Data & Knowledge Engineering xxx (2013) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Editorial

OPQL: Querying scientific workflow provenance at the graph level

Chunhyeok Lim [a], Shiyong Lu [a,*], Artem Chebotko [b], Farshad Fotouhi [a], Andrey Kashlev [a]

[a] Department of Computer Science, Wayne State University, Detroit, MI 48202, USA
[b] Department of Computer Science, University of Texas-Pan American, Edinburg, TX 78539, USA

ARTICLE INFO

Article history:
Received 21 December 2011
Received in revised form 30 August 2013
Accepted 31 August 2013
Available online xxxx

Keywords:
OPQL
Provenance query language
Scientific workflow provenance

ABSTRACT

Provenance has become increasingly important in scientific workflows to understand, verify, and reproduce the result of scientific data analysis. Most existing systems store provenance data in provenance stores with proprietary provenance data models and conduct query processing over the physical provenance storages using query languages, such as SQL, SPARQL, and XQuery, which are closely coupled to the underlying storage strategies. Querying provenance at such low level leads to poor usability of the system: a user needs to know the underlying schema to formulate queries; if the schema changes, queries need to be reformulated; and queries formulated for one system will not run in another system. In this paper, we present *OPQL*, a provenance query language that enables the querying of provenance directly at the graph level. An *OPQL* query takes a provenance graph as input and produces another provenance graph as output. Therefore, *OPQL* queries are not tightly coupled to the underlying provenance storage strategies. Our main contributions are: (i) we design *OPQL*, including six types of graph patterns, a provenance graph algebra, and *OPQL* syntax and semantics, that supports querying provenance at the graph level; (ii) we implement *OPQL* using a Web service via our OPMProv system; therefore, users can invoke the Web service to execute *OPQL* queries in a provenance browser, called OPMProVis. The result of *OPQL* queries is displayed as a provenance graph in OPMProVis. An experimental study is conducted to evaluate the feasibility and performance of OPMProv on *OPQL* provenance querying.

1. Introduction

Provenance, which is one kind of metadata that captures the derivation history of a data product, including its original data sources, intermediate products, and the steps that were applied to produce it, has become increasingly important in the area of scientific workflows [1-8] to interpret, validate, and analyze the result of scientific computing. In general, provenance captures past workflow execution and data derivation information (i.e., which tasks were performed and how data products were derived) via a provenance collection mechanism during workflow execution. Provenance captured typically holds data dependencies, process dependencies, causality between data and processes, and annotations. Such provenance is often represented by a provenance graph. For example, Fig. 1(a) shows a sample scientific workflow (which is the *Load Workflow* defined in the Third Provenance Challenge [9]) that checks and reads CSV files before loading, creates a database to load CSV files, loads them into tables and validates tables, and compacts a database after loading. Fig. 1(b) shows a sample provenance graph produced via the execution of the *Load Workflow*, where a node of a rectangle shape represents a process (i.e., a task), a node of an ellipse shape represents an artifact (i.e., a data product), which was used or generated by a process, a node of an octagon shape represents a contextual entity acting as a catalyst of a process, and an edge represents a causal dependency between its source denoting the

* Corresponding author. Tel.: +1 313 577 1667; fax: +1 313 577 6868.
E-mail addresses: chlim@wayne.edu (C. Lim), shiyong@wayne.edu (S. Lu), chebotkoa@utpa.edu (A. Chebotko), fotouhi@wayne.edu (F. Fotouhi), andrey.kashlev@wayne.edu (A. Kashlev).

0169-023X/$ – see front matter © 2013 Published by Elsevier B.V.
http://dx.doi.org/10.1016/j.datak.2013.08.008

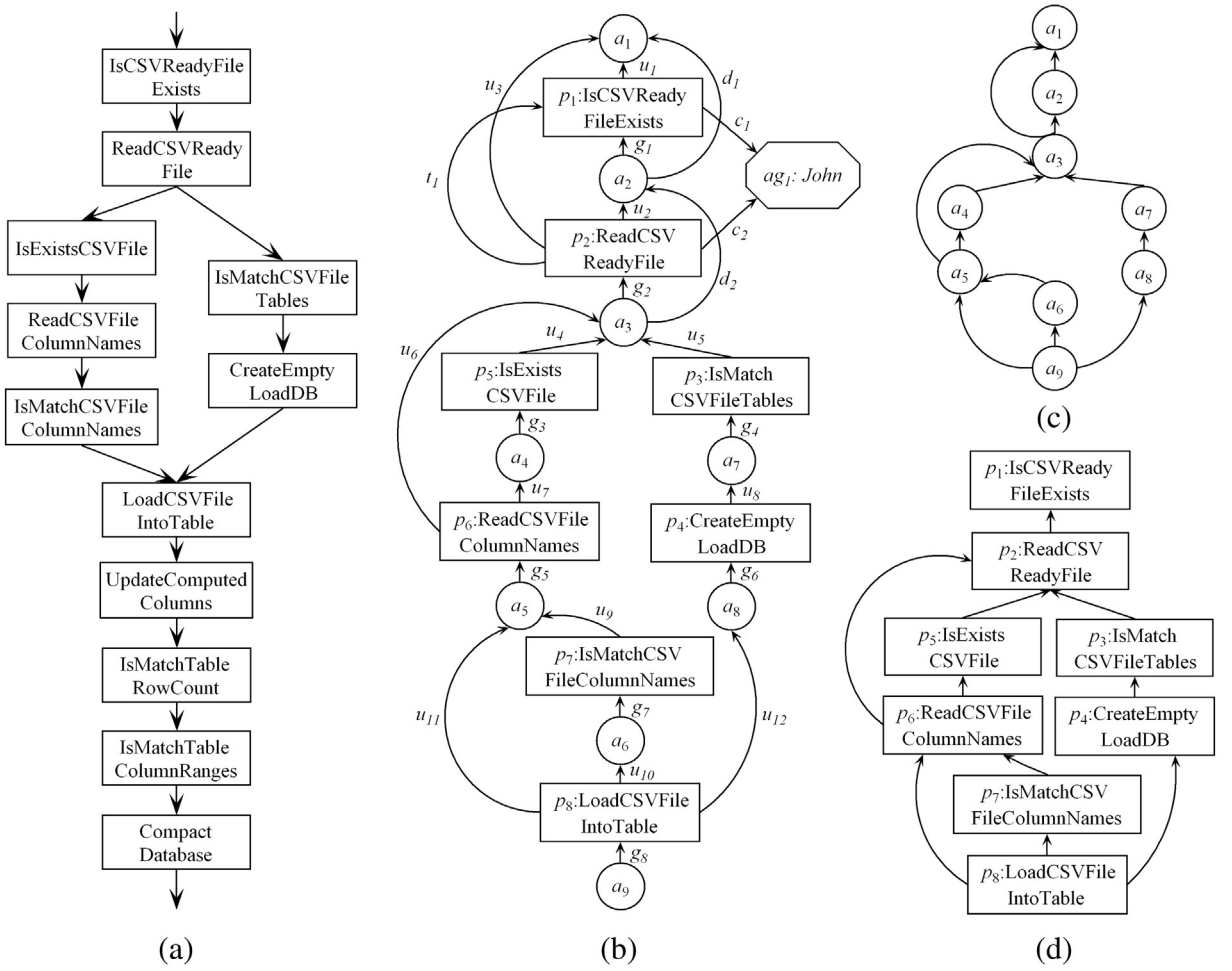


Fig. 1. An example of a scientific workflow and its provenance: (a) the *Load Workflow* defined in the Third Provenance Challenge; (b) a provenance graph generated via the execution of the *Load Workflow*; (c) a provenance graph representing data dependencies associated with artifact a_9; and (d) a provenance graph representing process dependencies associated with process p_8.

effect and its destination denoting the cause. Fig. 1(c) and (d) also represents a data dependency graph associated with artifact a_9 and a process dependency graph associated with process p_8 from the provenance graph, respectively.

Most existing systems [10,11,13,14] store provenance data in provenance stores with proprietary provenance data models and conduct provenance querying using languages such as SQL, SPARQL, and XQuery, over the physical provenance storages (i.e., relational, RDF, and XML). Such query languages are closely coupled to the underlying provenance storage strategies. As a result, querying provenance at such a low level leads to poor usability of the system because users need to know the underlying storage schema to formulate queries; if the schema changes, queries need to be reformulated; and queries formulated for one system will not run in another system. Moreover, to formulate complicated provenance queries, a user requires the expertise about grammars, syntax, and semantics of a query language. Using existing approaches, *provenance lineage queries* (queries for tracking ancestor nodes) often require a user to write recursive queries, directly typing recursive statements or using recursive functionality. For example, Fig. 2 shows two query languages SQL and *OPQL* answering for a provenance query (Q1), which asks for "which artifacts contributed to derive artifact a_5" over the provenance graph in Fig. 1(b). First, the SQL statement is expressed by a recursive query via the WITH ~ UNION ALL clause to track ancestor nodes associated with artifact a_5; thus, to answer query Q1, a user has to know the information of table *WasDerivedFrom* (i.e., how attributes define and what the attributes mean), a user needs the expertise to formulate a recursive query, which is nontrivial, and if the schema (i.e., the table information) changes, the SQL query needs to be reformulated, which is cumbersome. On the other hand, since the *OPQL* query is formulated at the graph level, a user does not have to know the storage schema; therefore, even though the storage schema changes, it does not affect the query. Moreover, *OPQL* construct WDF supports a recursive query pattern; thus, *OPQL* is easy and convenient to formulate a recursive query pattern.

In this paper, to address these issues, we propose *OPQL*, a provenance query language that enables the querying of provenance directly at the graph level. *OPQL* relies on the Open Provenance Model [15], a community-driven data model, which captures main aspects of the workflow provenance and does not enforce a particular physical representation of the provenance data. An *OPQL*

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

Q1: Which artifacts contributed to derive artifact a_5 ?

```
<SQL>
WITH Lineage (EffectArtifactID, CauseArtifactID) AS
    (SELECT EffectArtifactID, CauseArtifactID FROM WasDerivedFrom
    WHERE EffectArtifactID = 'a5'
    UNION ALL
    SELECT L.EffectArtifactID, D.CauseArtifactID FROM WasDerivedFrom D, Lineage L
    WHERE L.CauseArtifactID = D.EffectArtifactID)
SELECT * FROM Lineage;
* Table: WasDerivedFrom (EffectArtifactID, CauseArtifactID)

<OPQL>
    WDF (a5);
```

Fig. 2. A sample provenance query answered by query languages SQL and *OPQL*, respectively.

query takes an OPM-compliant provenance graph as input and produces another OPM-compliant provenance graph as output. Thus, *OPQL* queries are not tightly coupled to the underlying storage strategies. This paper has the following main contributions: (1) we design *OPQL*, including six types of graph patterns, a provenance graph algebra, and *OPQL* syntax and semantics, that supports querying provenance at the graph level; and (2) we implement *OPQL* using a Web service via the OPMProv system; therefore, users can invoke the Web service to execute *OPQL* queries in a provenance browser, called OPMProVis. The result of *OPQL* queries is displayed as a provenance graph in OPMProVis.

The rest of the paper is organized as follows. In Section 2, we overview the OPMProv system that stores, reasons, and queries scientific workflow provenance using a relational database. In Section 3, we present six types of graph patterns, a provenance graph algebra, and *OPQL* syntax and semantics to support querying provenance at the graph level. In Section 4, we discuss how *OPQL* is implemented via our OPMProv system. Section 5 describes how OPM graphs are reconstructed using our *GraphConstruct* algorithm. Section 6 reports the experimental study confirming the feasibility of using *OPQL* to query provenance at the graph level. Section 7 discusses related work on provenance query processing in existing systems. Section 8 presents conclusions and discusses possible future research directions. Finally, Acknowledgments section concludes the paper.

2. OPMProv overview

OPMProv is a relational database-based provenance system that stores, reasons, and queries scientific workflow provenance [16]. OPMProv supports storing and querying both prospective provenance that captures an abstract workflow specification as a recipe for future data derivation and retrospective provenance that captures past workflow execution and data derivation, which are collected via a provenance collection framework [17] while most existing systems focus on storing and querying only retrospective provenance. Moreover, OPMProv supports provenance reasoning using recursive views and SQL queries alone without any external reasoning engine. More details on the implementation and performance of OPMProv can be found in [16].

In this paper, we continue our efforts with the OPMProv system to enhance the following main capabilities: (1) OPMProv supports *OPQL*, a provenance query language that enables the querying of provenance directly at the graph level while previous versions of OPMProv supported provenance query processing in SQL, which is closely coupled to the underlying storage schema. We also expose the functionality of *OPQL* through a Web service interface so that other systems can use it; and (2) OPMProv implements a provenance browser that enables a user to display provenance graphs and invoke the *OPQL* Web service for provenance querying. We present an overview of the current version of OPMProv in the context of the VIEW workflow system [18] in Fig. 3.

View is an online scientific workflow management system (www.viewsystem.org) that allows users to create, edit, and run visual scientific workflows online. A scientific workflow represents a multiple-step data analysis pipeline that chains several data analysis modules (e.g. Web Services, scripts) together via data links, which connect the output of one module to the input of another module. As shown in Fig. 3(a) View is composed of six major functional subsystems, including *Workbench*, *Workflow Engine*, *Worklfow Monitor*, *Data Product Manager*, *Task Manager*, and *Provenance Manager*. Workbench implements functions of workflow design, presentation and visualization. Workflow Engine is responsible for executing a workflow by running each module and passing the output result as input for the next module in the chain. Workflow Monitor oversees each workflow run by tracking execution status of individual modules and the entire workflow as a whole. Data Product Manager allows to store and retrieve artifacts (data products) from its repository. The purpose of Task Manager is to execute individual task-based modules. The underlying tasks include Web Services, scripts, local applications, grid jobs etc. Finally, Provenance Manager (OPMProv) is responsible for storing, querying and reasoning prospective and retrospective workflow provenance data. As shown in Fig. 3, OPMProv plays a role of *Provenance Manager* in our VIEW system.

