Datalog as a Lingua Franca for Provenance Querying and Reasoning

Saumen Dey[*] *Sven Köhler*[*] *Shawn Bowers*[†] *Bertram Ludäscher*[*]

1 Introduction

The Open Provenance Model (OPM) [20] provides a small, extensible core for representing and exchanging provenance information in a technology-neutral manner. By design, OPM is a least common denominator, leaving aside certain aspects, including how to query provenance information. Similarly, OPM comes with a set of inference rules (e.g., for transitively closing some relations, or for stating temporal constraints that are implied by provenance assertions), but as pointed out by [15], the temporal semantics of OPM graphs is only partially defined in [20], leading to ambiguous or incompletely specified situations.

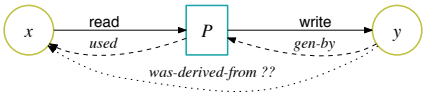


Fig. 1: Do the observables $x \overset{\mathsf{read}}{\rightarrow} P$ and $P \overset{\mathsf{write}}{\rightarrow} y$ imply that y *was-derived-from* x? Or that $t_{\mathsf{read}} < t_{\mathsf{write}}$ holds?

Example 1. Consider the provenance graph in Figure 1. In OPM, it shows a (data) *artifact* x that was *used* by *process* P, and another artifact y that *was-generated-by* (short: *gen-by*) P. In the terminology used in the rest of the paper, we say that the graph records a "read" and a "write" observable, denoted $x \overset{\mathsf{read}}{\rightarrow} P$ and $P \overset{\mathsf{write}}{\rightarrow} y$, respectively. Given this information, it may seem natural to assume that (1) y *was-derived-from* x (the dotted line), and that (2) the read event (t_{read}) occurred *before* the write event (t_{write}), i.e., $t_{\mathsf{read}} < t_{\mathsf{write}}$. However, neither (1) nor (2) are logical consequences of the provenance in Figure 1, i.e., we cannot infer that y *was-derived-from* x! Indeed, OPM correctly treats *was-derived-from* as a separate observable, e.g., P might have written y first, *then* read x afterwards, and so we should not assume (and thus not infer) that y *was-derived-from* x.[1]

If, on the other hand, the fact that y *was-derived-from* x has been (independently) asserted, can we then infer

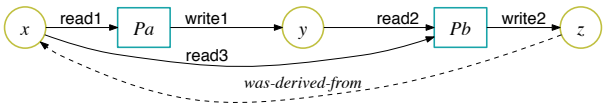


Fig. 2: z *was-derived-from* x. Does $t_{\mathsf{read}_2} < t_{\mathsf{write}_2}$ follow?

from Figure 1 that x was used by P *before* y was generated by P, i.e., that $t_{\mathsf{read}} < t_{\mathsf{write}}$ holds in our notation? Again, the somewhat surprising answer is: No! For example, the use (reading) and generation (writing) of x and y by P are not necessarily the only things that happened. In particular, there might be another derivation of y from x (which gave rise to the *was-derived-from* edge in the first place), making $t_{\mathsf{read}} > t_{\mathsf{write}}$ a real possibility.

Example 2. Consider the provenance graph in Figure 2, asserting that z *was-derived-from* x. Does $t_{\mathsf{read}_2} < t_{\mathsf{write}_2}$ or $t_{\mathsf{read}_3} < t_{\mathsf{write}_2}$ follow? As before, if no further information is available, we cannot infer either proposition: we simply don't know whether the path $x.P_a.y.P_b.z$ or the path $x.P_b.z$ (or yet another one, not included in the figure) are the reason for the *was-derived-from* edge in Fig. 2. The OPM semantics also handles this case correctly and does *not* imply that $t_{\mathsf{read}_i} < t_{\mathsf{write}_j}$ (for any $i \leq j$). On the other hand, OPM also does not provide a means to specify when such inferences would indeed be correct, e.g., in the common case where the result of a write is in fact directly dependent on an earlier read.

When employing OPM as a model for provenance recorded by a scientific workflow system [8], another limitation becomes apparent: OPM only deals with *retroactive provenance* (the usual data lineage captured in a trace graph T), but not with so-called *prospective provenance*[2], i.e., workflow specifications W, which are recipes for future (and past) derivations [18]. These were out of scope for OPM and, apparently, are also out of scope for current W3C standardization efforts [21].

On the other hand, it is easy to see that distinguishing between traces T and workflows W (and then interpreting the former as instances of the latter) can provide valuable information and additional functionality for provenance applications.

