing the application and its use. Noting that provenance was a key component in answering similar questions in the context of database queries, we develop a provenance-based model and efficient algorithms for these problems in the context of data-centric applications. Novel challenges here include the dynamic update of data, combined with the possibly complex workflows allowed by applications. We nevertheless achieve theoretical guarantees for the algorithms performance, and experimentally show their efficiency and usefulness, even in presence of complex applications and large-scale data.

I. INTRODUCTION

Many real-life applications rely on, and dynamically update, an underlying database in the course of their execution. For example, E-commerce applications rely on a database for management of products and orders which affects the possible execution of the application and its interaction with potential users.

The complexity of such applications leads to many challenges, faced by both the application owners and users. The owner needs to analyze the application and logs of its executions, so that she can identify bugs and misuses and ultimately optimize the application. The users need to obtain a better understanding of the application and optimal ways she can use it for her particular goals. The following are some core questions that owners, analysts and users would like to ask:

- Why? When a surprising result is observed in an execution, such as an incorrect product price presented to the user at checkout, a first fundamental question is why was it obtained as such. One possible answer to this question involves showing the full execution log, but for complex applications such logs would be too complicated to view. Instead, it would be much more beneficial to view a concise explanation of the result, involving only the relevant aspects of the execution.

- What-if? The application owners may further attempt a fix to the application logic or database. Before applying the actual fix, they may wish to consider multiple hypothetical such fixes, and observe their anticipated effect on the execution and its artifacts. Making the actual changes to the process logic/data/etc. and re-running the application may be inefficient or even impossible, and thus a more effective solution is required.

- How-to? The application users would typically be interested in optimizing their use of the application (rather than its logic or database, both of which they cannot change). In our example, if there are multiple ways to choose a product (e.g. through a “standard” or through “discount deals” page), they would like to be shown a sequence of navigation actions that would lead to purchase at the discounted price.

In the context of database queries, a prominent approach for answering such questions is based on the tracking of provenance, i.e. a record of the transformations that data undergoes. The idea is to efficiently track the “core” aspects of the transformations that have taken place, and then use it for answering questions such as the above (see discussion of related work in Section IX). In particular, the use of data provenance for explanations (“Why”) has been studied in e.g. [1], [2], its use for hypothetical reasoning (“What-if”) has been studied in e.g. [3], and its use for how-to analysis (“reverse data management”) has been studied in [4], [5]. We consequently study in this paper the tracking and use of provenance, in the context of possibly complex applications that query and update an underlying database. The presence of control flow combined with data updates, absent from the above works, lead to novel challenges both in provenance generation as well as in the precise formulation of the above questions (why, what-if, how-to), and the design of algorithms for their solutions. We next outline the paper structure, detailing our contributions.

Process Model (Section II): We start by presenting a simple, yet rich, model for processes whose executions will be the subject of provenance tracking introduced in subsequent sections. We extend the model of data-dependent processes (DDPs) presented in [6], which models applications whose control flow is guided by a finite state machine, as well as the state of a static underlying database. A novelty w.r.t. [6] is that transitions are further associated with actions, which are update (insertion/deletion/modification) queries over the database. We also allow user input which may be accessed by both guard and action queries.

Provenance Model (Section III): We then introduce a (boolean) provenance model for executions of such processes. We design the model first for update queries, then for combinations of guard and action queries, and finally for executions (sequences of such combinations). The applications
in the following sections will also serve as indication for the soundness of the model.

Efficient Provenance Generation (Section IV): To be of practical use, provenance should be generated and stored efficiently. A first reasonable choice for this purpose is to use boolean formulas, but we show that this may entail an exponential size overhead with respect to the length of the execution for which provenance is tracked. Instead, we provide an efficient construction based on boolean circuits, where the sharing of sub-formulas allows for an efficient construction.

What-if (Section V): We then begin our study of provenance use by considering the “what-if” question. We capture what-if analysis through the notion of commutation with truth valuations, that may be applied to annotations of state machine transitions, database tuples, or both. Each valuation stands for a scenario of deleting some tuples or transitions along with their queries (by assigning them the truth false). The effect of modification or insertion scenarios that have been prepared in advance can also be analyzed, by introducing appropriate queries as a part of the application logic that is initially disabled, and then “enabling” them only for hypothetical reasoning purposes by mapping the corresponding annotation to true. We show that commutation holds for our provenance construction, thus allowing a practically efficient way of examining the effect of hypothetical scenarios on execution results.

Why (Section VI): While the provenance of an execution already serves as an explanation for its result, full provenance size would typically be too large to allow obtaining insights by simply viewing it. Moreover, while the entire provenance is needed for applications such as what-if, not all parts of it are equally interesting for presentation purposes. As a simple example, if a tuple is inserted, deleted, and inserted again, then a short yet sufficient explanation for its existence is its last insertion. We formalize such a notion of minimal sufficient explanations, which accounts for arbitrarily complex process logics (In particular, the existence of guards further complicates the question of what is a good explanation). We furthermore provide an efficient algorithm for computing a minimal sufficient explanation, starting from a provenance circuit representing an execution that has taken place.

How-to (Section VII): How-to questions are formalized as the retrieval of an execution that yields a result instance or sub-instance of interest. The problem is undecidable in general and is NP-complete for processes whose underlying state machine is acyclic, and whose relations arity is bounded. Still, under those restrictions we are able to provide a practically efficient solution, even in presence of infinitely many possible user inputs (and consequently infinitely many possible executions). The solution is based on a different construction of a provenance circuit, compactly representing equivalence classes of the possible executions, namely executions that may be grouped together with “parameters” replacing the different possible user inputs. The circuit is constructed in a way that guarantees that satisfying assignments to its variables correspond to executions yielding tuples of interest (and executions yielding an instance of interest are then satisfying assignments to boolean combinations of the corresponding sub-circuits). We can consequently use a SAT solver to find these executions in a practically efficient way (even though the worst case complexity remains EXPTIME).

Experimental study (Section VIII): We have implemented our provenance constructions as well as the algorithms that use it for the various tasks outlined above. We have conducted an extensive experimental study of the efficiency and usefulness of our solutions, based on a mix of synthetic and real data. We note that there are two different provenance constructions in play here. For “what-if” and “why”, one needs to efficiently track provenance alongside an actual single execution, and then efficiently use it for the analysis. Indeed, our results indicate a reasonable overhead of provenance tracking (with respect to a provenance-oblivious execution) as well as good performance of the algorithms that use it. For “how-to” analysis, generation of provenance for all possible executions is done statically, and may be considered as a “pre-processing” step. Our results indicate that this step may indeed be computationally costly for complex application logic, but serves its purpose in allowing an efficient response time to how-to queries.

We survey previously proposed process models and provenance solutions in Section IX and conclude in Section X.

II. Process Model

We next introduce a simple yet rich process model, including the ability to (1) query and modify an underlying database and (2) be guided by input from external sources.

A. Update Queries

We focus first on conjunctive queries, augmented to capture updates.