Fig. 3(b) shows the three-layer architecture of OPMProv. The provenance presentation layer provides users with the functionalities of provenance querying, data insertion, and provenance visualization via OPMProVis^{D} (desktop version) and OPMProVis^{W} (web version) as user-friendly GUIs. The provenance presentation layer interacts with the provenance service layer

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

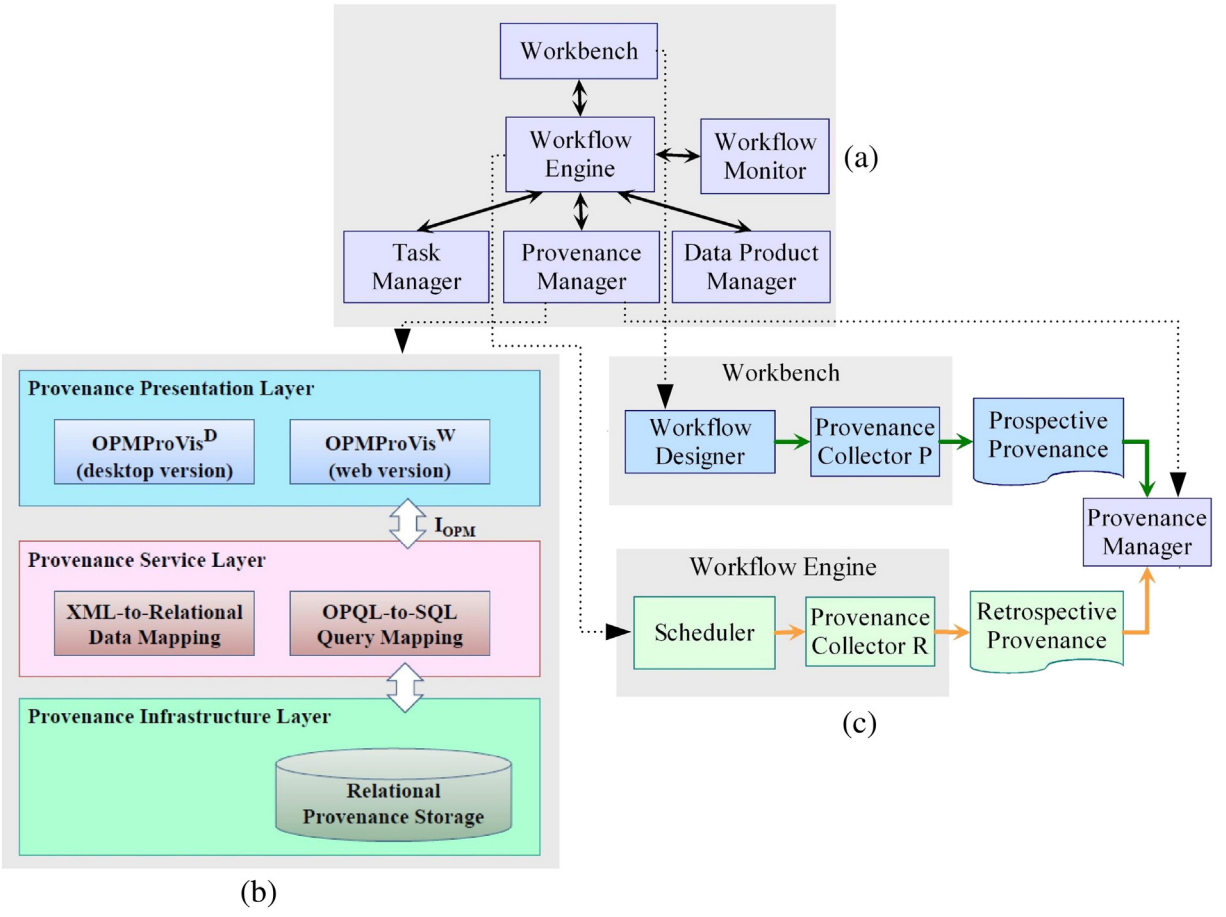


Fig. 3. An overview of the OPMPROV system: (a) an architecture of the View system; (b) an architecture of the OPMPROV system; and (c) an overview of the provenance collection framework.

via interface I_{OPM} that is defined and described by WSDL. The provenance service layer provides users with provenance services. OPMPROV currently provides two Web services that employ two mappings: (1) one is to insert provenance data into OPMPROV using an XML-to-Relational data mapping that maps XML documents to relational tuples; and (2) the other is to execute *OPQL* queries from OPMPROV using an OPQL-to-SQL query mapping that translates OPQL queries into SQL queries. These mappings interconnect the provenance service layer and the provenance infrastructure layer, where the latter is represented by a relational database management system that plays a role of a relational provenance storage backend. Fig. 3(c) also shows an overview of our provenance collection framework, where both prospective and retrospective provenance captured are stored and managed in OPMPROV.

Because experiment reproducibility and provenance reasoning are known to be vital aspects in scientific workflow management [16,18], the role of OPMPROV in the workflow management system is crucial.

3. The OPQL query language

In this section, we present *OPQL*, a provenance query language that enables the querying of provenance at the graph level. We first describe the OPMPROV provenance model which is used as a fundamental provenance model for *OPQL*. Next, we define six types of graph patterns which are the main building blocks of an *OPQL* query and a provenance graph algebra for *OPQL*. We then propose *OPQL* syntax and semantics. Finally, we discuss how provenance queries can be expressed in *OPQL*.

3.1. The OPMPROV provenance model

We adopt the notion of nodes and edges proposed in OPM [15] to define the OPMPROV provenance model. As a result, the OPMPROV provenance model is represented as a directed graph expressing the causal dependencies between nodes. In particular, a provenance graph is composed of three types of nodes (i.e., *Artifact*, *Process*, and *Agent*) and five types of edges (i.e., *WasGeneratedBy*, *Used*, *WasDerivedFrom*, *WasTriggeredBy*, and *WasControlledBy*), which represent causal dependencies between nodes. For example, in Fig. 1(b), an artifact (*an immutable piece of state*), process (*an action or a series of actions*), and

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

agent (*a contextual entity acting as a catalyst of a process*) are represented as an ellipse, rectangle, and octagon shape, respectively, and an edge is represented by an arc and denotes the presence of a causal dependency between the source of the arc (the effect) and the destination of the arc (the cause). Each node and edge has a set-valued *Account* attribute, which allows to associate it with multiple accounts. In Fig. 1(b), the edges represent the following causal dependencies: (1) edge *Used* (u_1–u_{12}): p_1 *used* a_1, p_2 *used* a_1 and a_2, p_3 *used* a_3, and so on; (2) edge *WasGeneratedBy*(g_1–g_8): a_2 *was generated by* p_1, a_3 *was generated by* p_2, a_4 *was generated by* p_5, and so on; (3) edge *WasDerivedFrom* (d_1, d_2): a_2 *was derived from* a_1 and a_3 *was derived from* a_2; (4) edge *WasTriggeredBy* (t_1): p_2 *was triggered by* p_1; and (5) edge *WasControlledBy* (c_1, c_2): p_1 and p_2 *were controlled by* ag_1.

We formalize a provenance graph as follows. A *provenance graph* $PG = (N, E)$ consists of:

1 a set of nodes $N = A \cup P \cup AG$, where A is a set of artifacts, P is a set of processes, and AG is a set of agents;
2 a set of directed edges $E = E_u \cup E_g \cup E_d \cup E_t \cup E_c$, where i) $E_u \subseteq P \times A$ and $(p, a) \in E_u$ states that process p *used* artifact a, ii) $E_g \subseteq A \times P$ and $(a, p) \in E_g$ states that artifact a *was generated by* process p, iii) $E_d \subseteq A \times A$ and $(a_1, a_2) \in E_d$ states that artifact a_1 *was derived from* artifact a_2, iv) $E_t \subseteq P \times P$ and $(p_1, p_2) \in E_t$ states that process p_1 *was triggered by* process p_2, and v) $E_c \subseteq P \times AG$ and $(p, ag) \in E_c$ states that process p *was controlled by* agent ag.

Moreover, in the OPMProv provenance model, each node and edge have arbitrary properties; thus, we use a tuple, a list of name and value pairs, to denote these properties. Fig. 4 shows a provenance graph that represents dependencies associated with process p_2 in Fig. 1(b).

3.2. Graph patterns

We extend the notion of graph pattern proposed in [19] to support provenance queries over a provenance graph. In this work, we define six types of graph patterns, which are the main building blocks of an *OPQL* query.

Definition 1. Graph pattern: type $\mathbb{B}$.

A *graph pattern* P_b is a pair (M, C), where M is a graph motif and C is a predicate on the properties of the motif.

A sample graph pattern P_b is shown in Fig. 5. It includes the two node graph motif and the predicate stating specific values of the nodes.

Definition 2. Graph pattern: type $\mathbb{O}$.

A *graph pattern* P_o is a triple (M, O, C), where M is a graph motif, O is an inverse-functional one-to-many mapping that returns a set of nodes that have direct causal dependencies associated with a node, and C is a predicate on the properties of the motif. To handle five causal dependencies (i.e., *WasGeneratedBy*, *Used*, *WasDerivedFrom*, *WasTriggeredBy*, and *WasControlledBy*) in a provenance graph, O is composed of ten types of mapping functions $(i.e., O \in \{O_u, O_{\dot{u}}, O_g, O_{\dot{g}}, O_d, O_{\dot{d}}, O_t, O_{\dot{t}}, O_c, O_{\dot{c}}\})$ as defined below:

$$\begin{aligned} O_u(p) &= \{a \mid (p,a) \in E_u\} \\ O_{\dot{u}}(a) &= \{p \mid (p,a) \in E_u\} \\ O_g(a) &= \{p \mid (a,p) \in E_g\} \\ O_{\dot{g}}(p) &= \{a \mid (a,p) \in E_g\} \\ O_d(a_1) &= \{a_2 \mid (a_1,a_2) \in E_d\} \\ O_{\dot{d}}(a_2) &= \{a_1 \mid (a_1,a_2) \in E_d\} \\ O_t(p_1) &= \{p_2 \mid (p_1,p_2) \in E_t\} \\ O_{\dot{t}}(p_2) &= \{p_1 \mid (p_1,p_2) \in E_t\} \\ O_c(p) &= \{ag \mid (p,ag) \in E_c\} \\ O_{\dot{c}}(ag) &= \{p \mid (p,ag) \in E_c\}. \end{aligned} \tag{1}$$

graph *PG* {
  node v_1<id='a_2', value=''>;
  node v_2<id='a_3', value=''>;
  node v_3<id='p_1', value='IsCSVReadyFileExists'>;
  node v_4<id='p_2', value='ReadCSVReadyFile'>;
  node v_5<id='ag_1', value='John'>;
  edge $e_1(v_4,v_1)$<id='u_2', role='used'>;
  edge $e_2(v_2,v_4)$<id='g_2', role='wasGeneratedBy'>;
  edge $e_3(v_4,v_3)$<id='t_1', role='wasTriggeredBy'>;
  edge $e_4(v_4,v_5)$<id='c_2', role='wasControlledBy'>;
};

Fig. 4. A provenance graph representing dependencies associated with process p_2 in Fig. 1(b).

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

graph P_b {
  node v_1;
  node v_2;
}
where v_1.value = 'butter'
and v_2.value = 'bake';

Fig. 5. Sample graph pattern P_b.

Graph pattern P_o is a derived graph pattern. It enables users to formulate provenance queries that find direct causal dependencies associated with a node. Fig. 6 shows a sample graph pattern P_o that includes the two node graph motif, the mapping function (O_u), and the predicate stating the identifier of node v_1.

Next, we define the following four graph patterns to support tracking of ancestor nodes.

Definition 3. Graph pattern: type $\mathbb{D}$.

A *graph pattern* P_d is a triple (M, D, C), where M is a graph motif, D is an inverse-functional one-to-many mapping that returns a set of artifacts that were applied to derive an artifact, and C is a predicate on the properties of the motif. To support tracking of ancestor nodes associated with artifacts forward and backward, D is composed of two types of mapping functions (i.e., $D \in \{D^{fwd}, D^{bwd}\}$) as defined below:

$$D^{fwd}(a) = \bigcup_{a' \in O_d(a)} D^{fwd}(a') \cup O_d(a) \tag{2}$$

$$D^{bwd}\left(a'\right) = \bigcup_{a \in O_d\left(a'\right)} D^{bwd}(a) \cup O_{\hat{d}}(a'). \tag{3}$$

Graph pattern P_d is a derived graph pattern. It enables users to formulate recursive queries to track ancestor nodes associated with artifacts. For example, Fig. 7 shows a sample graph pattern P_d that includes the two node graph motif, the mapping function (D^{fwd}), and the predicate stating the identifier of node v_1.

Definition 4. Graph pattern: type $\mathbb{T}$.

A *graph pattern* P_t is a triple (M, T, C), where M is a graph motif, T is an inverse-functional one-to-many mapping that returns a set of processes that were applied to trigger a process, and C is a predicate on the properties of the motif. To support tracking of ancestor nodes associated with processes forward and backward, T is composed of two types of mapping functions (i.e., $T \in \{T^{fwd}, T^{bwd}\}$) as defined below:

$$T^{fwd}(p) = \bigcup_{p' \in O_t(p)} T^{fwd}(p') \cup O_t(p) \tag{4}$$

graph P_o {
  node v_1;
  node v_2;
}
mapping $O_u : v_1 \xrightarrow{used} v_2$
where v_1.id = 'p_1';

Fig. 6. A sample graph pattern P_o.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

graph P_d {
  node v_1;
  node v_2;
}
mapping D^{fwd} : $v_1 \xrightarrow{wasDerivedFrom^*} v_2$
where v_1.id = 'a_n';

Fig. 7. A sample graph pattern P_d.

$$T^{bwd}(p') = \bigcup_{p \in O_t(p')} T^{bwd}(p) \cup O_t(p'). \tag{5}$$

Graph pattern P_t is also a derived graph pattern. It enables users to formulate recursive queries to track ancestor nodes associated with processes. Fig. 8 shows a sample graph pattern P_t that includes the two node graph motif, the mapping function (T^{fwd}), and the predicate stating the identifier of node v_1.

Definition 5. Graph pattern: type $\mathbb{G}$.

A *graph pattern* P_g is a triple (*M*,*G*,*C*), where *M* is a graph motif, *G* is an inverse-functional one-to-many mapping that returns a set of processes that were applied to generate an artifact, and *C* is a predicate on the properties of the motif. *G* is defined as:

$$G(a) = \bigcup_{p' \in O_g(a)} T^{fwd}(p') \cup O_g(a). \tag{6}$$

Definition 6. Graph pattern: type $\mathbb{U}$.