[*]UC Davis, (scdey | svkoehler | ludaesch)@ucdavis.edu
[†]Gonzaga University, bowers@gonzaga.edu

[1]If the computation P is a function or service call, then P indeed *first* consumes all inputs, *then* produces all outputs. This assumption is often correct, but a *process* P is not necessarily limited to such strict behavior and may interleave read/write events in many ways [13, 17].

[2]This "near-oxymoron" captures a practically useful notion: When a scientist is asked to explain how a certain result was obtained, in the absence of runtime traces, he can point to the script/workflow that was used to generate the data products; so workflows are provenance, too.

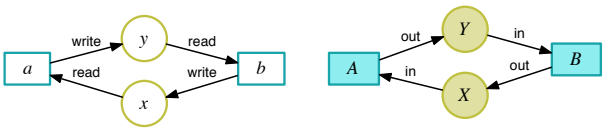


Fig. 3: Trace T (left) and workflow W (right).

Example 3. Consider the graphs in Figure 3: executing workflow W, might have produced the trace T. In order to validate T's structure, i.e., to check whether T can be an instance of W, we link its nodes and edges to W: e.g., edges $x \xrightarrow{\text{read}} a$ and $a \xrightarrow{\text{write}} y$ in T (data x was *read* and data y was *written* by process *invocation* a), have correponding edges $X \xrightarrow{\text{in}} A$ and $A \xrightarrow{\text{out}} Y$ in W, linking data *containers* X and Y to the *process* (or: actor) A. Clearly, in order to validate T's structure, we have to have a representation of the workflow structure W in the first place.

However, even if T is structurally valid w.r.t. W, other (here: temporal) inconsistencies may arise: The cycle in T indicates an inconsistent trace,[3] but we cannot be sure (for similar reasons as those in the previous examples). On the other hand, the cycle in W is usually *not* a concern: it simply means that W has a feedback loop, which is a rather common workflow pattern (cf. Appendix A).

Overview and Contributions. We propose logic rules as a formal foundation for graph-based, temporal models of provenance, for querying provenance graphs (traces), and for reasoning about traces and their connection to the workflows that generated them. In particular, we argue that Datalog provides a "lingua franca" for provenance:

(1) We adopt a semistructured data model, i.e., graphs with labeled edges $x \xrightarrow{\ell} y$, as a uniform representation of all provenance information, i.e., traces (à la OPM) and associated workflows of which they are instances. This allows us to employ regular path queries [3] as a convenient "macro-language" for concisely expressing generalized reachability queries on traces and workflows. By representing schema-level information (workflows) and instance-level information (traces) together in a single model, structural constraints can be expressed and checked easily using Datalog rules.

(2) We propose to use integrity constraints, i.e., rules of the form $\mathsf{false}_{\mathsf{ic}}(\bar{Y}) \leftarrow \mathit{denial}(\bar{X})$[4] as a way to express provenance semantics. Workflow systems differ in their models of computation (MoCs) and thus make different assumptions about how workflow components (a.k.a. *actors*, *processors*, *modules*, etc.) work, i.e., whether, and in which way, they can be stateful; how they consume their inputs, produce their outputs; and so on. As a result, different systems use different models of provenance (MoPs), with different temporal semantics. Thus, instead of "hard-wiring" a fixed temporal semantics to a particular graph-based MoP, we again use logic constraints to obtain a "customizable" temporal semantics.

[3] If read and write observables are temporally or causally linked, a *strict* partial order is implied and a cycle shouldn't have been observed.

[4] The rule body $\mathit{denial}(\bar{X})$ specifies the "bad" situations to be avoided; $\bar{Y} \subseteq \bar{X}$ are witnesses of constraint violations.

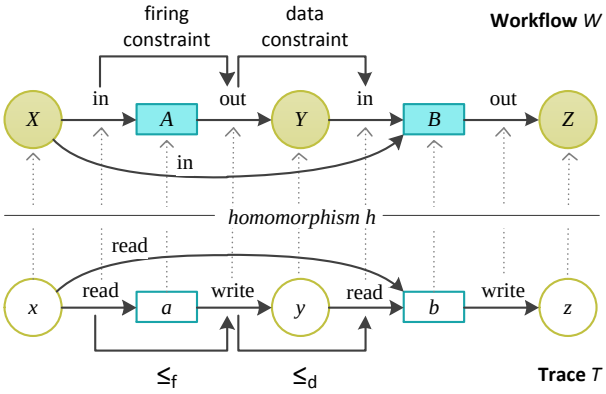


Fig. 4: Workflow W (top) vs Trace T (bottom): Traces are associated to workflows, guaranteeing structural consistency; workflow-level (firing or data) constraints induce temporal constraints $\leq_{\mathsf{f}}$ and $\leq_{\mathsf{d}}$ on traces.

