

# OPQL: A First OPM-Level Query Language for Scientific Workflow Provenance

Chunhyeok Lim\*, Shiyong Lu\*, Artem Chebotko<sup>†</sup>, and Farshad Fotouhi\*  
\*Department of Computer Science, Wayne State University, Detroit, MI 48202

Email: {chlim, shiyong, fotouhi}@wayne.edu

<sup>†</sup>Department of Computer Science, University of Texas-Pan American, Edinburg, TX 78539

Email: artem@cs.panam.edu

**Abstract**—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 services computing and scientific workflows to validate, interpret, and analyze the result of scientific computing. Most existing systems store provenance data captured into their own provenance storages of proprietary provenance 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 provenance storage strategies. In this paper, we present *OPQL*, an OPM-level provenance query language, that is directly defined over the Open Provenance Model (OPM). An *OPQL* query takes an OPM graph as input and produces an OPM 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 graph patterns and an OPM-based graph algebra for *OPQL*, that efficiently supports provenance lineage queries; (ii) we implement *OPQL* in our OPMPROV system, where the result of *OPQL* queries is displayed as an OPM graph via the OPMPROV browser. An experimental study is conducted to evaluate the performance and feasibility of *OPQL* for provenance querying. To our best knowledge, *OPQL* is the first OPM-level query language for scientific workflow provenance.

**Keywords**-OPQL; OPM; Provenance; OPM-compliant provenance;

## I. 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 areas of services computing [1],[2],[3] and scientific workflows [4],[5],[6] to validate, interpret, and analyze the result of scientific computing.

Most existing systems [6],[10],[11],[13] store provenance data captured into their provenance storages of proprietary provenance models and conduct provenance querying using query languages such as SQL, SPARQL, and XQuery over the physical provenance storages (i.e., RDB, RDF, and XML). Such query languages are closely coupled to the underlying provenance storage strategies, and therefore users have to know structures or schemas of such provenance storages, as well as semantics of provenance models that have been applied to the provenance storages to formulate

provenance queries. Moreover, users require the expertise about grammars, syntax, and semantics of such languages to formulate complicated provenance queries. For example, using existing approaches, *provenance lineage queries* (queries for tracking ancestor nodes) often require users to write recursive queries (directly typing recursive statements or using recursive functionality), which are nontrivial.

To address these issues, in this paper, we propose *OPQL*, an OPM-level provenance query language that efficiently supports provenance lineage queries. *OPQL* is a graph query language that is directly defined over the Open Provenance Model (OPM) [8], which is a standard provenance model in the community. An *OPQL* query takes one OPM graph as input and produces an OPM graph as output. Therefore, *OPQL* queries are not tightly coupled to the underlying storage strategies. This paper has the following main contributions: (1) we design *OPQL*, an OPM-level provenance query language, that efficiently supports provenance lineage queries. To our best knowledge, *OPQL* is a first proposal on the OPM-level provenance query language for scientific workflows; (2) we implement *OPQL* in the OPMPROV system introduced in our prior work [14], where the result of *OPQL* queries is displayed as an OPM graph via the provenance browser of OPMPROV.

## II. OPMPROV OVERVIEW

OPMPROV is a relational database-based scientific workflow provenance system that stores, reasons, and queries OPM-compliant provenance data. Originally, the development of OPMPROV was motivated by the OPM model [8], which is a standard provenance model to facilitate and promote provenance interoperability among heterogeneous systems. In the community, there is an effort to support the OPM model in existing provenance systems. While most existing systems [6],[10],[11],[13] focus on enhancing existing infrastructure with the import or export capability by means of a mapping between their own proprietary provenance models and the OPM model, OPMPROV uses the OPM model as a conceptual data model to design a native OPM provenance store, and therefore an input or output of OPMPROV is OPM-compliant provenance data. OPMPROV is compliant with the OPM model (v1.1) [8]. Therefore, without any transformation between the OPM

model and our provenance model, provenance data represented in XML documents, which conform to the XML-schema specification for the OPM model, can be inserted into OPM PROV using a data mapping procedure that shreds the XML documents into relational tuples and stores them to the corresponding relational tables in OPM PROV. Moreover, OPM PROV can sufficiently support provenance reasoning (i.e., by completion rules and multi-step inferences) defined in the OPM model using recursive views and SQL queries alone without any additional reasoning engine. More details on the implementation and performance of OPM PROV can be found in [14].