Definition 2.1: An insertion query has the form

\[ R^+ (u) : \neg R_1(u_1) \ldots \neg R_n(u_n) \]

where \( R_i \)'s are relation names, and \( u, u_1, \ldots u_n \) are sets of variables with appropriate arities. Every variable occurring in \( u \) must occur in at least one of \( u_1, \ldots u_n \). Similarly, a deletion query has the form

\[ R^- (u) : \neg R_1(u_1) \ldots \neg R_n(u_n) \]

Last, a modification query has the form

\[ R^M (u) : \neg R_1(u_1) \ldots \neg R_n(u_n) \]

where \( u = (u'_1, u'_2) \) is of arity \( 2 \cdot r \) where \( r \) is the arity of \( R \).

We generally refer to such queries (insertion, deletion, or modification) as update queries.

The semantics of queries is then realized through the standard notion of instantiation. Every instantiation of the variables of \( u_1, \ldots u_n \) grounded in the database state before evaluation of the query has started gives rise to an instantiation of \( u \). Then, for each such instantiation, the following effect is inflicted by each type of query:

- (Insertion) The corresponding tuple is added to the relation \( R \) residing in the head of the rule.
- (Deletion) The corresponding tuple is deleted from the relation \( R \) residing in the head of the rule.
such that $v$ products with the same name) will be added to $newprice$
notation naturally extends to sequences of update queries. By
match of $(prod, newprice)$ in $Deals$
boolean combination $\neg Q \lor \neg D$
Products

Consider Table I, describing the database of an E-commerce application. It includes a binary relation storing available products and their prices and a unary relation storing products that have been selected so far as a particular user (“shopping cart”) in her navigation through the application. It further includes an auxiliary unary relation $R_d$ storing a set of products currently marked for selection. The following query adds all selected products to the shopping cart:

\[ \text{ShoppingCart}^+ (\text{prod}) := \text{Products}(\text{prod}, \text{price}), R_d(\text{prod}) \]

Now assume also a $Deals$ relation including details of discount deals; the following query changes the prices of all products in the $Products$ relation to their discounted prices. Each match to $(\text{prod}, \text{oldprice})$ in $Products$ is modified to the match of $(\text{prod}, \text{newprice})$ in $Deals$:

\[ \text{Products}^M (\text{prod}, \text{oldprice}, \text{prod}, \text{newprice}) := \neg \text{Products}(\text{prod}, \text{oldprice}), \text{Deals}(\text{prod}, \text{newprice}) \]

Note that in principle the same product may yield multiple matches for $\text{newprice}$, in which case multiple tuples sharing the same product name (supposedly standing for different products with the same name) will be added to $\text{Products}$.

Given an update query $Q$ and a database $D$, we denote by $Q(D)$ the database resulting from applying $Q$ to $D$. The notation naturally extends to sequences of update queries.

**Boolean queries** We will also be referring in the sequel to Boolean expressions, captured by a Boolean combination of CQs (or union thereof) with no head variables. Such CQ is said to be satisfied if there exists a valid instantiation of its body. The notion of satisfaction is extended to boolean combinations (using $\land$, $\lor$, $\neg$) of such queries in the standard way.

**B. Dynamic Data-Dependent Processes**

We next define a model of data dependent processes. We use the notion of DDP presented in [6], and extend it in order to capture updates and user input. The model is that of state machines whose transitions are governed by guards that may query an underlying database, as well as actions that may modify the database through update queries. To facilitate user input, we distinguish a set of singleton relations which we collectively refer to as “the user database”. Then, both guards and actions of such transitions may query (but not update) user relations.

**Definition 2.3:** A dynamic Data-Dependent Process (dDDP) is a tuple $(V, E, v_0, v_F, D_{\text{internal}}, D_{\text{user}}, Q_g, Q_a)$ such that $(V, E)$ is a directed graph referred to as the dDDP state machine, with a distinguished “initial” node $v_0 \in V$ and a distinguished set of “final” nodes $V_F \subseteq V$. $D_{\text{internal}}$ and $D_{\text{user}}$ are two database schemas (for internal application data and user input, respectively). For each edge $e \in E$ there is a dedicated singleton relation $R_e \in D_{\text{user}}$, intuitively storing the possible user input, so that $D_{\text{user}} = \bigcup R_e$. Each edge $e \in E$ is further associated with an event which is a pair of two queries: a guard and an action. The guard $Q_g(e)$ is a boolean query over $D_{\text{internal}} \cup R_e$; the action $Q_a(e)$ is an update query over $D_{\text{internal}}$ whose updates are performed only to $D_{\text{internal}}$.

**Example 2.4:** Consider the process logic of an e-commerce application shown in Figure 1 (ignore for now the $p_i$ annotations above the queries). A user may navigate to view all products or those with discount deals, and choose products to add to the shopping cart. Before payment, she can remove products from the shopping cart and finally she can confirm the order. If the order is not empty she can pay and exit. $D_{\text{internal}}$ consists of four relations. The relation $Products$ includes all the products in the system, the relation $Deals$ includes products that are offered with special discount prices, the relation $ShoppingCart$ includes the products selected by the user during the execution, and the relation $Order$ includes the list of products before payment. $D_{\text{user}}$ includes the relations $R_d$, $R_p$ and $R_e$, corresponding to user input relevant for the transitions in which they appear. For simplicity, when a guard (action) is true (no-op) we do not specify it.

We next define executions and their output. We first define the effect of a single event $Ev$ (a guard $Q_g$ and an action $Q_a$) on a database $D$, denoted $Ev(D)$ to be the result of $Q_a$ applied to $D$ if $Q_g(D)$ is true, and to $D$ otherwise. The definition of executions follows.

**Definition 2.5 (Executions):** An execution $Ex$ of a dDDP $(V, E, v_0, v_F, F_g, F_a, D_{\text{internal}}, D_{\text{user}})$ with respect to an initial instance $D_{\text{internal}}$ of $D_{\text{internal}}$, is a sequence of pairs $[[e_1, D_{\text{user}}], ..., (e_n, D_{\text{user}})]$ where each $e_i$ is an edge and each $D_{\text{user}}$ is an instance of $D_{\text{user}}$ containing exactly one tuple, satisfying:

- The sequence of edges $e_1, ..., e_n$ is a finite path in $(V, E)$ from the initial state $v_0$ to a final state in $v_F$
- For $i = 1, ..., n$, let $Ev_i$ be the event associated with $e_i$. We then define $D_{\text{internal}}$ in the following recursive manner: $D_{\text{internal}}^0$ is given as input, and $D_{\text{internal}}$ is obtained as $Ev_i(D_{\text{internal}}^0)$.

We say that $D_{\text{internal}}$ is the execution result and also denote $D_{\text{internal}}^n = Ex(D_{\text{internal}}^0)$. 

![Dynamic Data-Dependent Process](image-url)
Example 2.6: Consider the Deals and Products relations of Table I (ignore for now the Prov column) serving as an initial database instance. Further consider the path [HomePage, Deals, Deals, Products, OrderConfirm, Payment]; the relevant part of $D_\text{user}$ is the instance of $R_d$ queried in the second step of the execution (i.e. in $D_\text{internal}$). The action in this step adds a tuple $\text{ShoppingCart}(\text{iPhone6})$ to $D_\text{internal}$. A further update adds $\text{Order}(\text{iPhone6}, 320\$)$ to $D_\text{internal}$, and finally, since the Order table is not empty, the guard $Q_9$ of the transition from OrderConfirm to Payment is satisfied by the database. Note that in this execution, the user has selected the product iPhone6 while viewing a discounted price of 299$ but at the end of the run, the Order table contains a tuple with the price 320$. Intuitively, this is due to the fact that the price update was done with respect to the Products table rather than the Deals table. We later show how provenance can be used to find such errors.