(3) We illustrate this concept by providing *firing constraints* at the workflow level, which induce *temporal constraints* $\leq_{\mathsf{f}}$ at the level of traces (cf. Figure 4). These temporal-constraint generating rules can be chosen to conform to the temporal axioms in [15], or to accommodate a different temporal semantics, as implied by the MoC of a specific workflow or workflow system.

Due to limited space, we only illustrate some of the many ways in which logic rules and Datalog can be harnessed for provenance querying and reasoning; a detailed exposition is reserved for a long version of this paper. In the remainder, we give an overview of our approach and provide a few illustrative examples; additional examples and rules can be found in the appendix.

2 Unified Provenance Model

We would like to accommodate trace-level provenance information, schema-level information (workflow specifications), and temporal information in a single, uniform representation. To this end, we employ an underlying semistructured data model, which consists of labeled, directed graphs of the form $G = (V, E, L)$, with vertices V, labels L, and labeled edges $E \subseteq V \times L \times V$. In the following, we view workflows W and traces T as subgraphs of G. Similarly, our temporal model will consist of labeled edges (modeling one or more "*before*" relations).

Workflows. A workflow $W = (V_W, E_W, L_W)$ is a labeled graph whose nodes $V_W = C \cup P$ are data *containers* C and *processes* P. Processes are computational entities (often consisting of smaller internal steps, a.k.a. *invocations* or *firings*) that can send and receive data. *Containers* represent structures, e.g., FIFO queues, that hold data during the communication between processes. In a workflow W, edges $E_W = E_{\mathsf{in}} \cup E_{\mathsf{out}}$ are either *input* edges $E_{\mathsf{in}} \subseteq C \times \{\mathsf{in}\} \times P$, or *output* edges $E_{\mathsf{out}} \subseteq P \times \{\mathsf{out}\} \times C$, so $L_W = \{\mathsf{in}, \mathsf{out}\}$. We also write in(C,P) and out(P,C) to denote edges $\mathsf{C} \xrightarrow{\mathsf{in}} \mathsf{P}$ and $\mathsf{P} \xrightarrow{\mathsf{out}} \mathsf{C}$, respectively. The former means process P can read data artifacts from container C, while the latter means that process P can write data artifacts to container C.

Traces. A trace $T = (V_T, E_T, L_T)$ is a labeled graph whose nodes $V_T = D \cup I$ are *data artifacts* D or *process invocations* I; edges $E_T = E_{\mathsf{read}} \cup E_{\mathsf{write}} \cup E_{\mathsf{df}}$ are *read* edges $E_{\mathsf{read}} = D \times \{\mathsf{read}\} \times I$, *write* edges $E_{\mathsf{write}} = I \times \{\mathsf{write}\} \times D$, or *derived-from* edges $E_{\mathsf{df}} = D \times \{\mathsf{df}\} \times D$, so $L_T = \{\mathsf{read}, \mathsf{write}, \mathsf{df}\}$. The link between traces and workflows is established through homomorphisms:

Definition 1. Let $G = (V, E, L)$ and $G' = (V', E', L')$ be labeled graphs, and let $h = (h_1, h_2)$ be a pair of mappings $h_1 : V \to V'$ and $h_2 : L \to L'$. Then h is a *homomorphism* from G to G' if

$$(x \xrightarrow{\ell} y) \in E \;\text{ implies }\; (h_1(x) \xrightarrow{h_2(\ell)} h_1(y)) \in E'.$$

The functions h_1 and h_2 map nodes and labels from G to corresponding ones in G'. The model described here associates read and write edges in T with in and out edges in W, so $h_2 = \{\mathsf{read} \mapsto \mathsf{in}, \mathsf{write} \mapsto \mathsf{out}\}$. The mapping h_1 is usually given as part of a trace, i.e., when testing whether T is valid w.r.t. a workflow W, we do not have to *search* for h. Instead, traces needing validation already have corresponding workflow annotations embedded within them: We use mappings cont: $D \to C$ and proc: $I \to P$ to associate data items and process invocations with data containers and processes, respectively. The following rules derive false iff trace T with workflow mappings cont and proc are *not* a homomorphism:

```
false_Hom(read(D,I),in(C,P)) :-
    read(D,I), cont(D,C), proc(I,P), ¬ in(C,P).
false_Hom(write(I,D),out(P,C)) :-
    write(I,D), cont(D,C), proc(I,P), ¬ out(P,C).
```

Thus, if $\mathsf{false}_{\mathsf{Hom}}$ is empty, trace T is *structurally-valid* w.r.t. workflow W. Additional structural constraints for traces can be easily defined in a similar manner:

A *write-conflict* occurs when a data artifact has multiple incoming write edges; a *type-conflict* occurs when edges link nodes of the wrong type (e.g., directly linking invocations, instead of going through data nodes). Rules for checking such constraints are listed in Appendix C.