### III. PROVENANCE QUERY LANGUAGE: OPQL

In this section, we describe the *OPQL* query language to efficiently support provenance lineage queries. We first propose the *OPQL* provenance model, graph patterns, and OPM-based graph algebra for *OPQL*, and then we discuss how provenance queries can be expressed by *OPQL*.

#### A. OPM and OPQL Provenance Model

The OPM model [8] is a standard provenance model in the community to facilitate and promote provenance interoperability among heterogeneous systems. In essence, the OPM model consists of a directed graph expressing the dependencies (i.e., how “things” depended on others and resulted in specific states). An OPM 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. An artifact is an immutable piece of state, a process is an action or a series of actions, and an agent is a contextual entity acting as a catalyst of a process. The five edges are described through the following sample provenance graph.

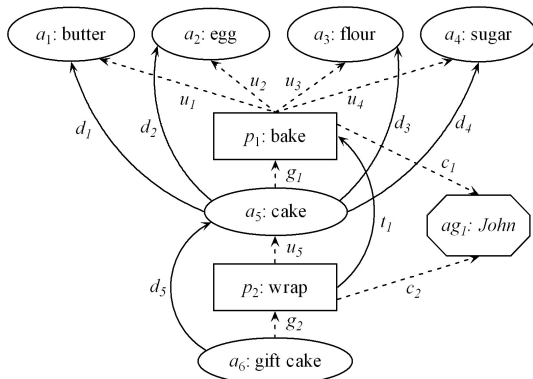


Figure 1: A sample OPM graph.

In Figure 1 (which is extended from a figure presented in [8]), an artifact, process, and agent 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). As depicted in Figure 1, the edges represent the following causal dependencies: (1) edge *Used* ( $u_1$ - $u_5$ ):  $p_1$ (bake) *used*  $a_1$ (butter),  $a_2$ (egg),  $a_3$ (flour), and  $a_4$ (sugar), and  $p_2$ (wrap) *used*  $a_5$ (cake); (2) edge *WasGeneratedBy* ( $g_1$ ,  $g_2$ ):  $a_5$ (cake) *was generated by*  $p_1$ (bake) and  $a_6$ (gift cake) *was generated by*  $p_2$ (wrap); (3) edge *WasDerivedFrom* ( $d_1$ - $d_5$ ):  $a_5$ (cake) *was derived from*  $a_1$ (butter),  $a_2$ (egg),  $a_3$ (flour), and  $a_4$ (sugar), and  $a_6$ (gift cake) *was derived from*  $a_5$ (cake); (4) edge *WasTriggeredBy* ( $t_1$ ):  $p_2$ (wrap) *was triggered by*  $p_1$ (bake); (5) edge *WasControlledBy* ( $c_1$ ,  $c_2$ ):  $p_1$ (bake) and  $p_2$ (wrap) *were controlled by*  $ag_1$ (John), respectively. More details on the constituents of the OPM graph can be found in [8].

Based on the OPM model, we define the *OPQL* provenance model. An *OPM graph*  $OG = (V, E)$  consists of:

- 1) a set of vertices  $V = A \cup P \cup AG$ , where  $A$  is a set of artifacts,  $P$  is a set of processes, and  $AG$  is a set of agents;
- 2) a set of 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  $p_2$ ,
  - v)  $E_c \subseteq P \times AG$  and  $(p, ag) \in E_c$  states that process  $p$  *was controlled by* agent  $ag$ .

In the *OPQL* provenance model, each node and edge can have arbitrary properties. We use a *tuple*, a list of name and value pairs, to denote these properties. Figure 2 shows a sample OPM graph that represents dependencies associated with process  $p_1$ (bake) in Figure 1.

```

graph OG {
  node v1<id='p1', value='bake'>;
  node v2<id='a1', value='butter'>;
  node v3<id='a2', value='egg'>;
  node v4<id='a3', value='flour'>;
  node v5<id='a4', value='sugar'>;
  edge e1(v1,v2)<id='u1', role='used'>;
  edge e2(v1,v3)<id='u2', role='used'>;
  edge e3(v1,v4)<id='u3', role='used'>;
  edge e4(v1,v5)<id='u4', role='used'>;
};

```

Figure 2: A sample OPM graph representing dependencies associated with process  $p_1$ (bake) in Figure 1.

#### B. Graph Patterns

We extend the notion of graph pattern proposed in [7] to efficiently support provenance lineage queries over the OPM 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. Figure 3 shows a sample graph pattern of  $P_b$ .

```

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

```

Figure 3: A sample graph pattern of  $P_b$ .