A *graph pattern* P_u is a triple (*M*,*U*,*C*), where *M* is a graph motif, *U* is an inverse-functional one-to-many mapping that returns a set of artifacts that were used by a process, and *C* is a predicate on the properties of the motif. *U* is defined as:

$$U(p) = \bigcup_{a' \in O_u(p)} D^{fwd}(a') \cup O_u(p). \tag{7}$$

Graph patterns P_g and P_u are derived graph patterns. These graph patterns enable users to formulate recursive queries to track ancestor nodes associated with processes and artifacts, respectively. It can be shown that graph patterns P_g and P_u can be derived by graph pattern P_b similarly to the previous examples.

Next, we define three types of *graph pattern matching* which generalize subgraph isomorphism over six graph patterns.

Definition 7. Graph pattern matching α.

A *graph pattern* P_b is matched with a graph *PG* if there exists an injective mapping ϕ_a: $N(M) \rightarrow N(PG)$ such that i) For $\forall\, e(u,v) \in E(M)$, $(\phi_a(u),\phi_a(v))$is an edge in *PG*, and ii) predicate $C_{\phi_\alpha}(PG)$ holds.

graph P_t {
  node v_1;
  node v_2;
}
mapping T^{fwd} : $v_1 \xrightarrow{wasTriggeredBy^*} v_2$
where v_1.id = 'p_m';

Fig. 8. A sample graph pattern P_t.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

Definition 8. Graph pattern matching β.

A *graph pattern* P_o is matched with a graph PG if there exists an injective mapping ϕ_β: $N(M) \rightarrow N(PG)$ such that i) for $\forall\ e(u,v) \in E(M)$, $(\phi_\beta(u),\phi_\beta(v))$ is an edge in PG, ii) function $O_{\phi_\beta}(PG)$ holds, and iii) predicate O_{ϕ_β} (PG) holds.

Definition 9. Graph pattern matching γ.

Each of graph patterns (P_d, P_t, P_g, and P_u) is matched with a graph PG if there exists an injective mapping ϕ_γ: $N(M) \rightarrow N(PG)$ such that i) For $\forall\ e(u,v) \in E(M)$, $(\phi_\gamma(u),\phi_\gamma(v))$ is an edge in PG, ii) each function ($D_{\phi_\gamma}(PG)$, $T_{\phi_\gamma}(PG)$, $G_{\phi_\gamma}(PG)$, and $U_{\phi_\gamma}(PG)$) holds, and iii) each predicate $C_{\phi_\gamma}(PG)$ holds.

To denote the binding between a graph pattern and a provenance graph, we define a *matched graph* as follows.

Definition 10. Matched graph.

Given an injective mapping $\phi \in \{\phi_a, \phi_\beta, \phi_\gamma\}$ between a pattern $P \in \{P_b, P_o, P_d, P_t, P_g, P_u\}$ and a provenance graph PG, a matched graph is a triple (ϕ, P, PG) and is defined as $\phi_P(PG)$.

3.3. Provenance graph algebra

We propose a provenance graph algebra for the *OPQL* query language. The provenance graph algebra is based on four operators, which operate on a provenance graph. Each operator takes a provenance graph as input and produces another provenance graph as output. Our union, intersection, and difference operators take two provenance graphs that are subgraphs of the same provenance graph produced by other queries as input and produce a provenance graph as output. We define the following four operators to manipulate and query a provenance graph.

3.3.1. Extract operator (δ)

One of the most frequent operations performed on a provenance graph is the extraction of a set of nodes and edges. An extract operator is defined using a graph pattern P. It takes a provenance graph (PG) as input and produces a new provenance graph that matches the graph pattern as output, denoted by $\delta_p(PG)$. For example, let Fig. 1(b) be a provenance graph (PG). You might want to find all artifacts that contributed to derive artifact a_6. Using the extract operator, this query can be expressed as:

$$\delta_{[P_d:D^{fwd}(a_6)]}(PG). \tag{8}$$

This query first generalizes a matched graph which consists of a set of artifacts ($a_1 - a_6$) and a set of edges ($d_1 - d_5$) via the *graph pattern matching* γ (i.e., ϕ_γ) and then it produces a new provenance graph by combining information from the matched graph. The output of the extract operator is a provenance graph:

$$\delta_p(PG) = \phi_P(PG). \tag{9}$$

Next, we define set operators, union, intersection, and difference. These operators are operated on a provenance graph, but they take two subgraphs produced by other queries as input and produce a provenance graph as output. Let PG be a provenance graph, and let PG_1 and PG_2 be the output of $\delta_{P_1}(PG)$ and $\delta_{P_2}(PG)$, respectively. Given two subgraphs $PG_1 = (N_1, E_1)$ and $PG_2 = (N_2, E_2)$, where PG_1 and $PG_2 \subseteq PG$, these operators are defined as follows.

3.3.2. Union operator ($\cup$)

The union operator calculates the union of two subgraphs. A union operation is defined by $PG_1 \cup PG_2$, resulting in a provenance graph $PG' = (N', E')$, where

$$\begin{aligned} N' &= \{n \mid n \in N_1 \text{ or } n \in N_2\} \\ E' &= \{e \mid e \in E_1 \text{ or } e \in E_2\}. \end{aligned} \tag{10}$$

For example, let Fig. 9(a) be a provenance graph (PG). Then, Fig. 9(b) and (c) represents the output of $\delta_{[P_d:D^{fwd}(a_5)]}(PG)$ and $\delta_{[P_d:D^{fwd}(a_8)]}(PG)$, respectively. You might want to find all artifacts that contributed to derive either artifact a_5 or artifact a_8 over provenance graph G. Using the union operator, this query can be expressed as $\delta_{[P_d:D^{fwd}(a_5)]}(PG) \cup \delta_{[P_d:D^{fwd}(a_8)]}(PG)$. The result of the query is shown in Fig. 9(d).

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

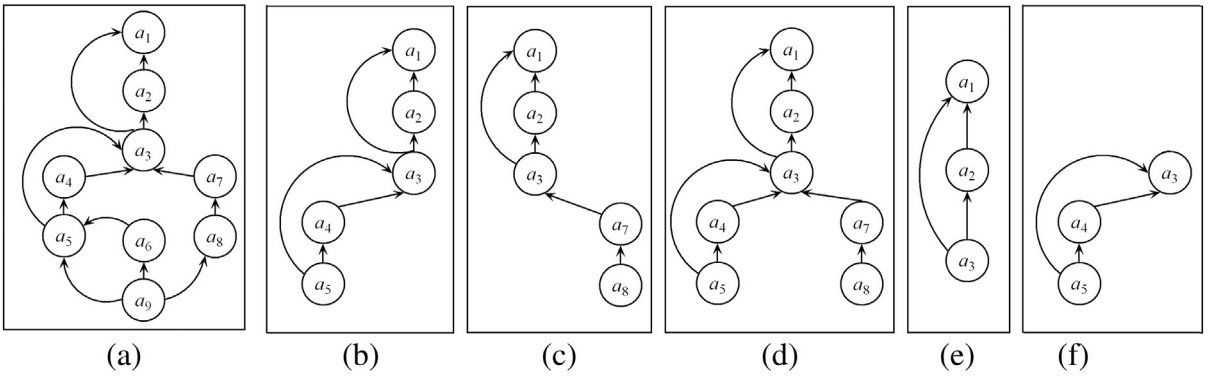


Fig. 9. The output produced by the operation of different operators: (a) an example provenance graph PG; (b) $\delta_{[P_d:D^{fwd}(a_5)]}(PG)$; (c) $\delta_{[P_d:D^{fwd}(a_8)]}(PG)$; (d) $\delta_{[P_d:D^{fwd}(a_5)]}(PG) \cup \delta_{[P_d:D^{fwd}(a_8)]}(PG)$; (e) $\delta_{[P_d:D^{fwd}(a_5)]}(PG) \cap \delta_{[P_d:D^{fwd}(a_8)]}(PG)$; and (f) $\delta_{[P_d:D^{fwd}(a_5)]}(PG) - \delta_{[P_d:D^{fwd}(a_8)]}(PG)$.

3.3.3. Intersection operator ($\cap$)

The intersection operator calculates the intersection of two subgraphs. An intersection operation is defined by $PG_1 \cap PG_2$, resulting in a provenance graph $PG' = (N', E')$, where

$$\begin{aligned} N' &= \{n \mid n \in N_1 \text{ and } n \in N_2\} \\ E' &= \{e \mid e \in E_1 \text{ and } e \in E_2\}. \end{aligned} \tag{11}$$

For example, you might want to find all artifacts that contributed to derive both artifact a_5 and artifact a_8 over provenance graph PG. Using the intersection operator, this query can be expressed as $\delta_{[P_d:D^{fwd}(a_5)]}(PG) \cap \delta_{[P_d:D^{fwd}(a_8)]}(PG)$. The result of the query is shown in Fig. 9(e).

3.3.4. Difference operator ($-$)

The difference operator calculates the difference of two subgraphs. A difference operation is defined by $PG_1 - PG_2$, resulting in a provenance graph $PG' = (N', E')$, where

$$\begin{aligned} E' &= \{e \mid e \in E_1 \text{ and } e \notin E_2\} \\ N' &= \{n \mid n \in \text{nodes}(E')\}. \end{aligned} \tag{12}$$

Here, *nodes* denotes a function returning a set of all nodes in E'. For example, you might want to find all artifacts that contributed to derive artifact a_5, but not artifact a_8 over provenance graph PG. Using the difference operator, this query can be expressed as $\delta_{[P_d:D^{fwd}(a_5)]}(PG) - \delta_{[P_d:D^{fwd}(a_8)]}(PG)$. The result of the query is shown in Fig. 9(f).

3.3.5. Example provenance queries expressed using the provenance graph algebra

To evaluate the feasibility of the operators defined in the provenance graph algebra, we use eight example provenance queries (see Table 1), which require the computation of transitive relationships to track ancestor nodes. These queries, including four queries (Q1–Q4) for the Load Workflow defined in the Third Provenance Challenge [9] and four queries (Q5–Q8) for a synthetic workflow consisting of a large number of steps, can be expressed using our provenance graph algebra (these queries can be also expressed in *OPQL* as shown later in Fig. 15). First, let PG_1 and PG_2 be the provenance graphs produced by the execution of the

Table 1
The example provenance queries expressed using the provenance graph algebra.

Q1:	For a given detection (detectID), which CSV files contributed to it? $\Rightarrow \delta_{[P_d:D^{fwd}('detectID')]}(PG_1) \cap \delta_{[P_b:value='\%CSV\%']}(PG_1)$
Q2:	Which steps were completed successfully before the halt occurred? $\Rightarrow \delta_{[P_g:G('\%success\%')]}(PG_1)$
Q3:	Why is this entry (ccdID) in the database? $\Rightarrow \delta_{[P_g:G('ccdID')]}(PG_1)$
Q4:	Which operation executions were necessary for the Image table to contain a particular value? $\Rightarrow \delta_{[P_g:G('\%Image\%')]}(PG_1)$
Q5:	Display dependencies of all the data products that contributed to derive the last data product (id $= a_n$). $\Rightarrow \delta_{[P_d:D^{fwd}(a_n)]}(PG_2)$
Q6:	Display dependencies of all the steps that were applied to trigger the last step (id $= p_n$). $\Rightarrow \delta_{[P_t:T^{fwd}(p_n)]}(PG_2)$
Q7:	Display dependencies of all the data products that were used by the last step (id $= p_n$). $\Rightarrow \delta_{[P_u:U(p_n)]}(PG_2)$
Q8:	Display dependencies of all the steps that contributed to generate the last data product (id $= a_n$). $\Rightarrow \delta_{[P_g:G(a_n)]}(PG_2)$

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

query :: = basic-query
| query UNION query
| query INTERSECT query
| query MINUS query
basic-query :: = single-node construct (arg)
| single-step-edge-forward construct (arg)
| single-step-edge-backward construct (arg)
| multi-step-edge construct (arg)
single-node construct :: = A | P | AG
single-step-edge-forward construct :: =
USD | WGB | WCB | WDF | WTB
single-step-edge-backward construct :: =
$USD^{\wedge} \mid WGB^{\wedge} \mid WCB^{\wedge} \mid WDF^{\wedge} \mid WTB^{\wedge}$
multi-step-edge construct :: =
USD* | WGB* | WDF* | WTB*
arg :: = basic-query | node-expression (X_n)
node-expression (X_n) :: = artifact-node-expression (X_a)
| process-node-expression (X_p)
| agent-node-expression (X_{ag})
artifact-node-expression (X_a) :: =
artifact-identifier (a_n) | %artifact-value% (a_v) | a^*
process-node-expression (X_p) :: =
process-identifier (p_n) | %process-value% (p_v) | p^*
agent-node-expression (X_{ag}) :: =
agent-identifier (ag_n) | %agent-value% (ag_v) | ag^*

(a)

$$\|a_n\| = \{a_n\}$$
$$\|p_n\| = \{p_n\}$$
$$\|ag_n\| = \{ag_n\}$$
$$\|a_v\| = \{a_n \mid a_n \in ids(a_v) \ \ and \ \ a_n \in A\}$$
$$\|p_v\| = \{p_n \mid p_n \in ids(p_v) \ \ and \ \ p_n \in P\}$$
$$\|ag_v\| = \{ag_n \mid ag_n \in ids(ag_v) \ \ and \ \ ag_n \in AG\}$$
$$\|a^*\| = \{a_n \mid a_n \in A\}$$
$$\|p^*\| = \{p_n \mid p_n \in P\}$$
$$\|ag^*\| = \{ag_n \mid ag_n \in AG\}$$