Last, it will further be useful to consider sub-sequences of an execution. A sub-sequence consists of a subset of the execution events, in an order that is consistent with their order in the execution.

III. Provenance Model

We introduce in this section a boolean provenance construction for update queries, that will serve as the basis for our applications. We start by a simple modification to the syntax of update queries, so that they include an annotation as part of their specifcation. Intuitively this annotation may stand for an identifier of the query, or any other meta-data associated with it. We will show different kinds of annotations tailored for different applications below. For now, we simply fix a set $P$ of symbols to be used as annotations and augment the heads of queries so that they also specify an annotation. For instance, the head of a provenance-aware insertion query has the form $R^+ p(u)$, where $p \in P$ is the annotation; similarly for deletion and modification, as well as for guards. For instance, the query $\text{ShoppingCart}^+ p(x) :- \text{Deals}(x,y), R_d(x)$ (see again Figure 1) is an annotated insertion query with the annotation being $p$.

We further introduce the notion of tuple-annotated relations. Given a set of annotations $X$, let $B[X]$ be the domain of boolean expressions over $X$. An $B[X]$-relation over some schema $U$ is then a mapping from the domain of tuples conforming to $U$ to $B[X]$, such that a finite number of tuples are mapped to an element which is not $\text{false}$. Intuitively each tuple is mapped to its annotation (which is a boolean expression), so that $R(t)$ is the annotation of the tuple $t$ in the relation $R$. Finally, a semantics of a query is a mapping of $B[X]$-relations to $B[X]$-relations. In our case, we will use an infinite set $X$ of annotations, which is the union of two disjoint sets $P$ and $T$. Intuitively $P$ (query annotations) will be used to track which queries have contributed to the generation of an output tuple, and $T$ (tuple annotations) will be used to track which input tuples have contributed in this respect. Having $X$ (the set of “potential” annotations) being of infinite size is a mere technicality, that will allow us not to worry about having to use new provenance tokens for query results.

Provenance for update queries: We use $Q$ to denote an update query, $H^\alpha(y)$ (where $\alpha$ is $+$, $-$ or $\emptyset$ where the query is insertion, deletion or modification respectively) to denote the head, and $B(x)$ to denote the body (the variables in $y$ must occur in $x$). Given a database instance $D$ and an assignment $A$ to the query body, we use $b_1(A), \ldots, b_n(A)$ to denote the tuples in $D$ corresponding to the instantiation of $B(x)$ according to $A$. The provenance expressions of these tuples in $D$ are denoted $p_1(A), \ldots, p_n(A)$ respectively. For every tuple $t$ obtained as the result of some assignment $A$ (denote this by $t = A(y)$), we further let $B(D)(t)$ be $\bigvee_{\{A|t=A(y)\}} \bigwedge_{i=1}^{n} p_i(A)$. Intuitively $B(D)(t)$ is the condition corresponding to $t$ being derived.

We now define the provenance of the output of $Q$ with respect to an instance $D$. In what follows $H'(t)$ is the provenance annotation of a tuple $t$ in the relation $H$, after the query was executed.

Definition 3.1: (Boolean Provenance for update queries)

- If $Q = H^+ p(y) :- B(x)$ is an insertion query, then we define for every result tuple $t$,
  \[ H'(t) = H(t) \lor (p \land B(D)(t)) \]

- If $Q = H^- p(y) :- B(x)$ is a deletion query, then we define for every result tuple $t$,
  \[ H'(t) = H(t) \land \neg (p \land B(D)(t)) \]

- If $Q=H^M p(y_1, y_2) :- B(x)$, then let $A$ be an assignment to the variables of $Q$ such that $A(y_1) = t_1$ and such that $A(y_2) = t_2$. We define $(t_1 \cdot t_2)$ for the concatenation of $t_1$ and $t_2$:
  \[ H'(t_1) = H(t_1) \land \neg (p \land B(D)(t_1 \cdot t_2)) \]
  and
  \[ H'(t_2) = H(t_2) \lor (p \land B(D)(t_1 \cdot t_2)) \]

Intuitively, the condition for existence of an inserted tuple is that either it existed before existence (captured by $H(t)$), or the insertion has taken place ($p$) and $t$ was yielded by an instantiation ($B(D)(t)$). For deletion, instead of actually performing the deletion we keep $t$ but annotate it with a condition that its deletion has not taken place (an immediate “undoing” the deletion). In a similar spirit, for modification we keep both old and new version of the tuple with appropriate conditions (in practice of course we will avoid duplicate tuples and only keep track of the values that have changed).

We note that provenance for update queries has been studied in previous work (see e.g. [7], [8], [9]), albeit with different models that make it unsuitable for our intended applications in the context of workflows. See Section IX.
Provenance for events: We next extend the provenance definition to account for events (recall that an event includes a boolean guard and an update query). To this end, we associate provenance tokens from \( P \) with both the guard and the update query of the event, now referred to as annotated event and denoted \( Ev^p = (Q^p, Q_u^p) \) (for simplicity we will assume the same annotation is used for the guard and the update queries).

The result then involves further conditioning on satisfaction of the annotated guard \( Q^p \) with respect to an instance \( D \). The expression capturing this condition is denoted by \( Q^p(D) \); if \( Q^p \) is a CQ then its result is defined by \( Q^p(D) = \bigwedge_A \bigvee p_i(A) \) where \( A \) is an assignment of the variables of \( Q^p \) witnessing the satisfaction of \( Q^p \) in \( D \). Boolean combinations of CQs lead to the same boolean combination applied to their provenance formulas (but \( p \) appears only once).

Finally, the definition of provenance for annotated events is obtained by simply applying the semantics of Definition 3.1 to \( Q^p \) and the instance \( D \), but replacing every occurrence of \( p \) with \( Q^p(D) \). The result for an annotated event \( Ev^p \) and a database \( \hat{D} \) of \( B[X] \)-relations is denoted by \( Ev^p(\hat{D}) \).

Provenance for dDDP executions: An important property of our construction is that it is compositional, in the sense that both the input and output of an update may be captured by \( B[X] \)-relations. Thus extending the definition to sequences of events, and in particular to executions, is straightforward. We say that a provenance-aware dDDP is a dDDP whose events are annotated and whose relations are in fact \( B[X] \)-relations. Then the provenance-aware result of an execution \( Ex \) w.r.t. an initial database \( D \), denoted \( Ex(D) \), is immediately well-defined by recalling the definition of execution results (Def. 2.5), and plugging-in the above definition of provenance-aware results of (sequences of) annotated events (note that the latter is a distinction with e.g. the model in [6]).