**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 efficiently handle five causal dependencies between nodes,  $O$  is composed of five types of mapping functions (i.e.,  $O \in \{O_u, O_g, O_d, O_t, O_c\}$ ) as defined below:

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

Graph pattern  $P_o$  is a derived graph pattern. It enables users to efficiently formulate complicated provenance queries. Figure 4 shows a sample graph pattern of  $P_o$ . In Figure 4, the former (graph pattern  $P_o$ ) is derived by the latter (graph pattern  $P_b$ ).

```

graph  $P_o$  {
  node  $v_1$ ;
  node  $v_2$ ;
}
mapping  $O_u : v_1 \xrightarrow{\text{used}} v_2$ 
where  $v_1.id = \text{'p1'}$ ;

```

(is derived by)

```

graph  $P_b$  {
  node  $v_1$ ;
  node  $v_2$ ;
}
where  $e_1(v_1, v_2).role = \text{'used'}$ 
and  $v_1.id = \text{'p1'}$ ;

```

Figure 4: A sample graph pattern of  $P_o$ .

Next, we define the following four graph patterns to efficiently 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.  $D$  is defined as:

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

**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.  $T$  is defined as:

$$T(p) = \bigcup_{p' \in O_t(p)} T(p') \cup O_t(p) \tag{3}$$

**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:

$$G(a) = \bigcup_{p' \in O_g(a)} T(p') \cup O_g(a) \tag{4}$$

**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:

$$U(p) = \bigcup_{a' \in O_u(p)} D(a') \cup O_u(p) \tag{5}$$

Each of graph patterns ( $P_d$ ,  $P_t$ ,  $P_g$ , and  $P_u$ ) is a derived graph pattern. These graph patterns enable users to efficiently formulate recursive queries to track ancestor nodes. For example, Figure 5 shows a sample graph pattern of  $P_d$ . In Figure 5, the former (graph pattern  $P_d$ ) is derived by the latter (graph pattern  $P_b$ ) via the recursive graph pattern of  $P_b$ , which is represented in the *with ~ union all* clause. In a similar fashion, the rest of sample graph patterns ( $P_t$ ,  $P_g$ , and  $P_u$ ) can be described.

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

**Definition 7** (Graph Pattern Matching  $\alpha$ ). A *graph pattern*  $P_b$  is matched with a graph  $OG$  if there exists an injective mapping  $\phi_\alpha: V(M) \rightarrow V(OG)$  such that i) For  $\forall e(u, v) \in E(M)$ ,  $(\phi_\alpha(u), \phi_\alpha(v))$  is an edge in  $OG$ , and ii) predicate  $C_{\phi_\alpha}(OG)$  holds.

**Definition 8** (Graph Pattern Matching  $\beta$ ). A *graph pattern*  $P_o$  is matched with a graph  $OG$  if there exists an injective mapping  $\phi_\beta: V(M) \rightarrow V(OG)$  such that i) For  $\forall e(u, v) \in E(M)$ ,  $(\phi_\beta(u), \phi_\beta(v))$  is an edge in  $OG$ , ii) function  $O_{\phi_\beta}(OG)$  holds, and iii) predicate  $C_{\phi_\beta}(OG)$  holds.