Temporal Model. We obtain a temporal semantics on top of the graph-based model by using rules to define a "*before*" relation (e.g., $\leq_{\mathsf{f}}$ and $\leq_{\mathsf{d}}$ in Fig. 4) amongst events. During workflow execution, the following events are observed: When a process is executed an *invocation* event is recorded, when a data artifact is read by an invocation a *read* event is recorded, and when an invocation writes a data artifact a *write* event is recorded. In the temporal model we constrain the order of events: bf(E1,E2) means that E1 happened *before* E2 and bfs(E1,E2) means that E1 happened *before or simultaneously* with E2.

A *data-constraint* between out-edges $\mathsf{P}_A \xrightarrow{\mathsf{out}} \mathsf{C}_Y$ and in-edges $\mathsf{C}_Y \xrightarrow{\mathsf{in}} \mathsf{P}_B$ of a data container C_Y at the workflow-level, implies an obvious temporal constraint at the trace-level: A write (creation) of data y must come before any read of y (also see Figure 4):

```
bf(write(I,D), read(D,J)) :- write(I,D), read(D,J).
```

Read, write, and derived-from edges in T capture dataflow. This information gives rise to a temporal order ("*flow-time*") among events. We infer this temporal order and create the corresponding temporal relations bf and bfs using the rules described here (and in the appendix). We make (safe) use of Skolem functions, e.g., to create unique identifiers, and to reify edges as nodes of a temporal structure: e.g., the (unspecified) execution time of an invocation is represented by a term invoc(I).

Before an invocation reads data, it must have started:

```
bfs(invoc(I), read(D,I)) :- read(D,I).
```

Similarly, a data artifact could not have been written after an invocation has been completed:

```
bfs(write(I,D), invoc(I)) :- write(I,D).
```

When a data artifact is written by a process invocation and read by another invocation, the former must not have started after the latter has completed:

```
bfs(invoc(I), invoc(J)) :- write(I,D), read(D,J).
```

When a data artifact is derived from another data artifact, the latter must have been written before the fomer:

```
bf(write(J,Y), write(I,X)) :- df(X,Y), write(I,X), write(J,Y).
```

A *firing-constraint* between in-edges $\mathsf{C}_X \xrightarrow{\mathsf{in}} \mathsf{P}_A$ and out-edges $\mathsf{P}_A \xrightarrow{\mathsf{out}} \mathsf{C}_Y$ of a process P_A is defined via a relation $\mathsf{fc}(\mathsf{C}_X, \mathsf{P}_A, \mathsf{C}_Y)$. This constraint ensures that any invocation of process P_A that reads data from container C_X and that writes into container C_Y has an associated temporal constraint: the invocation output depends on the read input.

If a user-defined constraint is provided by fc(X,B,Z) as shown in Figure 4, then the bf(read(x,b),write(b,z)) temporal relation at the trace-level can be infered:

```
bf(read(X,I), write(I,Y)) :-
    fc(C1,P,C2), read(X,I), proc(I,P),
    write(I,Y), cont(X,C1), cont(Y,C2).
```

Wall-Clock Time: In addition to temporal dependencies inferred from dataflow observables ("flow-time"), a provenance recorder may record events with a *wall-clock* timestamp. The observable time(E,T) means that event E was recorded with wall-clock time T. Wall-clock time and flow-time constraints can combined to infer additional temporal information for a trace:

bf(E1, E2) :− time(E1, T1), time(E2, T2), T1 < T2.
bfs(E1, E2) :− time(E1, T1), time(E2, T2), T1 ≤ T2.

3 Conclusions

The theory and practice of provenance are not always as well aligned as one may wish for.[5] The DBLP (databases & programming languages) community, among others, has been advancing our understanding of fundamental principles of provenance, and notions such as *Why*, *How*, and *Where* provenance [2, 6], provenance semirings [11], provenance as proof-trees, dependencies, program slicing [5], relationships to causal reasoning [19, 4], etc. are slowly becoming more widely known and understood better. At the same time, the many practical applications of provenance have led practitioners and system developers to move ahead rapidly with "provenance-enabling" their systems, proposing models, languages, and even W3C standards. As a contribution to further grow the connections between theory and practice of provenance [1], we have proposed to use logic rules, and Datalog in particular for querying provenance, checking structural constraints (e.g., whether a trace is valid, i.e., homomorph to its associated workflow), and specifying temporal constraints.