Example 3.2: Reconsider our running example, but now note the annotations next to tuples in Table I. Further reconsider the execution in Example 2.6; its output is now a database of annotated relations, as follows. The effect of the first query \( ShoppingCart^+(\epsilon) \):- Deals(x, y), \( Ri_d(x) \) is the insertion of the tuple \( ShoppingCart(iPhone6) \) to the database, with provenance annotation \( p_1 \wedge (a \wedge c) \). The provenance reflects that the only satisfying assignment is based on the tuple \( Deals(iPhone6, 299) \), with annotation \( a \) and the tuple \( Rd(iPhone6) \) with annotation \( c \). The \( ShoppingCart(iPhone6) \) tuple was not in the database before the execution, thus is available if the query annotation \( p_1 \) is true and both \( a \) and \( c \) are true. The next update query in the execution is \( Order^+p_4(x, y) \):- ShoppingCart(x), \( Products(x, y) \) and as a result, the tuple \( Order(iPhone6, 320) \) is added to the database with the provenance annotation \( p_4 \wedge \{(p_3 \wedge (c \wedge a)) \wedge b\} \). As explained in Example 2.6, since the price update was done via the \( Products \) table (and not \( Deals \) table) using the query annotated with \( p_4 \), although the user selected the product \( iPhone6 \) with the price \( 299 \) (from \( Deals \) table, using the query annotated with \( p_1 \)). Observe that when a tuple annotation includes both \( p_4 \) and \( p_1 \), the product price in the \( Order \) relation is different from the price viewed by the user.

IV. EFFICIENT PROVENANCE GENERATION

So far, we have defined an abstract model of provenance and its usage through valuations. To be of practical use, the provenance should further be generated and stored efficiently. When considering individual update queries, the defined notion of provenance via boolean formulas already entails an efficient algorithm for computing it. The following is thus easy to show:

Proposition 4.1: Given an update query \( Q \) and a database \( D \) of \( B[X] \)-relations, we can compute \( Q(D) \) in polynomial time in the size of \( D \) (with the exponent depending on the size of \( Q \)).

When considering provenance for sequences of updates, we need to be more careful to avoid an exponential blow-up with respect to the number of updates, as illustrated by the following example.

Example 4.2: Let \( \{R_0, ..., R_n, U_0, ..., U_n\} \) be a set of relation names, with the \( R_i \)'s being binary and the \( U_i \)'s being unary. Let \( D \) be an instance such that the \( U_i \)'s are empty with the exception of \( U_0 \) that includes the tuples \( \alpha_0^1 \) and \( \alpha_0^2 \). Each \( R_i \) includes the tuples \( \{(\alpha_1, \alpha_2, \alpha_3, \alpha_4)\}; \{(\alpha_2, \alpha_1, \alpha_3, \alpha_4)\}; \{(\alpha_4, \alpha_1, \alpha_3, \alpha_2)\} \) and \( \{(\alpha_4, \alpha_2, \alpha_1, \alpha_3)\} \). Then, consider the sequence \( Ex \) of insertion queries, of length \( n \), where the \( i \)'th query \( Q_i \) is defined as \( U_i^+(y) :- U_{i-1}(x), R_{i-1}(x, y) \). We can show that each fact \( U_n(\alpha_0^1) \) or \( U_n(\alpha_0^2) \) in \( Ex(D) \) has a provenance formula of size \( O(2^n) \).

To achieve tractability (for a fixed arity of relations), we resort to boolean circuits rather than formulas. A boolean circuit is a labeled Directed Acyclic Graph whose “leaves” (nodes with out-degree 0) are labeled by boolean variables, and internal nodes are labeled by either \( \vee \), \( \wedge \), or \( \neg \). We consider here circuits with multiple “roots” (nodes with in-degree 0), to facilitate the representation of multiple formulas in a compact way using the same circuit. Each such formula may be “read” by starting at one of the roots and recursively unfolding the sub-circuit rooted at it into a formula. Finally, such a circuit can be used to represent an annotated database by simply having the tuples pointing at different roots.

The following result then allows for efficient provenance tracking when the arity of relations is fixed.

Proposition 4.3: For any annotated dDDP execution \( Ex \) and a \( B[X] \)-database \( D \), such that the arity of \( D \) and of queries occurring in \( Ex \) is fixed, computing a circuit representing \( Ex(D) \) may be done in polynomial time in the length of \( Ex \) and the size of \( D \).

Proof: (sketch) By bounding the arity we guarantee that the number of distinct tuples is polynomial (with the exponent possibly depending on the arity). As for provenance, for each individual update query, the boolean formula capturing its provenance is already of polynomial size and may be transformed to a polynomial-size circuit. Care is needed to avoid an exponential overhead when applying multiple update queries (see Example 4.2). To this end, we apply the following recursive algorithm. Assume that after applying the \( i \)'th update query, the annotation of every tuple in the result is represented by a circuit. Then we treat each annotation as a distinguished variable, and apply the \( i+1 \)'th update query, obtaining for each output tuple a polynomial size circuit whose nodes correspond
to these variables. Importantly, in the generation of these circuits we employ sharing of variables and basic annotations: each of these appear only once, and whenever their use is required for the provenance of some tuple, a pointer to it is used rather then generating the new variables. Finally, instead of each variable we “plug-in” pointers to the sub-circuit that it represents.

The resulting circuit is of polynomial size and linear depth in the worst case.

We next consider three important applications that are based on provenance.

V. WHAT-IF

As explained in [3], application analysts are, in many cases, interested in observing the effect of some change to the database or application logic, on the result of its executions. This may help them in identifying possible bugs, in optimizing the application towards better customer service or increased revenues, etc. A simple but ineffective solution is to apply the actual change and re-run the application. This may in practice be prohibitively inefficient. Instead, we would like to be able to apply the hypothetical scenario directly to the provenance, “reading” the hypothetical execution result from it. For relational queries this was formalized in [3], [10] through the notion of commutation with truth valuation, which we now transfer to our setting:

Definition 5.1: Given a set of annotations $X$, a valuation over $X$ is a mapping $v : X \rightarrow \{true, false\}$. Such a valuation lifts to mapping $B[X]$-relations to “standard” relations, as follows. First, $v$ lifts to $v : B[X] \rightarrow \{true, false\}$ in the standard way; then, we define $v(R)$ as the relation $R'$ defined by $\{ t \in R \mid v(R(t)) = true \}$. We use $v(D)$ to denote the result of the application of the valuation $v$ to all relations of a database $D$.

In the same way, we define the notion of valuation for executions. For a valuation $v$ and an execution $Ex$, we denote by $v(Ex)$ the sub-sequence of events in $Ex$ whose annotations are mapped by $v$ to true. The definition also extends naturally to partial valuations, that assign truth values to only a subset of the variables. In that case the result of applying the lifted valuation to a $B[X]$-relation is still a $B[X]$-relation.

Example 5.2: Consider the valuation $v : \{a, b, c, p_1, p_2\} \rightarrow \{true, false\}$, where $v(b) = false$ and $v(u) = true$ for every other annotation $u$. Intuitively it corresponds to a scenario where the tuple $\text{Products}(\text{iPhone6}, 320\text{S})$ is omitted. By applying $v$ to the output annotated relation in Example 3.2, we obtain a relation with the single tuple $\text{ShoppingCart}(\text{iPhone6})$; note that the annotation of the tuple $\text{Order}(\text{iPhone6}, 320\text{S})$ was mapped to false, and thus this tuple is omitted. In this case the guard on the edge from $\text{OrderConfirm}$ to $\text{Payment}$ is not satisfied, and the execution ends at $\text{OrderConfirm}$.