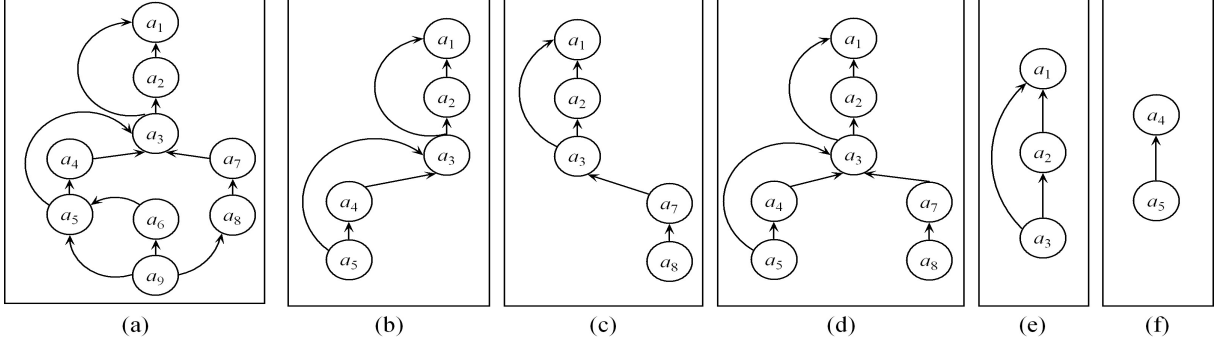


Figure 6: The output produced by the operation of different operators: (a) An example OPM graph  $OG$ ; (b)  $\delta_{[P_d:D(a_5)]}(OG)$ ; (c)  $\delta_{[P_d:D(a_8)]}(OG)$ ; (d)  $\delta_{[P_d:D(a_5)]}(OG) \cup \delta_{[P_d:D(a_8)]}(OG)$ ; (e)  $\delta_{[P_d:D(a_5)]}(OG) \cap \delta_{[P_d:D(a_8)]}(OG)$ ; and (f)  $\delta_{[P_d:D(a_5)]}(OG) - \delta_{[P_d:D(a_8)]}(OG)$ .

```

graph  $P_d$  {
  node  $v_1$ ;
  node  $v_2$ ;
}
mapping  $D : v_1 \xrightarrow{\text{derived}^*} v_2$ 
where  $v_1.\text{id} = 'a_6'$ ;

    (is derived by)

with graph  $P_b$  as  $A$  {
  node  $A.v_1$ ;
  node  $A.v_2$ ;
}
where  $A.e_1(A.v_1, A.v_2).\text{role} = \text{'derived'}$ 
and  $A.v_1.\text{id} = 'a_6'$ ;
union all
graph  $P_b$  as  $R$  {
  node  $R.v_1$ ;
  node  $R.v_2$ ;
}
where  $R.e_1(R.v_1, R.v_2).\text{role} = \text{'derived'}$ 
and  $R.v_1.\text{id} = A.v_2.\text{id}$ ;

```

Figure 5: A sample graph pattern of  $P_d$ .

**Definition 9** (Graph Pattern Matching  $\gamma$ ). Each of graph patterns ( $P_d$ ,  $P_t$ ,  $P_g$ , and  $P_u$ ) is matched with a graph  $OG$  if there exists an injective mapping  $\phi_\gamma: V(M) \rightarrow V(OG)$  such that i) For  $\forall e(u, v) \in E(M)$ ,  $(\phi_\gamma(u), \phi_\gamma(v))$  is an edge in  $OG$ , ii) each function ( $D_{\phi_\gamma}(OG)$ ,  $T_{\phi_\gamma}(OG)$ ,  $G_{\phi_\gamma}(OG)$ , and  $U_{\phi_\gamma}(OG)$ ) holds, and iii) each predicate  $C_{\phi_\gamma}(OG)$  holds.

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

**Definition 10** (Matched Graph). Given an injective mapping  $\phi \in \{\phi_\alpha, \phi_\beta, \phi_\gamma\}$  between a pattern  $P \in \{P_b, P_o, P_d, P_t, P_g, P_u\}$  and an OPM graph  $OG$ , a matched graph is a triple  $(\phi, P, OG)$  and is defined as  $\phi_P(OG)$ .

### C. OPM-Based Graph Algebra

We propose an OPM-based graph algebra for the *OPQL* query language. The OPM-based graph algebra is based on

four operators, which operate on the OPM graph. *OPQL* allows users to directly formulate an *OPQL* query against an OPM graph visualized by a graphical user interface. Therefore, each operator takes one OPM graph as input and produces another OPM graph as output. In particular, each of the union, intersection, and difference operators is basically operated on one OPM graph, but these operators take two OPM subgraphs produced by other queries as input and produce an OPM graph as output. We define the following four operators to manipulate and query an OPM graph.

1) *Extract operator* ( $\delta$ ): One of the most frequent operations performed on the OPM graph is the extraction of a set of nodes and edges, which are constituents of an OPM graph. An extract operator is defined using a graph pattern  $P$ . It takes one OPM graph ( $OG$ ) as input and produces a new OPM graph that matches the graph pattern as output, denoted by  $\delta_P(OG)$ . For example, let Figure 1 be an OPM graph ( $OG$ ). You might want to find all the artifacts that contributed to derive artifact  $a_6$ . Using the extract operator, this query can be expressed as:

$$\delta_{[P_d:D(a_6)]}(OG) \quad (6)$$

Actually, the 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*  $\gamma$  (i.e.,  $\phi_\gamma$ ), and then it produces a new OPM graph by combining information from the matched graph. The output of the extract operator is an OPM graph:

$$\delta_p(OG) = \phi_P(OG) \quad (7)$$

Next, we define the following three operators (*union*, *intersection*, and *difference*). These operators are basically operated on one OPM graph, but they take two OPM subgraphs produced by other queries as input and produce an OPM graph as output. Let  $OG$  be an OPM graph, and let  $OG_1$  and  $OG_2$  be the output of  $\delta_{P_1}(OG)$  and  $\delta_{P_2}(OG)$ , respectively. Given two OPM subgraphs  $OG_1 = (V_1, E_1)$  and  $OG_2 = (V_2, E_2)$ , where  $OG_1$  and  $OG_2 \subseteq OG$ , these operators are defined as follows.

2) *Union operator* ( $\cup$ ): The union operator calculates the union of two OPM subgraphs. A union operation is defined by  $OG_1 \cup OG_2$ , resulting in an OPM graph  $OG' = (V', E')$ , where

$$\begin{aligned} V' &= \{v \mid v \in V_1 \text{ or } v \in V_2\} \\ E' &= \{e \mid e \in E_1 \text{ or } e \in E_2\} \end{aligned} \quad (8)$$

For example, let Figure 6(a) be an OPM graph ( $OG$ ). Then, Figure 6(b) and Figure 6(c) represent the output of  $\delta_{[P_d:D(a_5)]}(OG)$  and  $\delta_{[P_d:D(a_8)]}(OG)$ , respectively. You might want to find all the artifacts that contributed to derive either artifact  $a_5$  or artifact  $a_8$  over OPM graph  $OG$ . Using the union operator, this query can be expressed as  $\delta_{[P_d:D(a_5)]}(OG) \cup \delta_{[P_d:D(a_8)]}(OG)$ . The result of the query is shown in Figure 6(d).

3) *Intersection operator* ( $\cap$ ): The intersection operator calculates the intersection of two OPM subgraphs. An intersection operation is defined by  $OG_1 \cap OG_2$ , resulting in an OPM graph  $OG' = (V', E')$ , where

$$\begin{aligned} V' &= \{v \mid v \in V_1 \text{ and } v \in V_2\} \\ E' &= \{e \mid e \in E_1 \text{ and } e \in E_2\} \end{aligned} \quad (9)$$

For example, you might want to find all the artifacts that contributed to derive both artifact  $a_5$  and artifact  $a_8$  over OPM graph  $OG$ . Using the intersection operator, this query can be expressed as  $\delta_{[P_d:D(a_5)]}(OG) \cap \delta_{[P_d:D(a_8)]}(OG)$ . The result of the query is shown in Figure 6(e).

4) *Difference operator* ( $-$ ): The difference operator calculates the difference of two OPM subgraphs. A difference operation is defined by  $OG_1 - OG_2$ , resulting in an OPM graph  $OG' = (V', E')$ , where

$$\begin{aligned} V' &= \{v \mid v \in V_1 \text{ and } v \notin V_2\} \\ E' &= \{e \mid e \in E_1 \text{ and } e \notin E_2\} \end{aligned} \quad (10)$$

For example, you might want to find all the artifacts that contributed to derive artifact  $a_5$ , but not artifact  $a_3$  over OPM graph  $OG$ . Using the difference operator, this query can be expressed as  $\delta_{[P_d:D(a_5)]}(OG) - \delta_{[P_d:D(a_3)]}(OG)$ . The result of the query is shown in Figure 6(f).

#### D. Expressing Provenance Queries in OPQL

We discuss how provenance queries can be expressed by *OPQL*. An *OPQL* query is expressed by a combination of *OPQL* constructs, each of which corresponds to each of graph patterns. These constructs are implemented by a graphical user interface. In this paper, we skip the description of *OPQL* constructs since the graph patterns have been defined. We just introduce an example about how an *OPQL* construct is expressed and operated over an OPM graph. Let  $WDF^*$  be an *OPQL* construct corresponding to graph pattern  $P_d$ . Figure 7 shows two different query expressions that generate a data dependency graph ( $DG$ ) regarding artifact  $a_6$  over the OPM graph depicted in Figure 1. First, Figure 7(a)

shows an *OPQL* query expression to answer the query via an *OPQL* construct, and then Figure 7(b) shows a GraphQL query expression [7], 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.