(b)

$$A(X_a) = \{a_n \mid a_n \in X_a\}$$
$$P(X_p) = \{p_n \mid p_n \in X_p\}$$
$$AG(X_{ag}) = \{ag_n \mid ag_n \in X_{ag}\}$$
$$USD(X_p) = \{a_n \mid p_n \in X_p \ \ and \ \ (p_n, a_n) \in E_u\}$$
$$WGB(X_a) = \{p_n \mid a_n \in X_a \ \ and \ \ (a_n, p_n) \in E_g\}$$
$$WCB(X_p) = \{ag_n \mid p_n \in X_p \ \ and \ \ (p_n, ag_n) \in E_c\}$$
$$WDF(X_a) = \{a_{n2} \mid a_{n_1} \in X_a \ \ and \ \ (a_{n_1}, a_{n_2}) \in E_d\}$$
$$WTB(X_p) = \{p_{n2} \mid p_{n_1} \in X_p \ \ and \ \ (p_{n_1}, p_{n_2}) \in E_t\}$$
$$USD^{\wedge}(X_a) = \{p_n \mid a_n \in X_a \ \ and \ \ (p_n, a_n) \in E_u\}$$
$$WGB^{\wedge}(X_p) = \{a_n \mid p_n \in X_p \ \ and \ \ (a_n, p_n) \in E_g\}$$
$$WCB^{\wedge}(X_{ag}) = \{p_n \mid ag_n \in X_{ag} \ \ and \ \ (p_n, ag_n) \in E_c\}$$
$$WDF^{\wedge}(X_a) = \{a_{n_1} \mid a_{n_2} \in X_a \ \ and \ \ (a_{n_1}, a_{n_2}) \in E_d\}$$
$$WTB^{\wedge}(X_p) = \{p_{n_1} \mid p_{n_2} \in X_p \ \ and \ \ (p_{n_1}, p_{n_2}) \in E_t\}$$

(c)

$$WDF^*(X_a) = \{a_n \mid \bigcup_{a_n \in WDF(X_a)} WDF^*(a_n) \ \cup \ WDF(X_a)\}$$
$$WTB^*(X_p) = \{p_n \mid \bigcup_{p_n \in WTB(X_p)} WTB^*(p_n) \ \cup \ WTB(X_p)\}$$
$$WGB^*(X_a) = \{p_n \mid \bigcup_{p_n \in WGB(X_a)} WTB^*(p_n) \ \cup \ WGB(X_a)\}$$
$$USD^*(X_p) = \{a_n \mid \bigcup_{a_n \in USD(X_p)} WDF^*(a_n) \ \cup \ USD(X_p)\}$$
$$WDF^{\wedge *}(X_a) = \{a_n \mid \bigcup_{a_n \in WDF^{\wedge}(X_a)} WDF^{\wedge *}(a_n) \ \cup \ WDF^{\wedge}(X_a)\}$$
$$WTB^{\wedge *}(X_p) = \{p_n \mid \bigcup_{p_n \in WTB^{\wedge}(X_p)} WTB^{\wedge *}(p_n) \ \cup \ WTB^{\wedge}(X_p)\}$$

(d)

Fig. 10. *OPQL* syntax and semantics: (a) the *OPQL* syntax; (b) the semantics of node expression X_n; (c) the semantics of the single-node constructs, single-step-edge-forward constructs, and single-step-edge-backward constructs; and (d) the semantics of the multi-step-edge constructs.

Load Workflow and synthetic workflow, respectively. Then, as depicted in Table 1, query Q1, which asks for CSV files that contributed to a given detection, can be answered by a query expressed as $\delta_{[P_d:D^{fwd}('detectID')]}(PG_1) \cap \delta_{[P_b:value='\%CSV\%']}(PG_1)$. It first finds all artifacts that contributed to derive the artifact with the value "detectID" by $\delta_{[P_d:D^{fwd}('detectID')]}(PG_1)$, and then it retains those artifacts whose values contain the CSV literal via the intersection with $\delta_{[P_b:value='\%CSV\%']}(PG_1)$. Similarly, query Q5 can be answered by $\delta_{[P_d:D^{fwd}(a_n)]}(PG_2)$. Second, query Q2, which asks for steps that were completed successfully before the halt occurred, can be answered by a query expressed as $\delta_{[P_g:G('\%success\%')]}(PG_1)$ to find all processes that contributed to generate artifacts with the value "%success%". In a similar fashion, the answers of queries Q3, Q4, and Q8 can be expressed as depicted in Table 1. Finally, query Q6, which asks for a process dependency view for all the steps that contributed to trigger the last step (id $= P_n$), can be satisfied by using $\delta_{[P_t:T^{fwd}(p_n)]}(PG_2)$ and query Q7, which asks for a data dependency view for all data products that were directly or indirectly used by the last step (id $= P_n$), can be satisfied by using $\delta_{[P_u:U(p_n)]}(PG_2)$.

3.4. OPQL syntax and semantics

We present an *OPQL* syntax that is required to formulate *OPQL* queries and a formal semantics for *OPQL* constructs. *OPQL* queries are formulated against a provenance graph displayed by a graphical user interface.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

3.4.1. OPQL syntax

OPQL queries are built from the syntax in Fig. 10(a). An *OPQL* query is composed of either a basic query or a set operation of two queries via set operators (i.e., UNION, INTERSECT, and MINUS). A basic query can be one of the single-node constructs (A, P, and AG), one of the single-step-edge-forward constructs (USD, WGB, WCB, WDF, and WTB), one of the single-step-edge-backward constructs (*USD^*, *WGB^*, *WCB^*, *WDF^*, and *WTB^*), or one of the multi-step-edge constructs (USD*, WGB*, WDF*, WTB*, WDF^*, and WTB^*). Each of these constructs has an argument (arg) that can be either a node expression (X_n) or a basic query. If a construct has a basic query as an argument, it means a nested *OPQL* query; otherwise, it means a simple *OPQL* query. A node expression (X_n) can be expressed by an artifact node expression (X_a), a process node expression (X_p), or an agent

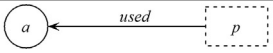
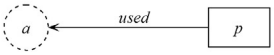
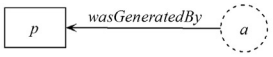
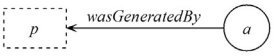
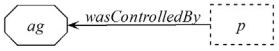
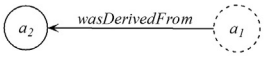
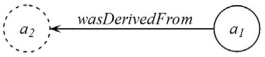
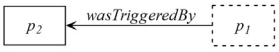
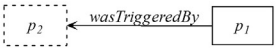
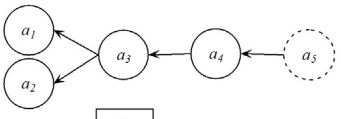
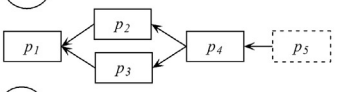
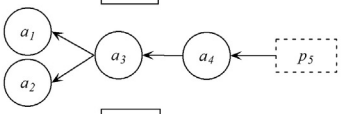
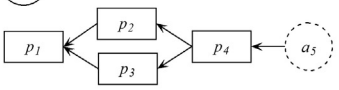
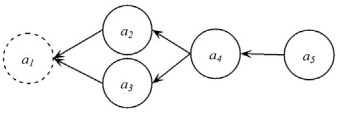
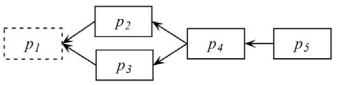
Construct	Description	Graphical Query Result
A (a)	Find artifacts satisfying artifact a	
P (p)	Find processes satisfying process p	
AG (ag)	Find agents satisfying agent ag	
USD (p)	Find artifacts that process p used	
USD^ (a)	Find processes that used artifact a	
WGB (a)	Find processes that generated artifact a	
WGB^ (p)	Find artifacts that process p generated	
WCB (p)	Find agents that controlled process p	
WCB^ (ag)	Find processes that agent ag controlled	
WDF (a_1)	Find artifacts that derived artifact a_1	
WDF^ (a_2)	Find artifacts that artifact a_2 derived	
WTB (p_1)	Find processes that triggered process p_1	
WTB^ (p_2)	Find processes that process p_2 triggered	
WDF* (a_5)	Find all the artifacts that were applied to derive artifact a_5	
WTB* (p_5)	Find all the processes that were applied to trigger process p_5	
USD* (p_5)	Find all the artifacts that process p_5 used directly or indirectly	
WGB* (a_5)	Find all the processes that were applied to generate artifact a_5	
WDF^* (a_1)	Find all the artifacts that were derived by artifact a_1	
WTB^* (p_1)	Find all the processes that were triggered by process p_1	

Fig. 11. The description of the *OPQL* constructs and the graphical query results.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

node expression (X_{ag}). The node expressions of an artifact, process, and agent can be a node identifier (i.e., a_n, p_n, *or* ag_n), a node value (i.e., a_v, p_v, *or* ag_v) starting and ending with %, or a wildcard (i.e., a^*, p^*, or ag^*), respectively.

3.4.2. OPQL semantics

Let $PG = (N, E)$ be a provenance graph such that $N = A \cup P \cup AG$ and $E = E_u \cup E_g \cup E_c \cup E_d \cup E_t$, as defined in Section 3.1. First, each node expression X_n (i.e., X_n $\{X_a, X_p, X_{ag}\}$) is defined as a function that maps a provenance graph (PG) to a set of nodes such that $X_n(PG)$ returns a subset of N as depicted in Fig. 10(b), where $ids(n_v)$ returns those nodes satisfying node value $n_v \in \{a_v, P_v, ag_v\}$.

We define the following three types of *OPQL* constructs including single-node constructs, single-step-edge-forward constructs, and single-step-edge-backward constructs. First, the single-node constructs play a role to retrieve nodes in a provenance graph, and they are defined as functions that take a provenance graph $PG = (N, E)$ and a node expression X_n and return those nodes satisfying node expression X_n such that $X_n(PG) \subseteq N$. Specifically, given single-node construct C_n (i.e., $C_n \in \{A, P, AG\}$), provenance graph $PG = (N, E)$, and node expression X_n, the semantics of the single-node constructs are defined by $C_n(X_n, PG) = \{n \mid n \in X_n (PG) \subseteq N\}$. For convenience, we generally omit PG when writing *OPQL* constructs and node expressions (as in Figs. 10(b), (c), (d), 11, and 15). Second, the single-step-edge-forward constructs and single-step-edge-backward constructs play a role to retrieve the cause node (the destination of an arc) and the effect node (the source of an arc) representing a causal dependency between two nodes in a provenance graph, respectively. The single-step-edge-forward constructs are defined as functions that take a provenance graph $PG = (N, E)$ and a node expression X_n for effect nodes and return cause nodes which have causal dependencies with effect nodes satisfying $X_n(PG)$, while the single-step-edge-backward constructs are defined as functions that take a provenance graph $PG = (N, E)$ and a node expression X_n for cause nodes and return effect nodes which have causal dependencies with cause nodes satisfying $X_n(PG)$. Specifically, given single-step-edge-forward construct C_e (i.e., $C_e \in \{USD, WGB, WCB, WDF, WTB\}$), single-step-edge-backward construct $C_{\hat{e}}$ (i.e., $C_{\hat{e}} \in \{USD^\wedge, WGB^\wedge, WCB^\wedge, WDF^\wedge, WTB^\wedge\}$), provenance graph $PG = (N, E)$, and node expression X_n, the semantics of the single-step-edge-forward constructs and single-step-edge-backward constructs are defined by $C_e(X_n, PG) = \{n^{cause} \mid n^{effect} \in X_n \text{ and } (n^{effect}, n^{cause}) \in E\}$ and $C_{\hat{e}}(X_n, PG) = \{n^{effect} \mid n^{cause} \in X_n \text{ and } (n^{effect}, n^{cause}) \in E\}$, respectively. More details on the semantics of these constructs are shown in Fig. 10(c).

```
Given provenance graph PG,
DG = WDF*(a6);
```

(a)

```
with graph Pb as A {
    node A.v1;
    node A.v2;
    }
    where A.e1(A.v1, A.v2).role = 'wasDerivedFrom'
    and A.v1.id = 'a6';
union all
    graph Pb as R {
        node R.v1;
        node R.v2;
    }
    where R.e1(R.v1, R.v2).role = 'wasDerivedFrom'
    and R.v1.id = A.v2.id;

DG = graph { };
for A in doc (PG)
let DG: = graph {
    graph DG;
    node A.v1, A.v2;
    edge A.e1(A.v1, A.v2);
    unify DG.v2, A.v1 where DG.v2.id = A.v1;
}
```

(b)

Fig. 12. Two different query expressions that generate a data dependency graph (DG) associated with artifact a_6 over the sample provenance graph (PG) presented in Fig. 1(b): (a) an *OPQL* query expression and (b) a GraphQL query expression.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

Next, we define the multi-step-edge constructs (i.e., *WDF**,*WTB*,**WGB**,*USD**,*WDF^**, and *WTB^**) as functions that take a provenance graph $PG = (N, E)$ and a node expression X_n and return all the nodes which have direct or indirect causal dependencies (i.e., transitive relationships) with those nodes satisfying $X_n(PG)$. These constructs allow user to track ancestor nodes without formulating recursive queries. For example, let Fig. 9(a) be a provenance graph (*PG*) as input. Then, multi-step-edge construct *WDF**(a_5) returns all artifacts that contributed to derive artifact a_5. That is, it returns a set of artifacts $\{a_1, a_2, a_3, a_4\}$ by the computation of transitive relationships associated with artifact a_5 via existing causal dependencies (i.e., $(\{(a_5, a_4),(a_5,a_3),(a_4,a_3),(a_3,a_2),(a_3,a_1),(a_2,a_1)\} \subset E_d)$). The semantics of these multi-step-edge constructs are depicted in Fig. 10(d).

A simple *OPQL* query is formulated by only an *OPQL* construct which has a node expression as an argument. Then, through graph pattern matching over the nodes returned by the computation of a single construct, a new provenance graph as output is extracted. In a similar way, a nested *OPQL* query is formulated by a combination of the *OPQL* constructs, and a new provenance graph as output is extracted via graph pattern matching over all the nodes that were returned by the computation of all the constructs in a nested *OPQL* query. To give a better understanding, we present simple *OPQL* query examples in Fig. 11, where the description of the *OPQL* constructs and the graphical query results are provided. In addition, an *OPQL* query can be expressed using a set operator between two *OPQL* queries since the result of a basic query is a provenance graph which consists of a set of nodes and a set of edges. For example, given two *OPQL* queries Q_1 and Q_2, a new *OPQL* query combining these two queries can be formulated using set operators UNION, INTERSECT, and MINUS (e.g., Q_1 UNION Q_2, Q_1 INTERSECT Q_2, and Q_1 MINUS Q_2). More details on the *OPQL* query expression are discussed in the following section.