Hypothetical scenarios as valuations: The above example shows the use of valuations to capture a “scenario” where a tuple is deleted. Through valuations to annotations of events, scenarios of insertion or modification may be captured as well (this is non-obvious to achieve e.g. in standard relational setting [3]): one may change the specification structure to include “hypothetical” update queries with particular annotations. Then after provenance is tracked along side with execution, one can read the “real” results by choosing a valuation that assigns false to these annotations, and simulate the hypothetical scenario by assigning true to the annotations.

Example 5.3: Reconsider the process logic shown in Figure 1, and assume that the analyst wishes to examine an alternative logic where the price of a shopping cart product chosen in the “Products” page is modified to its deal price if a deal exists, just before payment. Then the corresponding modification query is added to the transition from “OrderConfirm” to “Payment”, with a new annotation (say $p_7$). Note that the “real” (hypothetical) results are now obtained when $p_7$ is mapped to false (resp. true) and the other annotations are mapped to true.

We now define a fundamental property for the effective use of provenance for hypothetical scenarios: the construction commutes with valuations.

Definition 5.4: Let $X = T \cup P$ be a set of provenance annotations (where $T$ and $P$ are disjoint). We say that a provenance semantics satisfies commutation with valuations if for every sequence of annotated events $Ex$, every annotated database $D$ and every pair of (possibly partial) valuations $V_T, V_P$ over $T, P$ resp.: $V_T(Ex(D)) = V_P(Ex(T)(D))$

The rightmost side of the equation is the execution result, under the assumption that an hypothetical scenario corresponding to the valuation is applied to the underlying database and to the specification. The leftmost side gives us a much more efficient way of computing this result for multiple scenarios: compute the provenance representation once, and evaluate multiple scenarios with respect to this representation (evaluating a scenario corresponds to applying valuation to the boolean circuit, which is typically much faster than query evaluation). The additional expressions in the equation indicate that valuations may be applied separately with respect to either the database or the specification.

Proposition 5.5: The provenance semantics that we have defined satisfies commutation with valuations.

VI. WHY

We next consider the use of provenance for explanations. In a sense, the provenance of an execution already serves by itself as an explanation for the database obtained in the end of the execution (as illustrated in Example 3.2). But despite the polynomial time algorithm we have shown for generating full provenance (and see also our experimental study indicating the feasibility of full provenance generation and storage), it would typically be too large to allow obtaining insights by simply viewing it (unlike the case in e.g. workflow provenance where data is abstracted away, see e.g. [11], [12]). Furthermore, not all parts of the execution (and thus of its provenance) are really of interest for explanation purposes. We next demonstrate this through simple examples.
Example 6.1: Consider a simple execution involving the insertion, deletion, and then again the insertion of a tuple \( R(a) \) (in this order), with guards being trivially satisfied. Here the first insertion is not a good explanation for the tuple existence: the “real” reason for having \( R(a) \) in the output is the second insertion. This may get more complicated when guards are involved. For instance, consider a fourth step of the execution, with a guard asking for the existence of \( R(a) \) and a consequent action inserting \( R(b) \). Now the distinction of which events have contributed to the generation of \( R(a) \) needs to be taken into account in computing explanations for \( R(b) \) as well; this propagates to events querying \( R(b) \), etc.

To formalize the notion of a minimal relevant explanation, we introduce two novel notions: necessity of an action for a subsequence and a fact; and stability of an event for a subsequence.

Definition 6.2: Let \( Ex \) be an execution and let \( D \) be a database instance. Let \( t \) be a tuple appearing in \( Ex(D) \). Let \( Ex' \) be a subsequence of \( Ex \).

We say that \( Ex' \) is sufficient for \( t \) w.r.t. \( D \) if \( t \) appears in \( Ex'(D) \). We say that \( Ex' \) is stable for \( t \) with respect to \( D \) if for each subsequence \( Ex'' \) containing \( Ex' \) and contained in \( Ex \), it holds that \( Ex'' \) is sufficient for \( t \) w.r.t. \( D \).

An event \( Ev \) is necessary for \( Ex', t, D \) if the sequence \( Ex' - \{ Ev \} \) is not a stable subsequence for \( t \). A sufficient subsequence \( Ex' \) is minimal if all events of \( Ex' \) are necessary for \( Ex', t, D \).

Finally, \( Ex' \) is a minimal explanation with respect to \( Ex, t \) and \( D \) if \( Ex' \) is a stable and minimal subsequence of \( Ex, t \) and \( D \).

The number of minimal explanations may in the worst case be exponential in the execution length:

Proposition 6.3: Let \( n \) be an integer. There exist an instance \( D \), a sequence of events \( Ex \) (both of polynomial size in \( n \)) and a tuple \( t \) in \( Ex(D) \) such that the number of minimal explanations for \( Ex, t \) is exponential in \( n \).

Still, we next provide an efficient algorithm for finding (one) minimal explanation. For that, we use as input the provenance (represented as a circuit, see Proposition 4.3) of an execution that has taken place, restricted to an output tuple of interest. We will also require an additional information, in the form of a function \( comp \) that given two execution events indicates which one has preceded the other (this is not encoded in the provenance circuit but can easily be maintained).

Theorem 6.4: Let \( D \) be an input database, and let \( Ex \) be an execution with respect to \( D \). Further let \( t \) be a tuple in \( Ex(D) \). Given as input (1) a provenance circuit \( P \) with respect to \( Ex, D, t \), and (2) a function \( comp \) comparing two events of \( Ex \), finding a minimal explanation for \( Ex, D, t \) can be done in \( PTIME \) in the size of \( P \) (with linear dependency on the length of \( Ex \) if greater than the size of \( P \)).

Recall that the size of the provenance circuit is itself polynomial in the length of the execution that was tracked.

Algorithm 1 is a \( PTIME \) algorithm for computing such a minimal explanation. The general idea of the algorithm is to traverse the events from end to start, looking at the impact of forgetting each individual event – i.e. whether omitting it would result in \( t \) still being generated. If so then the update has no impact considering the previous updates and should be discarded from the explanation; otherwise we keep it. Intuitively, the notion of necessary event is closed under sufficient subsequences and thus the necessity of the last event can be determined by looking only at it.

In more detail, we extract the annotations of the execution events occurring in \( P \) and we order them, based on \( comp \) (line 1-3). We then initialize (line 4-7) a vector \( V \) of size \( n \) where \( n \) is the number of events in \( P \), of pairs event, boolean value, initially all the events in \( P \) at true. We traverse it from end to start (lines 8-13), and use the provenance circuit \( P \) to efficiently check the effect of omitting an event from the execution given the decision of keeping/omitting each of the events that follow. This is done by calling an Eval function (line 10) that applies to the provenance circuit \( P \) the valuation corresponding to \( V \). Finally (line 14), we return the vector \( V \), and present as a minimal explanation only the events whose are paired with true in \( V \).

Algorithm 1: Finding a minimal explanation

VII. How-to

We next aim at explaining how to obtain an instance of interest (i.e. via what execution).

Definition 7.1: Let \( S \) be a dDDP, let \( D \) be an initial database and let \( T \) be a finite set of tuples. We say that \( T \) is a possible result with respect to \( S \) and \( D \) iff there exists an execution \( Ex \) of \( S \) such that \( T \subseteq Ex(D) \). The problem of deciding if a result is possible is called \( POSS \).