Given OPM graph  $OG$ ,

$DG = WDF^*(a_6)$ ; (a)

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

```
DG = graph { };
for A in doc (OG)
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)
```

Figure 7: Two different query expressions that generate a data dependency graph ( $DG$ ) regarding artifact  $a_6$  over the OPM graph ( $OG$ ) presented in Figure 1: (a) An *OPQL* query expression and (b) A GraphQL query expression.

In addition, to evaluate the feasibility of *OPQL* operators, we use eight provenance queries, 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 by our OPM-based graph algebra (these queries can be also expressed in *OPQL*). First, let  $OG_1$  and  $OG_2$  be the OPM graphs produced by the execution of the Load Workflow and synthetic workflow, respectively. Then, as depicted in Table I, query Q1 can be answered by a query expressed as  $\delta_{[P_d:D('detectID')]}(OG_1) \cap \delta_{[P_b:value='CSV%']}(OG_1)$ . It first finds all the artifacts that contributed to derive the artifact with the value “detectID” by  $\delta_{[P_d:D('detectID')]}(OG_1)$ ,

Q1:	For a given detection (detectID), which CSV files contributed to it? $\Rightarrow \delta_{[P_d:D('detectID')]}(OG_1) \cap \delta_{[P_b:value='CSV%']}(OG_1)$
Q2:	Which steps were completed successfully before the halt occurred? $\Rightarrow \delta_{[P_g:G('%success%')]}(OG_1)$
Q3:	Why is this entry (ccdID) in the database? $\Rightarrow \delta_{[P_g:G('ccdID')]}(OG_1)$
Q4:	Which operation execution were necessary for the Image table to contain a particular value? $\Rightarrow \delta_{[P_g:G('%Image%')]}(OG_1)$
Q5:	Display dependencies of all the data products that contributed to derive the last data product (id = $a_n$ ). $\Rightarrow \delta_{[P_d:D(a_n)]}(OG_2)$
Q6:	Display dependencies of all the steps that were applied to trigger the last step (id = $p_n$ ). $\Rightarrow \delta_{[P_t:T(p_n)]}(OG_2)$
Q7:	Display dependencies of all the data products that were used by the last step (id = $p_n$ ). $\Rightarrow \delta_{[P_u:U(p_n)]}(OG_2)$
Q8:	Display dependencies of all the steps that contributed to generate the last data product (id = $a_n$ ). $\Rightarrow \delta_{[P_g:G(a_n)]}(OG_2)$

Table I: The provenance queries expressed by *OPQL* operators.

and then, it retrieves these artifacts whose value is CSV files via the intersection with  $\delta_{[P_b:value='CSV%']}(OG_1)$ . Similarly, query Q5 can be answered by  $\delta_{[P_d:D(a_n)]}(OG_2)$ . Second, query Q2 can be answered by a query expressed as  $\delta_{[P_g:G('%success%')]}(OG_1)$  to find all the 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 I. 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  $\delta_{[P_t:T(p_n)]}(OG_2)$  and query Q7, which asks for a data dependency view for all the data products that were directly or indirectly used by the last step (id =  $p_n$ ), can be satisfied by using  $\delta_{[P_u:U(p_n)]}(OG_2)$ .

#### IV. IMPLEMENTATION OF OPQL

In this section, we discuss how our *OPQL* is implemented in the OPMPROV system. OPMPROV stores an OPM graph into associated tables using a data mapping procedure presented in [14]. To support *OPQL* queries over OPM graphs stored in OPMPROV, we implement an application using Java and JGraph that translates an *OPQL* query to an equivalent SQL query and executes the SQL query translated in OPMPROV. The result of an *OPQL* query is displayed as an OPM graph via a provenance browser. In OPMPROV, efficient provenance visualization, either as part of visualizing a whole OPM graph or as part of visualizing the query result is important to reduce response time for provenance querying and visualization. In this work, we employ an efficient algorithm, called *OPM-GraphConstruct* (we omit its details here), to construct an OPM graph from OPMPROV. The algorithm retrieves OPM graph entities from the corresponding tables in OPMPROV, creates vertices and edges for an OPM graph, and constructs an OPM graph. Based on the algorithm, we implement a provenance browser that visualizes not only a whole OPM graph but also the result of an *OPQL* query. Figure 8(a) shows an example OPM graph displayed via the provenance browser supported by OPMPROV and Figure 8(b) shows the output of  $\delta_{[P_d:D(a_4)]}(OG)$  over the OPM graph (*OG*) in the provenance browser.