3.5. Expressing provenance queries in OPQL

We discuss how provenance queries can be expressed in *OPQL*. As described in Section 3.4, an *OPQL* query is expressed as a combination of *OPQL* constructs, each of which corresponds to each of the graph patterns. The *OPQL* query language provides users with effective query formulation. For example, Fig. 12 shows two different query expressions that generate a data dependency graph (*DG*) for artifact a_6 over the provenance graph depicted in Fig. 1(b). First, Fig. 12(a) shows an *OPQL* query expression to answer the query via an *OPQL* construct, and then Fig. 12(b) shows a GraphQL query expression [19], which is expressed by a graph pattern and a FLWR (*For*, *Let*, *Where*, *Return*) expression in XQuery. Although the query expressed in GraphQL results in the same output as that of the *OPQL* query, the GraphQL query requires users to directly write a recursive query with a graph pattern; on the other hand, *OPQL* allows users to effectively formulate the query with just writing *WDF** (a_6). *OPQL* supports graph queries at a higher level than GraphQL. Therefore, *OPQL* is not tightly coupled to any particular storage strategy and, as we show in the following example, enables simpler provenance queries than queries expressed in SQL, XQuery, and SPARQL. Fig. 13 shows query CQ1, which asks for CSV files that contributed to a given detection (detectID) expressed in *OPQL*, SQL, XQuery, and in SPARQL. The four queries are meant for the same provenance graph. As shown in Fig. 13, query languages SQL, XQuery, and SPARQL are closely coupled to the physical storage strategies and their query expressions are more complex than *OPQL*. SQL query uses relation *MultiStepWasGeneratedBy* to find all process identifiers contributed to the generation of the artifact with the value 'detectID' and then it retrieves artifacts that are CSV files used by those processes. To accomplish the latter, SQL query uses the string '%Detection.csv' as a pattern that matches against any

CQ1: For a given detection (detectId), which CSV files contributed to it?

<OPQL>
USD(WGB*(%detectID%)) INTERSECT A(%Detection.csv)

<SQL>
```
SELECT DISTINCT A2.Value FROM Artifact A2, Used U,
(SELECT DISTINCT TG.ProcessId FROM MultiStepWasGeneratedBy TG, Artifact A
WHERE TG.ArtifactId = A.ArtifactId AND A.Value = 'detectID') As Pv
WHERE U.ProcessId = Pv.ProcessId AND U.ArtifactId = A2.ArtifactId AND A2.Value LIKE '%Detection.csv'
```

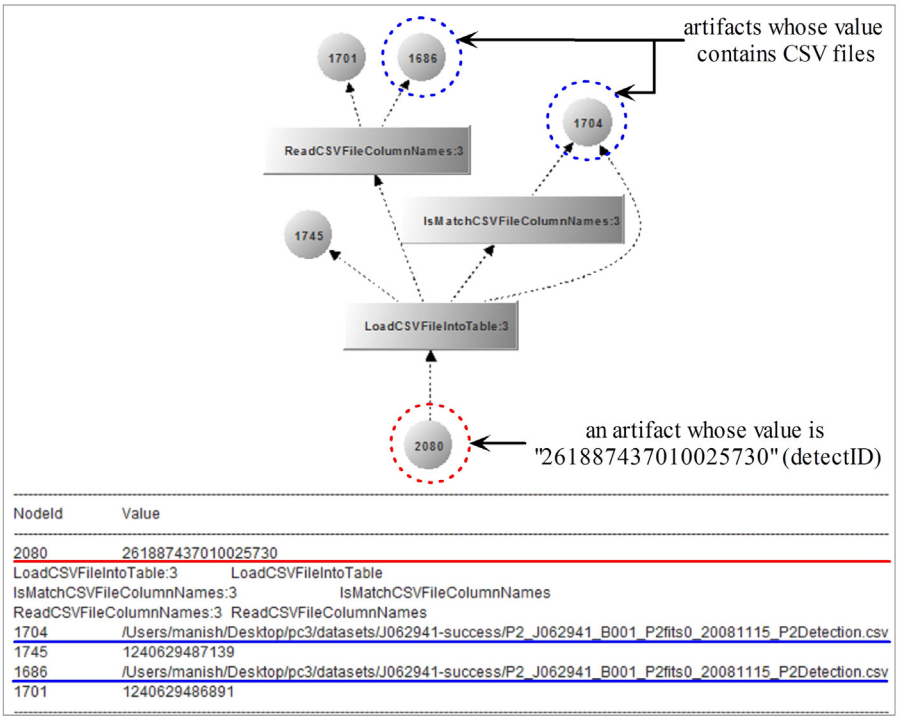
<XQuery>
```
let $d := doc('workflow_trace.xml');
let $a := $d//artifact[value/function/parameter/@val = 'Detection'];
return local:derivedFrom($d, $a)[ends-with(value/function/parameter/@val, 'Detection.csv')]
```

<SPARQL>
```
SELECT ?value
WHERE {?wgb PC3Prov:wgbSource pc3:DBEntryP2Detection_IterNumber3.
?wgb PC3Prov:wgbSource pc3:DBEntryP2Detection_IterNumber3. ?wgb PC3Prov:wgbTarget ?sxn.
?usd PC3Prov:usdSource ?sxn. ?usd PC3Prov:usdTarget ?var. ?var PC3Prov:hasType ?type.
FILTER(?type = "CSVFileEntry") ?var PC3Prov:hasValue ?value}
```

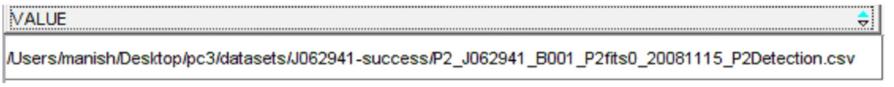
Fig. 13. A sample provenance query expressed by query languages *OPQL*, SQL, XQuery, and SPARQL, respectively.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

content of column *A2.Value* that ends with 'Detection.csv'. XQuery first reads an XML document (i.e., workflow_trace.xml), which contains workflow provenance data. Then it finds artifacts that have the detection value from the XML document and returns all the artifacts upstream containing a 'Detection.csv' file. In a similar way, SPARQL requires multiple triple patterns (resulting in multiple joins) with filtering 'CSVFileEntry' to find all the artifacts associated with CSV files. Thus, in these four queries 'Detection.csv' is not the artifact name, but rather a string used to find artifacts associated with CSV files. On the other hand, *OPQL* uses construct *WGB** to find all processes that contributed to generate an artifact whose value is 'detectID', and then it uses construct *USD* to find artifacts used by those processes, and then it intersects with construct *A*(*%Detection.csv*) to retrieve artifacts that are CSV files. As a result, in terms of usability, *OPQL* supports more effective query formulation than SQL, XQuery, and SPARQL. Furthermore, as depicted in Fig. 14, a query



(a)



(b)

```
<?xml version="1.0" encoding="UTF-8" ?>
- <artifact id="a77">
  - <value>
    - <function id="1" name="file">
        <parameter alias="" id="1" name="" type="edu.wayne.swf.test.basic:File"
          val="/swf_test/pc3/sampledata/J062945/P2_J062945_B001_P2fits0_20081115_P2Detection.csv" />
      </function>
    </value>
    <account id="acc01" />
  </artifact>
```

(c)

```
-------------------------------------------------------------------------------
 value
===============================================================================
./Data/J062941/P2_J062941_B001_P2fits0_20081115_P2Detection.csv-P2Detection
./Data/J062942/P2_J062942_B002_P2fits2_20081116_P2Detection.csv-P2Detection
-------------------------------------------------------------------------------
```

(d)

Fig. 14. Sample *OPQL*, SQL, XQuery, and SPARQL query results: (a) a provenance graph retrieved by an *OPQL* query; (b) a relation with a single attribute retrieved by an SQL query; (c) an XML document returned by an XQuery query; and (d) a set of variable bindings returned by a SPARQL query.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

result of an *OPQL* query is displayed as a provenance graph to give a better understanding to users while SQL returns a set of tuples (i.e., a table), XQuery returns a sequence of XML nodes or an XML document, and SPARQL returns a set of variable bindings.

In addition, to demonstrate the expressiveness of *OPQL*, we use 16 provenance queries, including three core queries (CQ) and 13 optional queries (OQ) defined in the Third Provenance Challenge [9]. In particular, we express some of these queries in *OPQL* that are executable over our OPMProv system. Fig. 15 (which is extended from a figure presented in [16]) shows 13 provenance queries in English, SQL, and *OPQL*, respectively (we omit the description of these queries and their answers. More details on these

CQ1:	For a given detection, which CSV files contributed to it?
SQL:	SELECT DISTINCT A2.Value FROM Artifact A2, Used U, (SELECT DISTINCT TG.OPMGraphId, TG.ProcessId FROM MultiStepWasGeneratedBy TG, Artifact A WHERE TG.OPMGraphId = A.OPMGraphId AND TG.ArtifactId = A.ArtifactId AND A.Value = '261887437010025730') As Pv WHERE U.OPMGraphId = Pv.OPMGraphId AND U.ProcessId = Pv.ProcessId AND U.OPMGraphId = A2.OPMGraphId AND U.ArtifactId = A2.ArtifactId AND A2.Value LIKE '%Detection.csv' AND U.OPMGraphId = '1'
OPQL:	**USD (WGB* (%261887437010025730%)) INTERSECT A (%Detection.csv%)**
CQ2:	The user considers a table to contain values they do not expect. Was the range check (IsMatchTableColumnRanges) performed for this table?
SQL:	SELECT 'YES' AS Answer FROM (SELECT COUNT(G.ArtifactId) AS Number FROM WasGeneratedBy G WHERE G.ProcessId LIKE '%IsMatchTableColumnRanges%' AND G.OPMGraphId = '1') AS Output WHERE Output.Number > 0
OPQL:	**WGB^ (%IsMatchTableColumnRanges%)**
CQ3:	Which operation executions were strictly necessary for the Image table to contain a particular (non-computed) value?
SQL:	SELECT Cp.Value As Operation, SUM(Cp.Number) As Count FROM (SELECT DISTINCT TT.OPMGraphId, TT.CauseProcessId, P.Value, 1 As Number FROM MultiStepWasTriggeredBy TT, Artifact A, Process P, WasGeneratedBy G WHERE TT.OPMGraphId = G.OPMGraphId AND G.OPMGraphId = A.OPMGraphId AND TT.EffectProcessId = G.ProcessId AND G.ArtifactId = A.ArtifactId AND A.Value LIKE '%Image%' AND TT.OPMGraphId = P.OPMGraphId AND TT.CauseProcessId = P.ProcessId) As Cp WHERE Cp.OPMGraphId = '1' GROUP BY Cp.Value
OPQL:	**WTB* (WGB (%Image%)) INTERSECT P (%LoadCSVFileIntoTable%)**
OQ1:	How many tables successfully loaded before the workflow halted due to a failed check?
SQL:	SELECT COUNT(*) AS Count FROM Artifact A, WasGeneratedBy G WHERE A.OPMGraphId = G.OPMGraphId AND A.ArtifactId = G.ArtifactId AND G.ProcessId LIKE 'IsMatchTableColumnRanges%' AND A.Value LIKE '%success%' AND A.OPMGraphId = '1'
OPQL:	**WGB^ (%IsMatchTableColumnRanges%) INTERSECT A (%success%)**
OQ4:	Why is this entry in the database?
SQL:	SELECT Pv.Value, SUM (Pv.Number) As Count FROM (SELECT DISTINCT TT.OPMGraphId, TT.CauseProcessId, P.Value, 1 As Number FROM MultiStepWasTriggeredBy TT, Artifact A, WasGeneratedBy G, Process P WHERE TT.OPMGraphId = G.OPMGraphId AND TT.EffectProcessId = G.ProcessId AND TT.OPMGraphId = P.OPMGraphId AND TT.CauseProcessId = P.ProcessId AND A.OPMGraphId = G.OPMGraphId AND A.ArtifactId = G.ArtifactId AND A.Value = ' 261887437010025730') AS Pv WHERE Pv.OPMGraphId = '1' GROUP BY Pv.Value
OPQL:	**WTB* (WGB (%261887437010025730%))**
OQ5:	A user executes the workflow many times (say 5 times) over different sets of data (j062941, ..., j062945). He wants to determine, which of the execution halted?
SQL:	SELECT A.OPMGraphId, A.Value AS NameOfDataset FROM Artifact A, WasGeneratedBy W WHERE A.OPMGraphId = W.OPMGraphId AND A.ArtifactId = W.ArtifactId AND A.Value LIKE 'J%halt%'
OPQL:	**WGB (%J%halt%)**
OQ6:	Determine the step where Halt occurred?
SQL:	SELECT Hp.Value As HaltStep, SUM(Hp.Number) As Count FROM (SELECT DISTINCT P.OPMGraphId, P.ProcessId, P.Value, 1 As Number FROM Artifact A, WasGeneratedBy G, Process P WHERE A.Value LIKE '%halt%' AND A.OPMGraphId = G.OPMGraphId AND A.ArtifactId = G.ArtifactId AND G.OPMGraphId = P.OPMGraphId AND G.ProcessId = P.ProcessId) As Hp WHERE Hp.OPMGraphId = '1' GROUP BY Hp.Value
OPQL:	**WGB (%halt%)**
OQ7:	Determine data and associated granularities of the data being processed, when halt occurred?
SQL:	SELECT DISTINCT A2.ArtifactId, A2.Value FROM Artifact A1, WasGeneratedBy G, Artifact A2, Used U WHERE A1.Value LIKE '%halt%occurred%' AND A1.OPMGraphId = G.OPMGraphId AND A1.ArtifactId = G.ArtifactId AND U.OPMGraphId = G.OPMGraphId AND U.ProcessId = G.ProcessId AND A2.OPMGraphId = U.OPMGraphId AND A2.ArtifactId = U.ArtifactId AND A2.OPMGraphId = '1'
OPQL:	**USD (WGB (%halt%occurred%))**
OQ8:	Which steps were completed successfully before the halt occurred?
SQL:	SELECT Sp.Value As Step, SUM (Sp.Number) As Count FROM (SELECT DISTINCT TW.OPMGraphId, TW.ProcessId, P.Value, 1 As Number FROM MultiStepWasGeneratedBy TW, Artifact A, Process P WHERE TW.OPMGraphId = A.OPMGraphId AND TW.ArtifactId = A.ArtifactId AND A.Value LIKE '%success%' AND TW.OPMGraphId = P.OPMGraphId AND TW.ProcessId = P.ProcessId) As Sp WHERE Sp.OPMGraphId = '1' GROUP BY Sp.Value
OPQL:	**WGB* (%success%)**
OQ10:	For a workflow execution, determine the user inputs?
SQL:	SELECT A.Value FROM Used U, Artifact A WHERE U.OPMGraphId = A.OPMGraphId AND U.ArtifactId = A.ArtifactId AND NOT EXISTS (SELECT * FROM WasGeneratedBy G WHERE G.OPMGraphId = A.OPMGraphId AND G.ArtifactId = A.ArtifactId) AND A.OPMGraphId = '1'
OPQL:	**A (a*) MINUS WGB^ (p*)**
OQ11:	For a workflow execution, determine steps that required user inputs?
SQL:	SELECT DISTINCT P.ProcessId, P.Value FROM Used U, Artifact A, Process P WHERE U.OPMGraphId = A.OPMGraphId AND U.ArtifactId = A.ArtifactId AND U.OPMGraphId = P.OPMGraphId AND U.ProcessId = P.ProcessId AND NOT EXISTS (SELECT * FROM WasGeneratedBy G WHERE G.OPMGraphId = A.OPMGraphId AND G.ArtifactId = A.ArtifactId) AND P.OPMGraphId = '1'
OPQL:	**USD^ (A (a*)) MINUS WGB^ (p*)**
OQ12:	For a workflow execution that halted, which files where processed successfully?
SQL:	SELECT DISTINCT A2.ArtifactId, A2.Value FROM WasGeneratedBy G, Artifact A1, Used U, Artifact A2 WHERE G.OPMGraphId = A1.OPMGraphId AND G.ArtifactId = A1.ArtifactId AND A1.Value LIKE '%success%' AND U.OPMGraphId = G.OPMGraphId AND U.ProcessId = G.ProcessId AND A2.OPMGraphId = U.OPMGraphId AND A2.ArtifactId = U.ArtifactId AND A2.Value LIKE '%CSVFileEntry%' AND A2.OPMGraphId = '1'
OPQL:	**USD (WGB (%success%)) INTERSECT A (%CSVFileEntry%)**
OQ13:	Display the following provenance views: data dependency view and step dependency view.
SQL:	SELECT EffectProcessId, CauseProcessId FROM MultiStepWasTriggeredBy WHERE OPMGraphId = '1'; SELECT EffectArtifactId, CauseArtifactId FROM MultiStepWasDerivedFrom WHERE OPMGraphId = '1';
OPQL:	**WDF* (a*); WTB* (p*)**