In standard relational settings where a counterpart of the how-to problem has been studied (see e.g. [4], [5]), it is usually decidable. Here, allowing general workflows leads to undecidability:

Proposition 7.2: \( POSS \) is undecidable even if the sub-instance \( T \) of interest includes a single tuple.

We consequently restrict the attention to an important subclass of acyclic dDDPs, namely those whose underlying state machine is acyclic. For simplicity, we will also assume a
single final state. Last, as an auxiliary step, we further restrict attention to grounded acyclic dDDPs, which means there is no user input. This restriction will be relaxed in the sequel.

**Definition 7.3:** Let $S$ be a dDDP. We say that $S$ is grounded if every atom of the form in $R_{\text{user}}(\tilde{x})$ occurring in an action query of $S$ is grounded.

For dDDPs which are grounded and acyclic, the problem becomes decidable, but is NP-complete (note that such dDDPs still admit exponentially many executions).

**Proposition 7.4:** POSS is NP-complete when restricted to grounded acyclic dDDPs whose relations are of fixed arity, and even if the sub-instance $T$ of interest includes a single tuple.

Even in presence of this hardness result, we may provide an efficient solution based on extending our notion of provenance, so that it captures a set of possible executions (rather than capturing a single execution, as done so far) along with their output. To formalize this, we start by defining the effect of a valuation $v$ on a grounded-aware dDDP $S$ (rather than on a single execution as done above). The result $v(S)$ is defined as a dDDP $S'$ obtained by removing from $S$ all transitions whose annotations were mapped by $v$ to false, and removing unreachable states. If the nodes and edges of $v(S)$ form an execution with respect to some initial database, we identify $v(S)$ with the execution it induces. We then show (recall that $X$ is the set of annotations; we assume that every annotation is used at most once so that it uniquely identifies events and tuples).

**Proposition 7.5:** Let $S$ be a grounded acyclic provenance-aware dDDP with fixed arities, let $D$ be an initial database and $T$ be a finite set of tuples. We can compute in PTIME (in the size of $S, D$) a circuit $C$ over the elements of $X$, such that for every valuation $V : X \mapsto \{\text{true, false}\}$, we have that $V(C) = \text{true}$ if $V(S)$ forms an execution in $S$ with respect to $D$, and $T$ is a sub-instance of the execution result.

**Proof:** (sketch) We give an algorithm for computing $C$. We denote by $D_s$ the annotated database of all tuples that appear in the result of some execution ending at a node $s$. We will use the annotations to keep track of which tuples belong to which execution (see below). The databases $D_s$ are computed in a dynamic programming fashion, following the structure of the acyclic dDDP, as follows: for the initial node $s_0$, we have $D_{s_0} = D$. Otherwise, let $s_1, \ldots, s_m$ be the nodes having an edge going to $s$. Since we traverse nodes while respecting their order in the dDDP, for each $j$ we have already computed $D_{s_j}$. Let $E_{v_j}$ be the event associated with the edge from $s_j$ to $s$; we can compute the annotation for $t$ in $D_s$ to be $D_s(t) = \bigvee_{j} E_{v_j}((D_{s_0})(t))$. Observe that since $S$ is grounded, the only potential result tuples $t$ are ones obtained by a combination of values from $D$ and $S$, so their number is polynomial. Furthermore, by sharing sub-circuits as explained in the proof of Proposition 4.1, each $C(s, t)$ can be generated in polynomial time.

In addition, we need to encode the constraint that a valuation indeed corresponds to an execution. We note that a set of edges $E$ forms an execution if every two edges are reachable; there is an edge in $E$ which originate in the initial node; there is an edge in $E$ whose target is a final node and for every edge $e \in E$ there is an edge $e'$ whose target is the source of $e$. This constraint may easily be encoded via a polynomial size circuit $C'$. The resulting circuit for $T$ is designed as $\bigwedge_{t \in T} \bigvee_{s \in e} D_s(t) \bigwedge C'$.

In practice, note that the generation of provenance circuits for the individual tuples would be done separately from the final step of combining circuits of multiple tuples into a single circuit. This last step may be done upon user request which may also encode that some tuples are required not to appear.

**Using a SAT solver:** Once a circuit was generated, the problem of finding an execution yielding a sub-instance of interest is reduced to classical SAT solving applied to the circuit. Of course, SAT solving still entails an exponential time in the worst case, but may be performed efficiently in many practical cases (see also our experimental study).

**Accounting for user input:** Last, we consider non-grounded acyclic dDDPs. The difficulty is that there may now be infinitely many user inputs, and consequently there may be (1) infinitely many possible executions and (2) infinitely many values appearing in the set of tuples $T$. If we are given a bound over the latter, we may address the former, as follows.

Assume first that the set $\text{cons}$ of fresh values (i.e. values that neither appear in the dDDP nor in the database) in $T$ is given (we will relax this assumption) and its size is bounded by $k$ (this assumption will be kept). We then look at equivalence classes of executions with respect to this set, based on the following notion of isomorphism.

**Definition 7.6:** Let $D^{i}_{\text{internal}}$ be an initial database, and let $E_x_1$ and $E_x_2$ be two executions, both of length $n$, such that the $j$'th internal (user) database of $E_x_i$ ($1 \leq i \leq 2$) is denoted $D^{i}_{\text{internal}}(D^{i}_{\text{user}})$. Let $\text{cons}$ be a finite set of values. Let $U_i$ be the set of values in user input facts for $E_x_i$. Given a mapping $\phi : U_1 \mapsto U_2$ that satisfies that for each $u \in \text{cons}$, $\phi(u) = u$, we can apply it to a database by simply replacing every $u \in U_1$ by $\phi(u)$, and keeping all values intact for $D$. We say that $\phi$ is an isomorphism w.r.t. $E_x_1$ and $E_x_2$ if for every $j \leq n$ it holds that $\phi(D^{i}_{\text{internal}}) = D^{i}_{\text{internal}}$ and $\phi(D^{i}_{\text{user}}) = D^{i}_{\text{user}}$.

We generalize this to isomorphisms between two sets $E_1, E_2$ of executions in the obvious manner: for every execution in $E_1$ there is an isomorphic execution in $E_2$ and vice versa. Given a dDDP $S$ and an initial database, we denote by $S_{\approx \text{cons}(D)}$ the set of equivalences classes of executions in $S$ (w.r.t. $D$ and $\text{cons}$). We may then show:

**Proposition 7.7:** Let $S$ be an acyclic dDDP with fixed relations arity. Let $D$ be a database. Let $\text{cons}$ be a set of $k$ fresh values. We can compute in polynomial time a grounded acyclic dDDP $S'$ such that $S_{\approx \text{cons}(D)}$ is isomorphic to $S'_{\approx \text{cons}(D)}$.

Consequently, we can transform any acyclic dDDP to a grounded one, and apply Prop. 7.5 to generate a circuit. Then SAT solving is applied for "how-to" analysis, as before.

Finally, we note that the assumption of the set $\text{cons}$ being given may be relaxed, and we only need to assume its size to be bounded by a constant $k$: in this case we similarly generate $k$ new stub values to be replaced by concrete values from $T$ at the time of analysis.
We have conducted an experimental study to examine the scalability and usefulness of the approach in terms of the time and space consumption of provenance generation and the time it takes to use provenance for the different applications.