Datalog seems particularly well-suited as a "lingua franca" and bridge between theory and practice: On the one hand, there is a rich body of research and formal results about the complexity and expressiveness of Datalog and numerous fragments or extensions [7]. Queries on labeled graphs are well supported, e.g., regular path queries have a direct encoding in Datalog, and theoretical results on query containment and view-based query processing [3] can be exploited in various ways. Similarly, Datalog variants such as Statelog [16] or Datalog-LITE [10] provide means to naturally express temporal queries over provenance graphs. From a practitioner's point of view, Datalog is also attractive: powerful Datalog engines are available for experimentation with rule sets (e.g., to test and compare different provenance semantics, to query and debug programs [14] etc.), and can be used to deploy provenance querying and reasoning systems. Datalog prototypes for our approach are under development and can be demonstrated at the workshop.

[5] In theory, there is no difference between theory and practice. But in practice, there is.

Acknowledgements. Work supported in part by NSF award OCI-0722079 and DOE award DE-FC02-07ER25811.

References

[1] Acar, U., Buneman, P., Cheney, J., Van den Bussche, J., Kwasnikowska, N., and Vansummeren, S. A graph model of data and workflow provenance. In *TaPP* (2010).

[2] Buneman, P., Khanna, S., and Wang-Chiew, T. Why and where: A characterization of data provenance. In *ICDT* (2001).

[3] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Reasoning on regular path queries. *ACM SIGMOD Record 32*, 4 (2003), 83–92.

[4] Cheney, J. Causality and the semantics of provenance. *Arxiv preprint arXiv:1004.3241* (2010).

[5] Cheney, J., Ahmed, A., and Acar, U. Provenance as dependency analysis. In *DBPL* (Vienna, Austria, 2007), LNCS 4797, pp. 138–152.

[6] Cheney, J., Chiticariu, L., and Tan, W. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases 1*, 4 (2009), 379–474.

[7] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. Complexity and expressive power of logic programming. *ACM Computing Surveys 33*, 3 (2001), 374–425.

[8] Davidson, S., and Freire, J. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD* (2008).

[9] Dijkstra, E. W. Hamming's exercise in sasl, 1981. EWD-792.

[10] Gottlob, G., Grädel, E., and Veith, H. Datalog lite: A deductive query language with linear time model checking. *TOCL 3*, 1 (2002), 42–79.

[11] Green, T., Karvounarakis, G., and Tannen, V. Provenance semirings. In *PODS* (2007), pp. 31–40.

[12] Hemmendinger, D. The "hamming problem" in prolog. *ACM SIGPLAN Notices 23*, 4 (1988), 81–86.

[13] Kahn, G. The semantics of a simple language for parallel programming. *IFIP 74* (1974), 471–475.

[14] Köhler, S., Ludäscher, B., and Smaragdakis, Y. Declarative datalog debugging for mere mortals. submitted, 2012.

[15] Kwasnikowska, N., Moreau, L., and Van den Bussche, J. A formal account of the open provenance model. Tech. Rep. 21819, University of Southampton, December 2010.

[16] Lausen, G., Ludäscher, B., and May, W. On active deductive databases: The statelog approach. *Transactions and Change in Logic Databases* (1998), 69–106.

[17] Lee, E., and Parks, T. Dataflow process networks. *Proceedings of the IEEE 83*, 5 (1995), 773–801.

[18] Lim, C., Lu, S., Chebotko, A., and Fotouhi, F. Prospective and retrospective provenance collection in scientific workflow environments. In *Services Computing (SCC)* (2010).

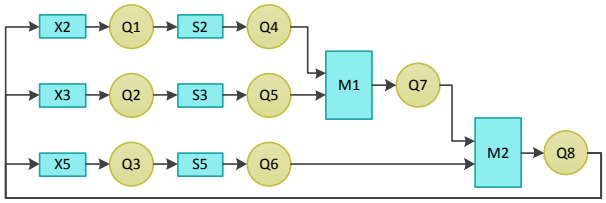
[19] Meliou, A., Gatterbauer, W., Halpern, J., Koch, C., Moore, K., and Suciu, D. Causality in databases. *IEEE Data Eng. Bull 33*, 3 (2010), 59–67.

[20] Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., et al. The open provenance model core specification (v1.1). *Future Generation Computer Systems 27*, 6 (2011), 743–756.

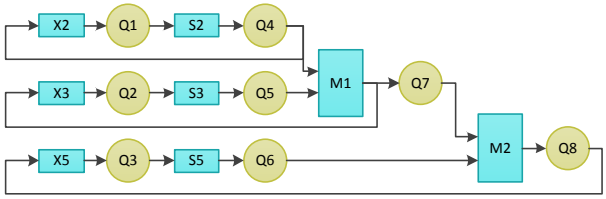
[21] W3C. Provenance working group. http://www.w3.org/2011/prov/. accessed 4/9/2012.