Fig. 15. Provenance queries expressed by *OPQL* for the Third Provenance Challenge questions.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

query processing can be found in [16]). As presented in Fig. 15, *OPQL* queries support simpler query expressions than SQL queries although query results are the same.

4. Implementation of OPQL

In this section, we discuss how our *OPQL* is implemented via OPMProv. As described in Section 2, OPMProv employs relational database technologies to manage and query scientific workflow provenance. As a result, the provenance store of OPMProv is implemented using RDBMS DB2 (v9.7.0.441). We implement *OPQL*, a graph-level provenance query language as a Web service using Java and Axis2. The *OPQL* Web service takes an *OPQL* query as input, translates an *OPQL* query to an equivalent SQL query and executes the SQL query translated in OPMProv, and returns a provenance graph as output. To invoke the *OPQL* Web service, we implement two kinds of user-friendly GUIs that allow users to formulate and execute *OPQL* queries over a provenance graph: one is OPMProVis^{D} (desktop version) implemented in Java and JGraph and the other is OPMProVis^{W} (web version) implemented in JSP and mxGraph [33]. As the need often arises to display provenance graphs or query results it is important to reduce response time for provenance querying and OPM graph reconstruction. To this end, we present an algorithm, called *GraphConstruct* to reconstruct an OPM provenance graph by translating provenance data stored in relational database into OPM-compliant XML serialization of the provenance graph. We present the details of this algorithm in Section 6. At the high level, the algorithm retrieves provenance graph constituents from the corresponding tables in OPMProv, creates nodes and edges for a provenance graph, and constructs a provenance graph. The algorithm is implemented in both OPMProVis^{D} and OPMProVis^{W} to visualize not only a whole provenance graph but also the result of an *OPQL* query. Fig. 16(a) shows the output of *OPQL* query $WDF^{*}(a_4)$ in OPMProVis^{D} and Fig. 16(b) shows the output of *OPQL* query $WTB^{*}(p_3)$ in OPMProVis^{W}.

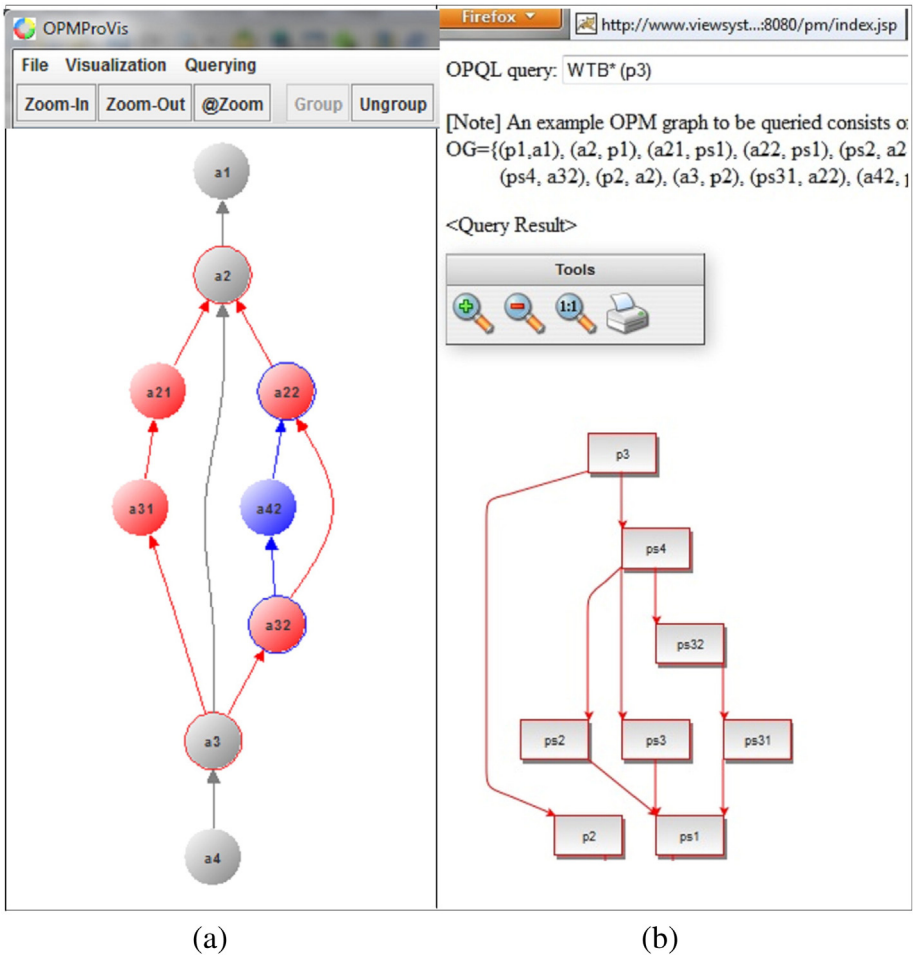


Fig. 16. Visualizing provenance graphs: (a) the output of *OPQL* query WDF^{*} (a_4) in OPMProVis^{D} and (b) the output of *OPQL* query WTB^{*} (p_3) in OPMProVis^{W}.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

5. Provenance graph reconstruction

Algorithm 1 shows an algorithm that retrieves an OPM graph constituents from OPMProv store and builds a provenance graph data structure *PG*. We illustrate the procedure for building a graph associated with one corresponding account as described in [15]. The account identifier *accId* is retrieved from the graphical user interface. The algorithm can be easily extended to handle multiple accounts. *GraphConstruct* algorithm contains four steps. First, *GraphConstruct* takes as input results obtained from a relational database storing OPM-compliant provenance data (lines 5–12), and creates sets R_a, R_p, R_{ag}, R_u, R_g, R_d, R_c, and R_t that correspond to three OPM nodes and five OPM edges, respectively. During the second step (lines 13–22) vertices are created that represent artifacts, processes and agents of an OPM graph (sets V_a, V_p, V_{ag} respectively). The third step (lines 23–38) in the algorithm creates edges of an OPM graph. In particular edges *Used*, *WasGeneratedBy*, *WasDerivedFrom*, *WasControlledBy*, and *WasTriggeredBy* are created by retrieving elements from the sets R_u, R_g, R_d, R_c, and R_t respectively. Finally, the fourth step (lines 39–44) builds a graph data structure *PG*, which is a tuple (N, E), where sets N and E represent vertices and edges of the OPM graph respectively. The obtained tuple *PG* is then returned as the final result of the algorithm.

6. Experimental study

The goal of our experimental study is to confirm the feasibility of using OPQL to query provenance graphs stored in a relational database. As we shall explain shortly, this can be achieved by verifying that the performance of SQL queries produced by OPMProv as well as the performance of our graph reconstruction algorithm is reasonable. The experiments presented below were conducted on a PC with one 2.27 GHz dual core processor and 4 GB main memory, running the Windows 7 operating system. In all the experiments, we show the results as the average of 20 trials.

Fig. 17 shows the process of provenance querying in OPMProv in detail. The querying lifecycle consists of five steps: 1) at the client side, obtain an OPQL query from the user via a GUI form and submit it to the server as a SOAP Web service request, 2) translate the OPQL query into an equivalent SQL query, 3) evaluate this SQL query against the relational provenance storage, 4) reconstruct the OPM graph, and send the generated provenance graph serialized as an XML document to the client as a Web service response, and 5) visualize the OPM-compliant XML document as a graph at the client side.

In the querying lifecycle, step one is trivial, and step five, which involves drawing the graph is accomplished by a third-party library JGraph [33]. Optimizing graph visualization performance as well as minimizing network overhead are distinct problems that are beyond the scope of this work.

Thus, from the perspective of querying provenance, which is the focus of this work, the crucial steps are two, three, and four. Of these three steps, the first one takes virtually no time. Indeed, translating OPQL to SQL involves manipulating a handful of objects that represent lexical constituents of a query, which is done momentarily in the main memory. Consequently, the most time consuming steps are (1) executing the SQL query and (2) reconstructing graph data structure from the obtained results. Therefore, to validate the feasibility of our approach, in our experimental study we show that OPMProv performs both of these steps in a reasonable time.

6.1. Provenance query performance

To ensure that our OPQL-to-SQL mapping returns queries with reasonable execution times, we evaluated the query performance of obtained SQL statements. To this end, we selected eight representative provenance queries and measured their execution time against a popular UCDGC (UC Davis Genome Center) dataset, used in the Third Provenance Challenge [9], which we placed in the storage of OPMProv (step three in Fig. 17). The UCDGC is a dataset that represents a provenance graph in which

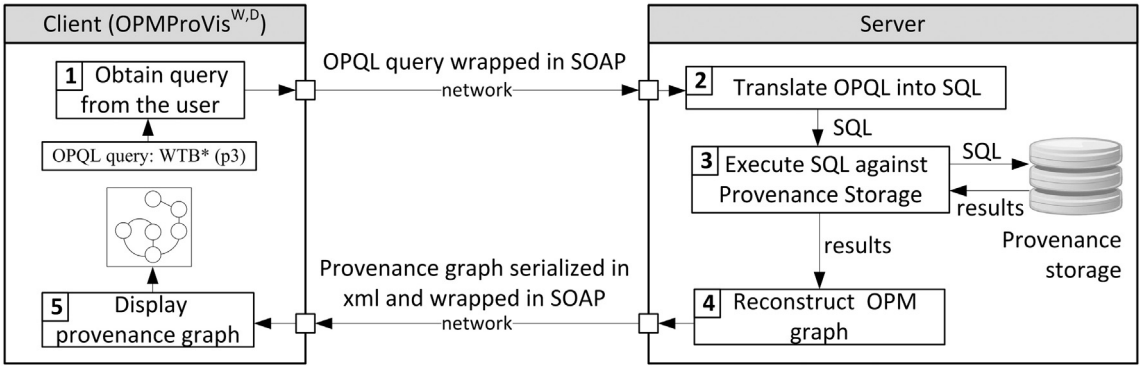


Fig. 17. Provenance querying lifecycle in OPMProv. Each step is labeled with a number in the top left corner.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

the total number of nodes and edges is 2909. Queries Q1–Q4 used in our experimental study are from the Third Provenance Challenge (CQ1, OQ8, OQ4, and CQ3 respectively). Queries Q5–Q8 were written for a synthetic workflow with large number of components.

Fig. 18(a) and (b) shows reasonable performance of our OPMProv implementation using off-the-shelf database system DB2. Given eight representative provenance queries from Table 1, our OPQL-to-SQL translation algorithm produced SQL queries with reasonable execution times of less than 0.06 s, as shown in Fig. 18(a).

Moreover, to explore the scalability of queries Q5, Q6, Q7, and Q8 that required more expensive computation of transitive relationships in the provenance graph, we used four provenance datasets generated via the simulation over four synthetic workflows. In these workflows, a workflow step was connected to only one other workflow step and the total number of steps (s) were 1000, 2000, 3000, and 4000, respectively. Note that the larger the number of steps in a workflow, the more expensive the computation of transitive relationships in its corresponding provenance graph. Queries Q5, Q6, Q7, and Q8 were evaluated on these larger datasets. The query evaluation times for these queries are reported in Fig. 18(b). Overall, these queries showed reasonable performance, returning results within around 12 s for the provenance dataset with 20,000 nodes and edges.