A. Evaluation Benchmark

To our knowledge, there is no “standard” benchmark for provenance tracking and no currently available system is able to support our analysis. We use a benchmark similar to the one we have developed in our previous work [6], enriched with update queries.

E-commerce: We have first considered a dDDP with a fixed topology, the one used in our running example (Figure 1). The underlying database was populated with synthetically and randomly generated data, of growing size, up to 3M tuples (to examine scalability w.r.t the database size). The user input was simulated, involving choices of varying number of products.

Arctic Stations: The second dataset is based on the “Arctic stations” benchmark, including processes that model the operation of meteorological stations in the Arctic [13]. Their flows are based on three kinds of topologies, serial, parallel and dense depicted in Figure 2. The number of nodes in the different process specifications is at most 25, but to examine scalability, we have varied the FSM sizes up to 1200 nodes, while following the topologies; this yields up to $10^{10}$ possible executions.

Scientific Workflows and Business Processes: We have further employed our construction in the context of Scientific Workflows from MyExperiment.org and Business Process (BP) specifications from bpmn.org. We show the result for three representative workflows and three representative BPs, named here Workflow1 (No. 16 in MyExperiment.org), Workflow2 (No. 204 there), Workflow3 (No. 72 there), BP1 (“order processing” process in bpmn.org), BP2 (“order fulfillment and procurement”), BP3 (“shipment process of hardware retailer”).

All experiments were executed on Windows 8, 64-bit, with 8GB of RAM and Intel Core i7 2.10 GHz processor.

B. Experimental Results

Tracking provenance for a single execution: We have simulated executions of the different workflows, tracked provenance for them (see Section IV) and measured the time and space required for doing so. A representative sample of the results for the E-commerce and arctic stations datasets is presented in Figures 3 and 4 respectively; similar results were obtained for the other datasets, since this part of the development is independent of the process state machine structure. First, we have fixed the execution length to be 5 (simulating the running example execution), and have gradually increased the database size. The results, in Figure 3a, indicate a moderate growth of the provenance size with respect to increasingly large databases, indicating the scalability of the approach (provenance size of about 11MB for output database of 3.5M tuples). Figure 3b presents the time required to generate the output provenance, compared with a provenance-oblivious execution (both using the same in-memory, non-optimized implementation). We note that the overhead incurred by provenance tracking is reasonable here (less than 25%). In Figure 4 we have fixed the output database size to be 1M tuples and have varied the execution length. The overall provenance size growth is moderate (since we have fixed the output database size) but the overhead in provenance tracking time significantly grows with respect to the execution length. Even so, its absolute running time is feasible: even for the unrealistically long execution with 400 steps, provenance tracking has incurred under 3 seconds.

What-if: We have further examined the usefulness of what-if analysis (see Section V), by comparing the performance of applying valuations to the tracked provenance expressions, with the time it takes to apply the hypothetical scenario directly to the underlying input database and then perform a provenance-oblivious execution. We have tested two versions of the latter: the same non-optimized implementation of query evaluation used when performing provenance tracking, and an optimized evaluation using an SQL engine (we have used MS SQL Server). The results are shown in Figure 3c: we observe that what-if analysis outperforms even the optimized re-computation; even for 3.5M output tuples, the time it takes to perform a what-if analysis was less than 0.5 second, while the computation time of the baseline approach was over a second, demonstrating a gain of up to 68%.

Why: Next, we have examined the performance of Algorithm 1, generating minimal explanations for a result of an execution for which provenance has been tracked. The results for a growing execution length in Figure 4c indicate good scalability in this respect. Observe that it returns a result in split-seconds even for unrealistically long executions. Note that the growth is more than linear, since greater execution length also results in longer evaluation time for the “what-if” analysis which is repeatedly invoked in course of the “why” analysis.

How-to: We have further implemented our solution (Section VII) for “how-to” analysis. Recall that a “pre-processing” step in our solution involves generating a compact representation of the provenance of all possible executions, which in turn serves as input to an “online” step of finding an execution of interest (which is done via SAT solving; we have used Limboole for this purpose). Naturally the dDDP structure has a significant effect on the algorithms performance; we have studied multiple (acyclic) structures and sizes of dDDPs, and next present a representative set of results.

First (Figure 5) we have examined the performance with respect to the number of possible executions. To this end, we have used the Arctic Stations dataset, have fixed the database size to be quite small (10K tuples), and have repeatedly duplicated the basic structures to obtain FSMs admitting up to $10^{112}$ different possible executions (the actual FSMs in this dataset include at most 24 nodes and entail at most 6561 executions, and all computations incurred split-seconds for

Fig. 2: Arctic Stations Finite State Machines

VIII. EXPERIMENTS

We have conducted an experimental study to examine the scalability w.r.t the database size). The user input was randomly generated data, of growing size, fixed topology, the one used in our running example (Figure 1). The number of possible executions is at most 25, but to examine scalability, we have varied the FSM sizes up to $10^{10}$ possible executions.

To our knowledge, there is no “standard” benchmark for provenance tracking and no currently available system is able to support our analysis. We use a benchmark similar to the one we have developed in our previous work [6], enriched with update queries.

E-commerce: We have first considered a dDDP with a fixed topology, the one used in our running example (Figure 1). The underlying database was populated with synthetically and randomly generated data, of growing size, up to 3M tuples (to examine scalability w.r.t the database size). The user input was simulated, involving choices of varying number of products.

Arctic Stations: The second dataset is based on the “Arctic stations” benchmark, including processes that model the operation of meteorological stations in the Arctic [13]. Their flows are based on three kinds of topologies, serial, parallel and dense depicted in Figure 2. The number of nodes in the different process specifications is at most 25, but to examine scalability, we have varied the FSM sizes up to 1200 nodes, while following the topologies; this yields up to $10^{10}$ possible executions.

Scientific Workflows and Business Processes: We have further employed our construction in the context of Scientific Workflows from MyExperiment.org and Business Process (BP) specifications from bpmn.org. We show the result for three representative workflows and three representative BPs, named here Workflow1 (No. 16 in MyExperiment.org), Workflow2 (No. 204 there), Workflow3 (No. 72 there), BP1 (“order processing” process in bpmn.org), BP2 (“order fulfillment and procurement”), BP3 (“shipment process of hardware retailer”).

All experiments were executed on Windows 8, 64-bit, with 8GB of RAM and Intel Core i7 2.10 GHz processor.

B. Experimental Results

Tracking provenance for a single execution: We have simulated executions of the different workflows, tracked provenance for them (see Section IV) and measured the time and space required for doing so. A representative sample of the results for the E-commerce and arctic stations datasets is presented in Figures 3 and 4 respectively; similar results were obtained for the other datasets, since this part of the development is independent of the process state machine structure. First, we have fixed the execution length to be 5 (simulating the running example execution), and have gradually increased the database size. The results, in Figure 3a, indicate a moderate growth of the provenance size with respect to increasingly large databases, indicating the scalability of the approach (provenance size of about 11MB for output database of 3.5M tuples). Figure 3b presents the time required to generate the output provenance, compared with a provenance-oblivious execution (both using the same in-memory, non-optimized implementation). We note that the overhead incurred by provenance tracking is reasonable here (less than 25%). In Figure 4 we have fixed the output database size to be 1M tuples and have varied the execution length. The overall provenance size growth is moderate (since we have fixed the output database size) but the overhead in provenance tracking time significantly grows with respect to the execution length. Even so, its absolute running time is feasible: even for the unrealistically long execution with 400 steps, provenance tracking has incurred under 3 seconds.