A Hamming Workflow Variants

To illustrate the earlier definitions of this paper, we use two variants of a workflow to compute the *Hamming numbers*[6] $H = \{2^i \cdot 3^j \cdot 5^k \mid i,j,k \geq 0\}$ incrementally, i.e., as an ordered sequence $1,2,3,4,5,6,8,9,10,12,15,\ldots$ Two workflow variants H_1 and H_3 are shown in Fig. 5. Note that the workflow graphs contain the same nodes (processes and containers), but are wired slightly differently (as it turns out, this makes a big difference). The data containers Q_i are queues (FIFO buffers); Q_8 is the distinguished output, where the Hamming numbers will appear in the correct order. M_1 and M_2 are *merge actors*, i.e., processes which take two ordered input sequences and merge them into an ordered output sequence. If presented with the same item in both streams, the output stream will only contain one copy of the element, so duplicates are removed. The actors X2, X3, and X5 multiply their inputs with 2, 3, and 5, respectively. Last not least, the *sample-delay actors* S2, S3, S5 are used "to prime the pump": initially (i.e., before reading any input), they output the number 1 to get the loop(s) going. Subsequently, they simply output whatever they received as an input. By design, the Hamming workflows H_1 and H_3 define an infinite output stream, i.e., the processes "run forever".



(a) Hamming workflow H_1: "one loop" variant



(b) Hamming workflow H_3: "three loops" variant

Fig. 5: Workflow variants H_1, H_3; output queue is Q_8.

A.1 Structural Validity

Figure 6 shows a (partial) trace T_H, i.e., for computing the Hamming numbers $n \leq 15$.[7] In order to test if a trace T is *structurally-valid* w.r.t. a workflow W, we check if there is a homomorphism from T to W. Here, a homomorphism between T_H, in Figure 6 and the 1-loop variant

[6] a.k.a. *regular numbers*; see [9, 12] for details

[7] Fig. 7 shows user-defined trace-views for $n \leq 1000$.

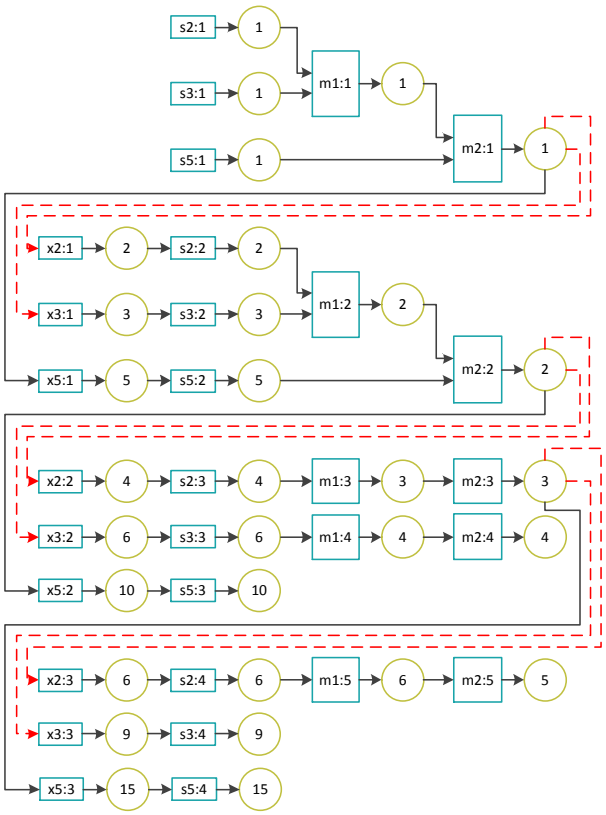


Fig. 6: Excerpt of a Hamming workflow trace T_H containing solid (black) and dashed (red) edges: This trace is homomorph to H_1 in Fig. 5(a) but it is *not* homomorph to H_3 in Fig. 5(b) since the dashed (red) edges in T_H cannot be mapped to corresponding edges in H_3.