6.2. OPM graph reconstruction performance

To show that OPMProv reconstructs an OPM graph data structure in a reasonable time, we conducted experiments in which we selected five provenance datasets (that represent provenance graphs) generated by different participants of the Third Provenance Challenge [9], and inserted these datasets into OPMProv. We measured the time taken by our *GraphConstruct* algorithm to reconstruct each graph, that is to build the graph data structure from the relational database, and serialize it as an OPM compliant XML document. The graph reconstruction performance for these datasets is reported in Fig. 18(c), where the

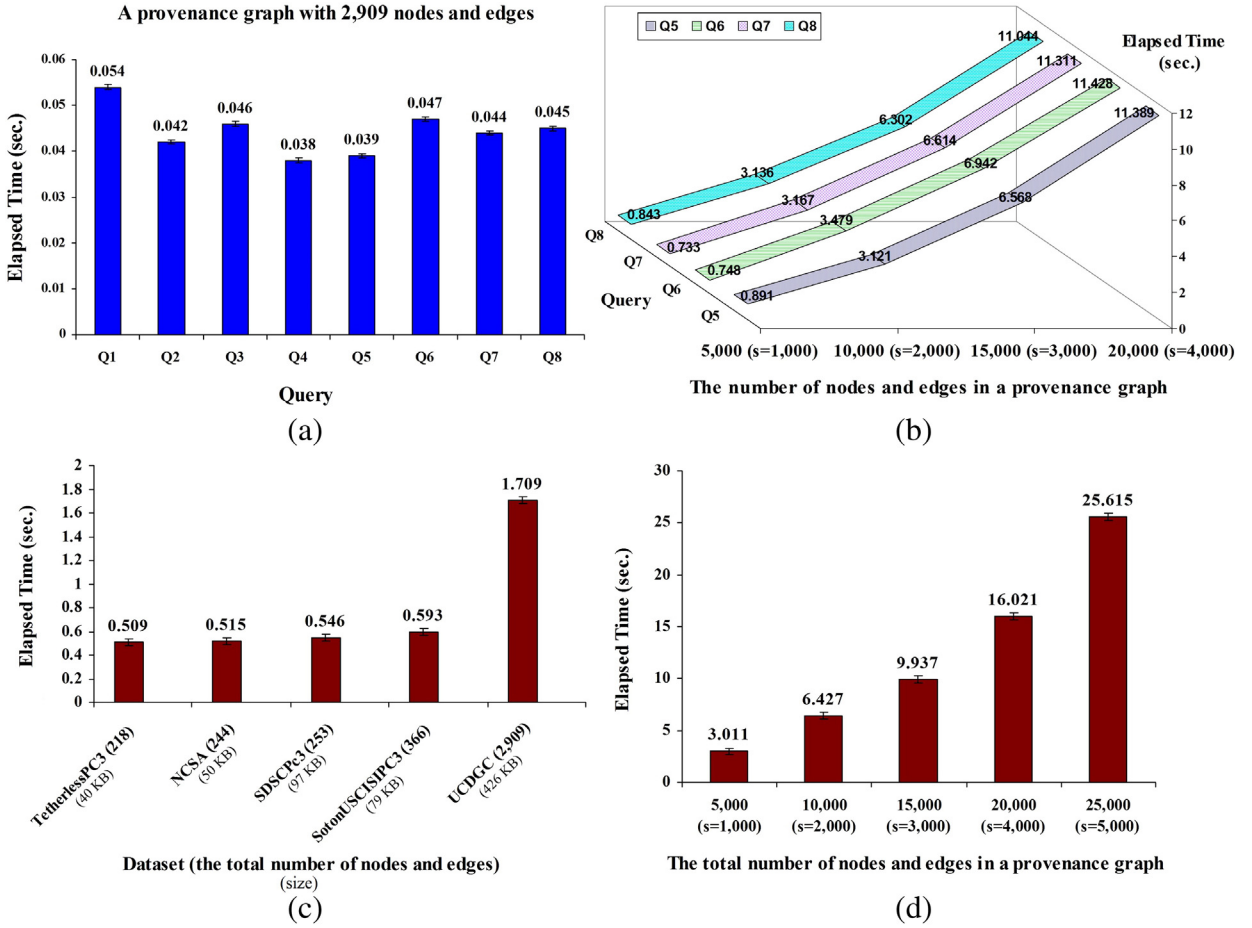


Fig. 18. Querying and graph reconstruction performance: (a) the query execution time when executed over the UCDGC dataset; (b) the query execution time over provenance graphs with varying complexity; (c) the OPM graph reconstruction time for different datasets; and (d) the OPM graph reconstruction time for provenance graphs with varying complexity.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

datasets are shown in the ascending order of the total number of nodes and edges. The obtained results show that OPMProv is able to reconstruct provenance graphs for each of the five datasets in less than 2 s (Fig. 18(c)).

To explore provenance graph reconstruction performance and scalability on larger datasets, we used five provenance datasets representing several provenance graphs with varying complexity. The complexity of each graph is measured in terms of the total number of nodes and edges in it. The higher the number is, the more complex the graph is. We vary complexity of a provenance graph by changing the total number of nodes and edges. Thus, in the five graphs we use in our study, the complexities vary as follows (from the simplest to the most complex dataset): 5000, 10,000, 15,000, 20,000, and 25,000, respectively. Thus, the last graph, which has a total 25,000 nodes and edges is the most complex of the five. The times taken by OPMProv to reconstruct each graph are reported in Fig. 18(d). Fig. 18 confirms that OPMProv shows reasonable performance when reconstructing graphs from the larger datasets, taking under 26 s to reconstruct the provenance graph with 25,000 nodes and edges.

7. Related work

In this section, we discuss related work on provenance query processing in existing systems. VisTrails [10] captures provenance of both the workflow evolution and associated data products by a change-based mechanism and visualizes the workflow evolution provenance as a version tree. VisTrails has the ability to visualize query results by highlighting workflow versions that match query conditions by using the VisTrails query language, called vtPQL. Kepler [11,12] implements an interactive provenance browser to visualize and query its proprietary workflow trace. In the provenance browser, users can create different views and express complex and recursive provenance queries by the Kepler's query language, called QLP. The QLP query language is defined on its proprietary provenance model, which explicitly supports workflow steps that process XML data and employ update semantics; thus, QLP allows users to use XML data structures and XPath expressions when provenance queries are formulated. A QLP query takes as input a workflow trace (T) defined as: $T = <V, F, L>$, where V is a set of vertices, which consists of a set of XML data structures and a set of invocation, F is a set of flow edges, and L is a set of lineage edges [11] and returns a set of lineage edges as output. In general, QLP can be useful in the situation where data is structured into nested collections like XML data and dependencies are defined among data nodes. ZOOM [20] enables users to construct appropriate user views for provenance graphs, and it provides users with an interface to query provenance information. Taverna [13] implements a semantic provenance infrastructure and visualizes semantic, RDF-based provenance graphs based on a provenance ontology. Taverna supports provenance queries using the SPARQL query language. Karma [14] presents an integrated provenance management architecture that supports automated data provenance collection, annotated provenance, and provenance visualization. Karma supports provenance queries in SQL and XPath. GraphQL [19] is a graph-based query language for graph databases. GraphQL is defined over a data model representing attributes of a generic graph, and a GraphQL query takes a collection of graphs as input and produces a collection of graphs using graph patterns. Like SQL, SPARQL, and XQuery, GraphQL requires users to directly formulate recursive queries to track ancestor nodes. Another graph query language is discussed in [31], in which authors propose Regular Path Queries variants that return provenance of graph queries. Other efforts to capture provenance of queries include querying semiring-annotated data [28], storing provenance of database queries [29], and provenance and evaluation of SPARQL queries on annotated RDF graphs [30]. However, the focus of these papers is distinct from our work, as the authors of [28-31] focus on capturing the *provenance of queries*, i.e. how the resulting tuples were obtained from the input data source, whereas we focus on *querying provenance*. In [34] authors propose algebraic operators as query language constructs to help users query provenance graph. Our work on the other hand, focuses on the query language *OPQL* including graph patterns and provenance graph algebra and presents its syntax and semantics. Finally, [32] discusses how data provenance can be used to empower integration process and does not focus on the querying aspect.

View [18] supports both prospective and retrospective provenance collection [17] and develops two independent systems, called RDFProv [1,21] and OPMProv [16], for provenance storage and querying. The OPMProv provenance model for the *OPQL* query language is developed based on our previous work on RDFProv and the Open Provenance Model (OPM) [15], which aims to define a standard provenance model to facilitate and promote provenance interoperability among heterogeneous systems. However, RDFProv uses the general semantic Web language, SPARQL, to query provenance, while OPM does not provide a provenance query language.

Recently, W3C Provenance Working Group [22] has been created to produce a set of documents that define data model, serialization formats and other definitions to provide a middle ground between various domain or software specific provenance models. As their effort is still ongoing, in the future we will ensure that our work complies with W3C provenance standards once they are completed.

Using *OPQL* together with OPMProv provides the following advantages for querying provenance over generic graph database language or other languages such as SQL or SPARQL: 1) *OPQL* is geared specifically towards provenance, featuring edges such as *WasGeneratedBy*, *WasDerivedFrom*, *WasTriggeredBy* and others, not available in other languages. Thus, *OPQL* offers conciseness and simplicity that neither generic languages nor SQL nor SPARQL can provide. For example, Fig. 13 presents the same query written in SQL, XQuery, SPARQL and *OPQL*. From the figure it is clear that *OPQL* is by far the easiest and the most natural of the four, 2) the fact that *OPQL* relies on the popular OPM provenance model minimizes the learning curve for the user. Indeed, as more and more scientists become familiar with OPM model, it is easier and more natural to use this knowledge to query provenance rather than to learn a new query language. 3) Unlike SQL, *OPQL* does not require the user to know the underlying storage schema. If the schema changes, the original *OPQL* query still produces the same result, whereas the SQL query needs be edited. 4) *OPQL* is technology-independent, and therefore can be integrated with any scientific workflow system. SQL on the other hand is only compatible with relational database technology which limits its use in workflow systems with non-relational storage. 5) OPMProv reconstructs provenance graph from the results obtained from executing an SQL query. Without OPMProv the user

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

would be forced to translate returned row sets into graphs himself, which is tedious. In summary, OPQL together with OPMProv free users from low-level details of provenance storage and querying a simple query language and intuitive results in the form of a graph. This paper provides *OPQL* to support the querying of scientific workflow provenance at the graph level, covering six types of graph patterns, a provenance graph algebra, *OPQL* syntax and semantics, implementation, and evaluation. This paper extends [23] with the following additional contributions:

1 We introduce five new mapping functions $(i.e., O_{\ddot{u}}, O_g, O_d, O_{\dot{t}}, O_{\dot{c}})$ for graph pattern P_o in Section 3.2. These mapping functions enable users to retrieve causal dependencies between nodes using our mphOPQL language.
2 We propose an *OPQL* syntax that is required to formulate *OPQL* queries and a formal semantics for *OPQL* constructs, resulting in a new Section 3.4. This provides a formal foundation for *OPQL*.
3 We present the architecture of our OPMProv system in the context of the scientific workflow management system.
4 We present 13 queries out of 16 queries defined in the Third Provenance Challenge [9] in *OPQL* to demonstrate the expressiveness of *OPQL* in Section 3.5. These queries expressed in *OPQL* are executable in the OPMProv system.
5 We implement the *OPQL* Web service to provide users with a provenance querying service for scientific workflows in Section 4. In addition, we implement user-friendly GUIs, such as OPMProVis^W (web version) and OPMProVis^D (desktop version), to invoke the *OPQL* Web service. This expands applicability of *OPQL*.

8. Conclusions and future work

In this paper, we designed the *OPQL* query language, including six types of graph patterns, a provenance graph algebra, and *OPQL* syntax and semantics, that enables querying of provenance at the graph level. We then implemented *OPQL* using a Web service via our OPMProv system; therefore, users can invoke the Web service to execute *OPQL* queries in user-friendly GUIs, such as OPMProVis^D and OPMProVis^W. Finally, we conducted experiments to evaluate the performance and feasibility of OPMProv on *OPQL* provenance querying, and the experimental results showed reasonable performance.

In the future we plan to continue our research with OPMProv in four major directions. First, we plan to benchmark the performance of OPMProv using tools such, as the University of Texas Provenance Benchmark [24], as well as to compare the querying performance of our system with other provenance querying systems. Second, while in this paper we focused on querying a single provenance graph, in the future we will explore querying multiple provenance graphs. For example, if several workflow runs used the same artifact as a workflow input, the need may arise to find all artifacts derived from this artifact across provenance graphs representing several workflow runs. Third, we plan to explore query optimization [27], in which among other things we will study the performance of all 14 queries from the Third Provenance Challenge. Finally, we plan to study provenance security [25], particularly compliance management. This may include, for example managing permissions to read, write and modify artifacts, as well as history of reads, writes and modifications.

While much research has been done on database usability, usability research in provenance querying is still in its infancy [26]. This paper takes one of the first steps to query provenance at the graph level. More study about usability, especially from an end user perspective is needed in the future.

Acknowledgments

The authors would like to thank Dr. Murali Mani [27,34] for providing valuable advices regarding OPMProv and for sharing his expertise in the area of provenance management.

Appendix A. Database schema

This appendix details our database schema that we use for storing, reasoning, and querying the OPM-compliant provenance data. Fig. A.1 presents the schema, which includes 29 relations, where the first twenty four of them are materialized relations and the remaining five are non-materialized views.

Each relation contains an OPM graph identifier (i.e., attribute *OPMGraphId*) that identifies multiple workflow runs of a certain workflow. We require every row in relations *Artifact*, *Process*, *Agent*, *Used*, etc. to have at least one account and thus at least one row in the corresponding *xxxHasAccount* relations. This participation requirement eliminates the burden of dealing with missing values when computing relational joins and can be met on the data insertion stage by introducing a default account.

The primary keys of these twenty four relations are also included in Fig. A.1. For example, the *ArtifactHasAccount* relation has (*OPMGraphId*, *ArtifactId*, *Account*) as the primary key and (*OPMGraphId*, *ArtifactId*) as the foreign key referencing the *Artifact* relation. Similarly, the *ArtifactAnnotation* relation has the primary key (*OPMGraphId*, *ArtifactId*, *Property*, *Value*) since according to OPM single property can have multiple values, and it has the foreign key (*OPMGraphId*, *ArtifactId*) referencing the *Artifact* relation. We also define a number of views in our schema. While the view *WasTriggeredBy* implements single-step inference (i.e., completion rule) defined in OPM [15], views *MultiStepWasDerivedFrom*, *MultiStepWasTriggeredBy*, *MultiStepUsed*, and *MultiStepWasGeneratedBy* implement multi-step inferences (i.e., multi-step versions of existing edges) presented in OPM [15].