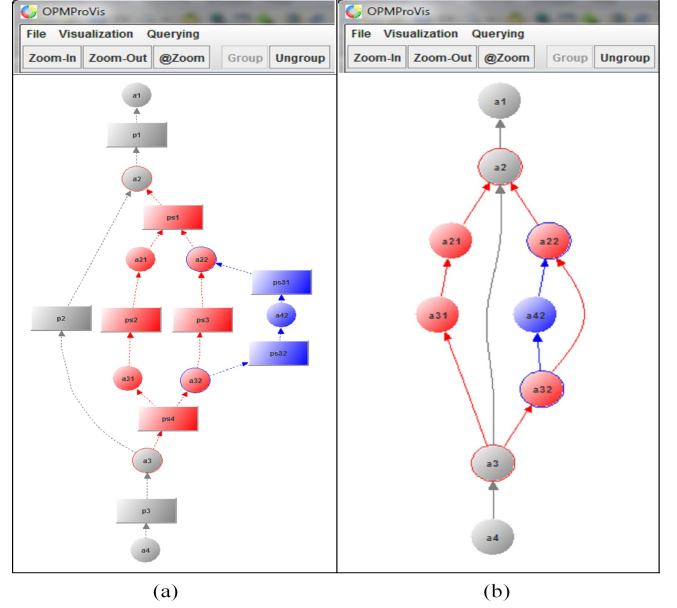


Figure 8: (a) An example OPM graph (*OG*) displayed via the provenance browser of OPMPROV and (b) The output of  $\delta_{[P_d:D(a_4)]}(OG)$  over the OPM graph (*OG*).

#### V. EXPERIMENTAL STUDY

In this section, we report on our experimental study that explored the query performance of *OPQL* over various datasets. We first performed provenance query experiments for *OPQL*, and then we performed provenance visualization performance experiments to demonstrate the visualization capability of OPMPROV. 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.

##### A. Provenance Query Performance Experiments

To evaluate the provenance query performance of *OPQL*, we used eight provenance queries depicted in Table I. These queries were executed on the dataset (UCDGC: UC Davis Genome Center), which represents an OPM graph in which the total number of nodes and edges is 2,909. The query performance experiments are reported in Figure 9(a). Overall, the query evaluation for *OPQL* showed to be very