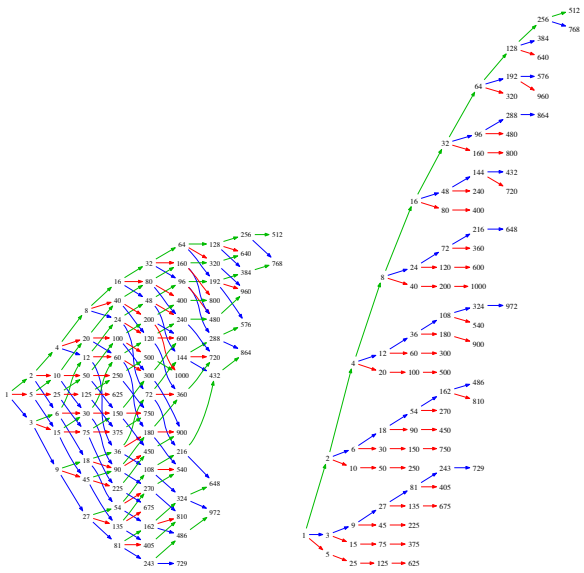
H_1 of the Hamming workflow in Figure 5(a) can be established. However, when attempting to find a homomorphism between the same trace and the 3-loop variant H_3 in Figure 5(b), corresponding in-edges cannot be found for the dashed (red) read-edges in Figure 6. Note that the "bad" (missing) edges can be found automatically with the Datalog (denial) rules for $\mathsf{false}_{\mathsf{Hom}}$ in Section 2.

A.2 User-Defined Provenance Queries

Our unified provenance model can easily be queried further using Datalog. For example, a user might want to know the lineage of a particular Hamming number (i.e., which other Hamming numbers "went into it"), or how many duplicates were derived in their particular workflow variant, etc. The dependencies between data items can be obtained by focusing on the read/write observables of certain processes P:

```
q(D1,P,D2) :- read(D1,I), write(I,D2), proc(I,P), focus(P).
```

Here, focus is a user-defined predicate to limit query answers to processes of interest, e.g., we may focus on



(a) (b)

Fig. 7: User-defined provenance for Hamming numbers up to 1000: (a) for H_1 ("*Fish*") and (b) for H_3 ("*Sail*")

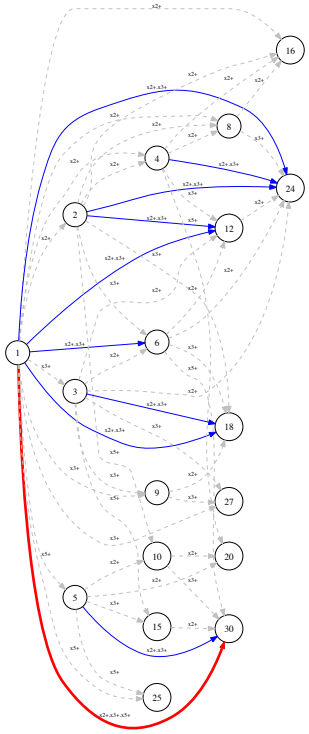


Fig. 8: Result of regular path query $\text{x2}^+ \cdot \text{x3}^+ \cdot \text{x5}^+$ on a subset of the Hamming graph (for nodes $N \leq 30$) in Fig. 7(a). Solid edges have labels $R_1 \cdot R_2$ (blue: intermediate results; bold, red: final results), while dashed edges have labels R^+. Base edges x2, x3, and x5 are omitted here.

x2, x3, and x5. Thus, tuples in the answer relation q can be viewed as edges $d_1 \xrightarrow{p} d_2$, linking data items to each other, with the label p denoting the process (multiplication factor) involved. Figure 7 shows the resulting graph structure[8] for a trace containing the computation of Hamming numbers up to 1000. Edge labels are represented via colors (here: green, red, and blue are used to represent labels "x2", "x3", and "x5", respectively).

One can clearly see on the in-degree of nodes that in Fig. 7(a) many Hamming numbers are produced in multiple ways, i.e., the custom-provenance graph is a DAG. In contrast, Hamming numbers in H_3 are produced by one path only *without* unnecessary duplicates as can be seen in Fig. 7(b), i.e., this graph is a tree.

B Regular Path Queries

Our unified provenance model in Section 2 is based on a semistructured data model, i.e., all information is represented using a labeled, directed graph $G = (V, E, L)$. Edges in this graph can be represented as triples $\mathsf{g}(x, \ell, y)$, where nodes $x, y \in V$ are connected via an ℓ-labeled edge.

A standard method to query such graphs is using regular path queries (RPQ) [3]. Let $\Sigma = \{\ell_1, \ldots, \ell_n\}$ be a set of base labels (here: $\Sigma = L$). Regular expressions over Σ are built in the usual way: if R, R' are regular expressions, then so are $R \mid R'$ (alternation) and $R \cdot R'$ (concatenation), as well as R^+ (transitive closure), R^* (reflexive transitive closure), etc. A number of additional operations can be defined, e.g., $R^?$ (optional edge), _ (any label), R^c (complement of R), etc. On labeled graphs, also R^- (inverse edge direction) is common. It is easy to see that RPQs can be easily expresses in Datalog: e.g., for $R \cdot R'$, $R \mid R'$, and R^+, the rules are (similarly for other expressions):

```
g(X,R1_conc_R2,Y) :- conc(R1,R2, R1_conc_R2),
        g(X,R1,Z),
        g(Z,R2,Y).
g(X,R1_or_R2,Y) :- or(R1,_, R1_or_R2),
        g(X,R1,Y).
g(X,R1_or_R2,Y) :- or(_,R2, R1_or_R2),
        g(X,R2,Y).

g(X,R_plus,Y) :- plus(R, R_plus),
        g(X,R,Y).
g(X,R_plus,Y) :- plus(R, R_plus),
        g(X,R,Z),
        g(Z,R_plus,Y).
```