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

```
1. Artifact (OPMGraphId, ArtifactId, Value)                                                    // key = {OPMGraphId, ArtifactId}
2. Process (OPMGraphId, ProcessId, Value)                                                      // key = {OPMGraphId, ProcessId}
3. Agent (OPMGraphId, AgentId, Value)                                                          // key = {OPMGraphId, AgentId}
4. Used (OPMGraphId, ProcessId, Role, ArtifactId, OTimeLower, OTimeUpper)                      // key = {OPMGraphId, ProcessId, ArtifactId, Role}
5. WasGeneratedBy (OPMGraphId, ArtifactId, Role, ProcessId, OTimeLower, OTimeUpper)            // key = {OPMGraphId, ArtifactId, ProcessId, Role}
6. WasControlledBy (OPMGraphId, ProcessId, Role, AgentId, OTimeStartLower, OTimeStartUpper,    // key = {OPMGraphId, ProcessId, AgentId, Role}
                    OTimeEndLower, OTimeEndUpper)
7. WasDerivedFrom (OPMGraphId, EffectArtifactId, CauseArtifactId, OTimeLower, OTimeUpper)      // key = {OPMGraphId, EffectArtifactId, CauseArtifactId}
8. ExplicitWasTriggeredBy (OPMGraphId, EffectProcessId, CauseProcessId, OTimeLower, OTimeUpper) // key = {OPMGraphId, EffectProcessId, CauseProcessId}
9. ArtifactHasAccount (OPMGraphId, ArtifactId, Account)                                        // key = {OPMGraphId, ArtifactId, Account}
10. ProcessHasAccount (OPMGraphId, ProcessId, Account)                                         // key = {OPMGraphId, ProcessId, Account}
11. AgentHasAccount (OPMGraphId, AgentId, Account)                                             // key = {OPMGraphId, AgentId, Account}
12. UsedHasAccount (OPMGraphId, ProcessId, Role, ArtifactId, Account)                          // key = {OPMGraphId, ProcessId, ArtifactId, Role, Account}
13. WasGeneratedByHasAccount (OPMGraphId, ArtifactId, Role, ProcessId, Account)                // key = {OPMGraphId, ArtifactId, ProcessId, Role, Account}
14. WasControlledByHasAccount (OPMGraphId, ProcessId, Role, AgentId, Account)                  // key = {OPMGraphId, ProcessId, AgentId, Role, Account
15. WasDerivedFromHasAccount (OPMGraphId, EffectArtifactId, CauseArtifactId, Account)          // key = {OPMGraphId, EffectArtifactId, CauseArtifactId, Account}
16. ExplicitWasTriggeredByHasAccount (OPMGraphId, EffectProcessId, CauseProcessId, Account)    // key = {OPMGraphId, EffectProcessId, CauseProcessId, Account}
17. ProcessAnnotation (OPMGraphId, ProcessId, Property, Value)                                 // key = {OPMGraphId, ProcessId, Property, Value}
18. ArtifactAnnotation (OPMGraphId, ArtifactId, Property, Value)                               // key = {OPMGraphId, ArtifactId, Property, Value}
19. AgentAnnotation (OPMGraphId, AgentId, Property, Value)                                     // key = {OPMGraphId, AgentId, Property, Value}
20. UsedAnnotation (OPMGraphId, ProcessId, Role, ArtifactId, Property, Value)                  // key = {OPMGraphId, ProcessId, Role, ArtifactId, Property, Value}
21. WasGeneratedByAnnotation (OPMGraphId, ArtifactId, Role, ProcessId, Property, Value)        // key = {OPMGraphId, ArtifactId, Role, ProcessId, Property, Value}
22. WasControlledByAnnotation (OPMGraphId, ProcessId, Role, AgentId, Property, Value)          // key = {OPMGraphId, ProcessId, Role, AgentId, Property, Value}
23. WasDerivedFromAnnotation (OPMGraphId, EffectArtifactId, CauseArtifactId, Property, Value)  // key = {OPMGraphId, EffectArtifactId, CauseArtifactId, Property, Value}
24. ExplicitWasTriggeredByAnnotation (OPMGraphId, EffectProcessId, CauseProcessId, Property, Value) // key = {OPMGraphId, EffectProcessId, CauseProcessId, Property, Value}
25. WasTriggeredBy (OPMGraphId, EffectProcessId, CauseProcessId, Account, OTimeLower, OTimeUpper)  // view
26. MultiStepWasDerivedFrom (OPMGraphId, EffectArtifactId, CauseArtifactId, Account)           // view
27. MultiStepWasTriggeredBy (OPMGraphId, EffectProcessId, CauseProcessId, Account)             // view
28. MultiStepUsed (OPMGraphId, ProcessId, ArtifactId, Account)                                 // view
29. MultiStepWasGeneratedBy (OPMGraphId, ArtifactId, ProcessId, Account)                       // view
```

Fig. A.1. Database schema.

References

[1] A. Chebotko, S. Lu, X. Fei, F. Fotouhi, RDFProv: a relational RDF store for querying and managing scientific workflow provenance, Data Knowl. Eng. 69 (8) (2010) 836–865.
[2] P. Missier, N.W. Paton, K. Belhajjame, Fine-grained and efficient lineage querying of collection-based workflow provenance, Proc. of the International Conference on Extending Database Technology (EDBT), 2010, pp. 299–310.
[3] M. Anand, S. Bowers, T. McPhillips, B. Ludäscher, Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs, Proc. of the International Conference on Scientific and Statistical Database Management (SSDBM), 2009, pp. 237–254.
[4] Z. Bao, S.C. Boulakia, S.B. Davidson, A. Eyal, S. Khanna, Differencing provenance in scientific workflows, Proc. of the IEEE International Conference on Data Engineering (ICDE), 2009, pp. 808–819.
[5] C. Silva, J. Freire, S. Callahan, Provenance for visualizations: reproducibility and beyond, IEEE Comput. Sci. Eng. 9 (5) (2007) 82–89.
[6] Y. Simmhan, B. Plale, D. Gannon, A survey of data provenance in e-Science, ACM SIGMOD Rec. 34 (3) (2005) 31–36.
[7] S. Sultana, E. Bertino, M. Shehab, A provenance based mechanism to identify malicious packet dropping adversaries in sensor networks, Proc. of the International Conference on Distributed Computing Systems Workshops (ICDCS), 2011, pp. 332–338.
[8] H. Lim, Y. Moon, E. Bertino, Provenance-based trustworthiness assessment in sensor networks, Proc. of the International Workshop on Data Management for Sensor Networks (DMSN), 2010, pp. 2–7.
[9] The Third Provenance Challenge home page, http://twiki.ipaw.info/Challenge/ThirdProvenanceChallenge/June 2009.
[10] C.E. Scheidegger, D. Koop, E. Santos, H.T. Vo, S.P. Callahan, J. Freire, C.T. Silva, Tackling the provenance challenge one layer at a time, Concurr. Comput. Pract. Exp. 20 (5) (2008) 473–483.
[11] M. Anand, S. Bowers, B. Ludäscher, Techniques for efficiently querying scientific workflow provenance graphs, Proc. of the International Conference on Extending Database Technology (EDBT), 2010, pp. 287–298.
[12] T. Stropp, T. McPhillips, B. Ludscher, M. Bieda, Workflows for microarray data processing in the Kepler environment, BMC Bioinforma. 13 (2012) 102.
[13] J. Zhao, C. Goble, R. Stevens, D. Turi, Mining Taverna's semantic web of provenance, Concurr. Comput. Pract. Exp. 20 (5) (2008) 463–472.
[14] Y. Simmhan, B. Plale, D. Gannon, Karma2: provenance management for data driven workflows, Int. J. Web Serv. Res. 5 (2) (2008) 1–22.
[15] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, J.V. Bussche, The Open Provenance Model core specification (v1.1), Futur. Gener. Comput. Syst. 27 (6) (2011) 743–756.
[16] C. Lim, S. Lu, A. Chebotko, F. Fotouhi, Storing, reasoning, and querying OPM-compliant scientific workflow provenance using relational databases, Futur. Gener. Comput. Syst. 27 (6) (2011) 781–789.
[17] C. Lim, S. Lu, A. Chebotko, F. Fotouhi, Prospective and retrospective provenance collection in scientific workflow environments, Proc. of the IEEE International Conference on Services Computing (SCC), 2010, pp. 449–456.
[18] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, J. Hua, A reference architecture for scientific workflow management systems and the VIEW SOA solution, IEEE Trans. Serv. Comput. 2 (1) (2009) 79–92.
[19] H. He, A.K. Singh, Graphs-at-a-time: query language and access methods for graph databases, Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2008, pp. 405–418.
[20] O. Biton, S.C. Boulakia, S.B. Davidson, C.S. Hara, Querying and managing provenance through user views in scientific workflows, Proc. of the IEEE International Conference on Data Engineering (ICDE), 2008, pp. 1072–1081.
[21] A. Chebotko, X. Fei, C. Lin, S. Lu, F. Fotouhi, Storing and querying scientific workflow provenance metadata using an RDBMS, Proc. of the IEEE eScience conference (eScience), 2007, pp. 611–618.
[22] The W3C Provenance Working Group home page, http://www.w3.org/2011/prov/.
[23] C. Lim, S. Lu, A. Chebotko, F. Fotouhi, OPQL: a first OPM-level query language for scientific workflow provenance, Proc. of the IEEE International Conference on Services Computing (SCC), 2011, pp. 136–143.
[24] A. Chebotko, E.D. Hoyos, C. Gomez, A. Kashlev, X. Lian, C. Reilly, UTPB: a benchmark for scientific workflow provenance storage and querying systems, Proc. of the IEEE International Workshop on Scientific Workflows (SWF), 2012, pp. 17–24.
[25] S. Xu, Q. Ni, E. Bertino, R.S. Sandhu, A characterization of the problem of secure provenance management, Proc. of the IEEE Intelligence and Security Informatics (ISI), 2009, pp. 310–314.
[26] H.V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, C. Yu, Making database systems usable, Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2007, pp. 13–24.
[27] H. Su, E.A. Rundensteiner, M. Mani, Semantic query optimization for XQuery over XML streams, Proc. of Very Large Data Bases (VLDB), 2005, pp. 277–288.
[28] G. Karvounarakis, T. Green, Semiring-annotated data: queries and provenance, ACM SIGMOD Rec. 44 (3) (2012) 5–14.
[29] Z. Bao, H. Köhler, L. Wang, X. Zhou, S.W. Sadiq, Efficient provenance storage for relational queries, Proc. of the ACM International Conference on Information and Knowledge Management (CIKM), 2012, pp. 1352–1361.
[30] F. Geerts, G. Karvounarakis, V. Christophides, I. Fundulaki, Algebraic structures for capturing the provenance of SPARQL queries, Proc. of the ACM International Conference on Database Theory (ICDT), 2013, pp. 153–164.
[31] S. Dey, V. Vicenttín, S. Köhler, E. Gribkoff, M. Wang, B. Ludäscher, On implementing provenance-aware regular path queries with relational query engines, Proc. of the Joint EDBT/ICDT Workshops, 2013, pp. 214–223.
[32] B. Tomazela, C.S. Hara, R.R. Ciferri, C.D.A. Ciferri, Empowering integration processes with data provenance, Data Knowl. Eng. 86 (2013) 102–123.
[33] JavaScript Graph Visualization and Layouts, http://www.jgraph.com/mxgraph.html(last accessed August 23 2013).
[34] M. Mani, M. Alawa, A. Kalyanasundaram, Query language constructs for provenance, Proc. of Symposium on International Database Engineering and Applications, 2011, pp. 254–255.

Chunhyeok Lim received the Ph.D. degree in computer science from Wayne State University in 2011, M.S. from Korea National Defense University in 2001, and B.S. from Korea Military Academy in 1996. He is currently an active officer in the Korean army. His research interests include scientific workflows and their provenance management.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008

Shiyong Lu received the Ph.D. degree in computer science from the State University of New York at Stony Brook in 2002, M.E. from the Institute of Computing Technology of Chinese Academy of Sciences at Beijing in 1996, and B.E. from the University of Science and Technology of China at Hefei in 1993. He is currently an Associate Professor in the Department of Computer Science, Wayne State University, and the director of the Scientific Workflow Research Laboratory (SWR Lab). His research interests include scientific workflows and their applications. He has published two books more than 100 papers in refereed international journals and conference proceedings. He is the founder and currently a program co-chair of the IEEE International Workshop on Scientific Workflows (2007–2011), an editorial board member for International Journal of Semantic Web and Information Systems and International Journal of Healthcare Information Systems and Informatics. He is a senior member of the IEEE.

Artem Chebotko received the Ph.D. degree in computer science from Wayne State University in 2008, M.S. and B.S. degrees in management information systems and computer science from Ukraine State Maritime Technical University in 2003 and 2001. He is currently an assistant professor in the Department of Computer Science, University of Texas-Pan American. His research interests include scientific workflow provenance data management, semantic web data management, and database systems. He has published more than 30 papers in refereed international journals and conference proceedings. He currently serves as a program committee member of several international conferences and workshops on scientific workflows, semantic web, and databases. He is a member of the IEEE.

Farshad Fotouhi received the Ph.D. degree in computer science from Michigan State University in 1988. In August 1988, he joined the faculty of Computer Science at Wayne State University, where he is currently a professor and the dean of the College of Engineering. His major areas of research include XML databases, semantic Web, multimedia systems, and biomedical informatics. He has published more than 100 papers in refereed journals and conference proceedings, served as a program committee member of various database-related conferences. He is on the Editorial Boards of the IEEE Multimedia Magazine and International Journal on Semantic Web and Information Systems. He is a member of the IEEE.

Andrey Kashlev is a PhD candidate at the Computer Science department of Wayne State University working under the supervision of Dr. Shiyong Lu. His research interests include scientific workflows, Big Data, databases, and services computing. He is a member of Big Data Research Laboratory as well as the IEEE.

Please cite this article as: C. Lim, et al., OPQL: Querying scientific workflow provenance at the graph level, Data Knowl. Eng. (2013), http://dx.doi.org/10.1016/j.datak.2013.08.008