What-if: We have further examined the usefulness of what-if analysis (see Section V), by comparing the performance of applying valuations to the tracked provenance expressions, with the time it takes to apply the hypothetical scenario directly to the underlying input database and then perform a provenance-oblivious execution. We have tested two versions of the latter: the same non-optimized implementation of query evaluation used when performing provenance tracking, and an optimized evaluation using an SQL engine (we have used MS SQL Server). The results are shown in Figure 3c: we observe that what-if analysis outperforms even the optimized re-computation; even for 3.5M output tuples, the time it takes to perform a what-if analysis was less than 0.5 second, while the computation time of the baseline approach was over a second, demonstrating a gain of up to 68%.

Why: Next, we have examined the performance of Algorithm 1, generating minimal explanations for a result of an execution for which provenance has been tracked. The results for a growing execution length in Figure 4c indicate good scalability in this respect. Observe that it returns a result in split-seconds even for unrealistically long executions. Note that the growth is more than linear, since greater execution length also results in longer evaluation time for the “what-if” analysis which is repeatedly invoked in course of the “why” analysis.

How-to: We have further implemented our solution (Section VII) for “how-to” analysis. Recall that a “pre-processing” step in our solution involves generating a compact representation of the provenance of all possible executions, which in turn serves as input to an “online” step of finding an execution of interest (which is done via SAT solving; we have used Limboole for this purpose). Naturally the dDDP structure has a significant effect on the algorithms performance; we have studied multiple (acyclic) structures and sizes of dDDPs, and next present a representative set of results.

First (Figure 5) we have examined the performance with respect to the number of possible executions. To this end, we have used the Arctic Stations dataset, have fixed the database size to be quite small (10K tuples), and have repeatedly duplicated the basic structures to obtain FSMs admitting up to $10^{112}$ different possible executions (the actual FSMs in this dataset include at most 24 nodes and entail at most 6561 executions, and all computations incurred split-seconds for
Fig. 3: Provenance Size, Generation Time and What-if Analysis Time (Single Execution) as a Function of the DB size

Fig. 4: Provenance Size, Generation Time and Why Analysis Time (Single Execution) as a Function of Execution Length

Fig. 5: Provenance Size, Generation and Usage Time for “How-to” Analysis as a Function of Number of Executions

Fig. 6: Provenance Size, Generation and Usage Time for “How-to” Analysis as a Function of Fan-Out
these sizes). Provenance generation grows very moderately with the number of executions for the “parallel” structure, and incurs just over a minute even for the unrealistically large case of $10^{112}$ executions. Generation time is significantly higher for the complex “dense” structure, yet is still feasible even for unrealistically large executions. Given the provenance structure, finding a recommended execution was extremely fast; it has incurred only 0.2 seconds for the parallel topology and less than 4 seconds for the complex dense topology.

We have then examined the performance as a function of the state machine fan-out, using the Arctic Stations dataset. We have fixed the number of copies of the parallel (dense) topology to 10 (3 respectively), varying the fan-out up to 50 (again the database size is 10K). Here generation is fast, but the resulting provenance expression is complex for large fan-out and the dense topology. Consequently the runtime of the “how-to” analysis was about 30 seconds for the dense topology with fan-out 50, compared with 0.2 seconds for the parallel topology. We note that a fan-out of 50 is well beyond the real one for these workflows (the maximum in the benchmark of [13] was 24), and that for fan-out of 24 a recommended execution was found in less than 5 seconds. Last, we have varied the database size (Figures 7 and 8 for scientific workflows and BPs respectively). For large databases provenance generation requires significant time (about 14 minutes for the most complex structure), but the “online” process of recommending executions is again very fast: a maximum of about 5 (2.5) seconds was observed for the workflows (BPs).

IX. RELATED WORK

We briefly survey related work in multiple areas.

Data-centric processes: Multiple models for data-centric processes have been proposed in different lines of work such as Declarative Networking [14], [15], Collaborative Workflows [16], Webdatalog [17], and others. These works have typically focused on semantics, static analysis of correctness [18], [19], [15] and related questions, and have not studied provenance propagation or the kind of analysis questions made possible by provenance tracking. Unlike our model, the application state in these models is typically only implicitly captured (may be simulated through relations). Explicitly modeling the states and transitions has an impact on our provenance model and choice of problem statements (e.g. minimal explanations that are described in terms of transitions). On the other hand, our process model does not capture the full generality laid out by these models, and important omissions are the modeling of post-conditions (captured by FO formulas) and distribution. Accounting for these features in provenance tracking and analysis is left for future work.

Data provenance: Data provenance has been studied for different data transformation languages, from the positive relational algebra to Nested Relational Calculus and Datalog, with different provenance models (see e.g. [20], [10], [21], [22], [23], [8], [24], [25]) and applications (see e.g. [26], [4], [5]). The use of data provenance for explanations has been studied in e.g. [1], [2], its use for hypothetical reasoning has been studied in e.g. [3], and its use for how-to analysis
(“reverse data management”) has been studied in [4], [5]. These works, however, have neither considered workflows nor update queries, which leads to differences in provenance tracking (e.g. our use of circuits), in problems formulation (e.g. the explicit consideration of states), and in their solutions. In contrast, provenance for updates have been studied in [9], [7], [8], but the models do not include workflows, and in particular (1) do not study valuations of boolean variables that enable the “why” and “how-to” analysis studied here and (2) support a different notion of explanation (recall that our notions relied heavily on the workflow structure).

**Workflow Provenance:** Different approaches for capturing workflow provenance appear in the literature (e.g. [27], [12], [11], [28]), however there the focus is typically on the control flow and the dataflow between process modules, treating the modules themselves and their processing of data as black boxes (in contrast to the explicit, unbounded manipulation of data in our model). Recent works have focused on fine-grained tracking of workflow provenance, (see e.g. [29], [30], [31]), but to our knowledge none of these have studied the complexity of the analysis questions that we have focused on here (why, what-if, how-to) in the context of data-centric workflow provenance.

**Program Slicing:** In [32], [33] the authors study the notion of program slicing for a highly expressive model of functional programs and for Nested Relational Calculus, where the idea is to trace only relevant parts of the execution. The use cases are similar in spirit to that of our “why” and “what-if” analysis, but the focus in these works is mainly on highly complex programs rather than large data manipulation, leading to different techniques and technical results.

**X. Conclusion**

We have studied the analysis of complex data-centric applications, and have proposed a provenance-based solution. There are multiple intriguing directions to explore in future research. First, our focus was on boolean provenance; tracking metadata in additional domains is an intriguing task in this respect. We also intend to explore the development of provenance-based solutions for further features of data-centric processes, in particular parallelism and distribution.

**Acknowledgments:** This research was partially supported by the Israeli Science Foundation (ISF, grant No. 1636/13), by ICRC - The Blavatnik Interdisciplinary Cyber Research Center, by the Broadcom Foundation and Tel Aviv University Authentication Initiative, and by the ANR Aggreg project ANR-14-CE25-0017.

**References**