The relations conc, or, and plus in the body can be defined, e.g., using Skolem functions as follows:

```
conc(R1, R2, R3) :- R3 = f_conc(R1,R2).
or(R1, R2, R3) :- R3 = f_or(R1,R2).
plus(R1, R2) :- R2 = f_plus(R1).
```

[8] Nodes are not intended to be readable, but if desired can be zoomed-into in the PDF version.

However, these rules are not safe under the standard bottom-up Datalog evaluation procedure[9], so instead we create a *finite* set of facts, consisting only of the subexpressions of a given user query R. For example consider the RPQ expression $R = (x2^+ \cdot x3^+) \cdot x5^+$. The bold, red relation in Fig. 8 shows the answer to the query:

```
ans(X,R,Y) :− g(X,R,Y), X ≤ 30, Y ≤ 30, derived(R).
```

where derived is defined as:

```
derived(R) :− conc(_,_,R).
derived(R) :− or(_,_,R).
derived(R) :− plus(_,R).
```

and where the facts in conc and plus are obtained via the subexpressions of the top-level user query:

```
conc(('x2+).(x3+)', 'x5+', '((x2+).(x3+)).x5+').
conc('x2+', 'x3+', '(x2+).(x3+)').

plus('x2', 'x2+').
plus('x3', 'x3+').
plus('x5', 'x5+').
```

Hamming Workflow Revisited. Consider again the Hamming workflow variants in Fig. 5 and the resulting user-defined provenance graphs in Fig. 7. It is easy to see that the "Sail" variant in Fig. 7(b) produces only paths that match the RPQ $x2^* \cdot x3^* \cdot x5^*$, whereas the more redundant "Fish" variant in Fig. 7(a) contains paths that match $(x2 \mid x3 \mid x5)^*$. In this way, RPQs can be used to query and inspect the structure of provenance graphs.

Note that via the construction outlined above, we can view RPQs as syntactic sugar for Datalog queries and freely mix them with other Datalog rules, including temporal queries in the style of Datalog-LITE [10].

[9]The usual top-down SLD-resolution of Prolog can handle such rules, however would in turn not be safe for the recursive RPQ rules.

C Further Examples

In the remainder, we present some further examples and rules, to illustrate our definitions in earlier sections. In our integrity constraint rules we employ the following auxilliary view for transitive dependencies:

```
dep(X,Y) :− read(X,Y).
dep(X,Y) :− write(X,Y).

tcdep(X,Y) :− dep(X,Y).
tcdep(X,Y) :− tcdep(X,Z), tcdep(Z,Y).

cycle(X,Y) :−tcdep(X,Y), tcdep(Y,X), ¬X=Y.
```

C.1 Structural Integrity Constraints

We compute the processes involved in a *write-conflict* using the following rule:

```
false_wc(X,Y) :−
    write(X,D), write(Y,D), ¬X=Y.
```

We compute *type-conflict* using the rule:

```
false_tc(X,Y) :−
    dep(X,Y), cont(X,C1), cont(Y,C2).
false_tc(X,Y) :−
    dep(X,Y), proc(X,P1), proc(Y,P2).
```

C.2 Temporal Integrity Constraints

An invocation must have started in order to read a data artifact:

```
false_rbi(D, I) :−
    bf(read(D,I), invoc(I)).
```

A write event must occur while the invocation is still executing:

```
false_ibw(I, D) :−
     bf(invoc(I), write(I,D)).
```

A dependent data must not be written before the write event of the data artifact on which it is dependent on:

```
false_dbd(X,Y) :−
     df(X,Y), bf(write(I,X), write(J,Y)).
```

When two nodes of a read event are in cycle, then data artifact must not be read before the invocation. When two nodes of a write event are in cycle, then invocation must not be completed before the write:

```
false_cdt(X,Y) :−
     cycle(X,Y), bf(read(X,Y), invoc(Y)).
false_cdt(X,Y) :−
     cycle(X,Y), bf(invoc(X), write(X,Y)).
```