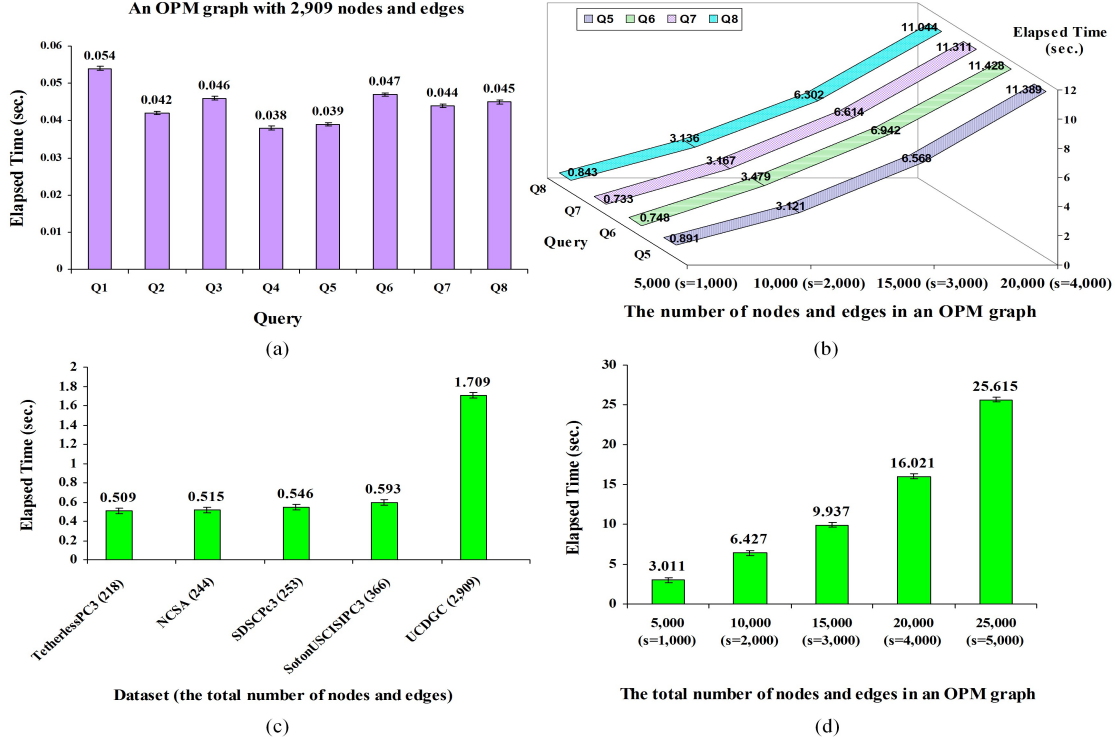


Figure 9: (a) The query performance on the UCDGC dataset; (b) The query performance on the OPM graphs with varying complexity; (c) The provenance visualization performance across different datasets; and (d) The provenance visualization performance on the OPM graphs with varying complexity.

efficient, returning results within 0.06 seconds for all the queries (Q1-Q8).

Moreover, to explore the scalability of queries Q5, Q6, Q7, and Q8 that required the more expensive computation of transitive relationships in the OPM graph, we used four OPM-compliant datasets generated via the simulation over four synthetic workflows, which are a sequential type of workflows (i.e., a workflow step is connected to only one workflow step) in which the total numbers of steps ( $s$ ) are 1,000, 2,000, 3,000, and 4,000, respectively. Note that the more the number of steps of the workflow, the more expensive the computation of transitive relationships in the OPM graph. Queries Q5, Q6, Q7, and Q8 were evaluated on these larger datasets. The response times for these queries are reported in Figure 9(b). Overall, these queries showed satisfactory performance, returning results within around 12 seconds for the provenance dataset with 20,000 nodes and edges.

### B. Provenance Visualization Performance Experiments

To perform provenance visualization experiments, we selected five OPM-compliant provenance datasets (which represent the OPM graphs) generated by different participants of the Third Provenance Challenge [9], and then we inserted these datasets into OPM PROV. The provenance visualization performance for these datasets is reported in Figure 9(c),

where the datasets are shown in the ascending order of the total number of nodes and edges. The results for provenance visualization showed to visualize all the datasets in less than 2 seconds.

To explore the provenance visualization performance and scalability on larger datasets, we used five OPM-compliant provenance datasets which represent the OPM graphs with varying complexity in which the total numbers of nodes and edges are 5,000, 10,000, 15,000, 20,000, and 25,000, respectively. The results for these datasets are reported in Figure 9(d). Overall, the provenance visualization performance over the larger datasets showed satisfactory performance, returning results within around 26 seconds for the dataset with 25,000 nodes and edges.

## VI. RELATED WORK

In this section, we discuss related work on provenance query processing and visualization in existing provenance 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] implements an interactive provenance browser to visualize and query data

dependency graphs. The provenance browser enables users to create different views for provenance graphs and express complex and recursive graph queries. Similar to *OPQL*, the Kepler's QLP query language provides a separation between the logical provenance model and its underlying physical representation. However, QLP is not directly defined over the OPM model, but on Kepler's proprietary provenance model. Thus, QLP has no support for direct query processing of OPM graphs. ZOOM [12] 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 [6] presents an integrated provenance management architecture that supports automated data provenance collection, annotated provenance, and provenance visualization. The Karma's provenance browser visualizes OPM graphs by a mapping between provenance events and OPM entities. Karma supports provenance queries in SQL and XPath. GraphQL [7] 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. GraphQL is a semi-structured query language, and therefore like SQL, SPARQL, and XQuery, GraphQL requires users to directly formulate recursive queries to support provenance lineage queries.

Although most existing systems have the capabilities to query and visualize provenance data in their systems, query languages supported by these systems are closely coupled to the underlying provenance storage strategies. Moreover, these systems query OPM graphs by means of a mapping between their proprietary provenance model and the OPM model. *OPQL*, on the other hand, is directly defined over the OPM model; therefore, *OPQL* is not tightly coupled to the underlying provenance storage strategies. Our *OPQL* features the native support for query processing of OPM graphs. That is, an *OPQL* query takes one OPM graph as input and produces an OPM graph as output. *OPQL* might be a cornerstone for a study on OPM-level provenance query languages.

## VII. CONCLUSIONS AND FUTURE WORK

In this paper, we designed and implemented *OPQL*, an OPM-level provenance query language, that efficiently support provenance lineage queries. To evaluate the performance and feasibility of *OPQL*, we conducted experiments on provenance querying and visualization, and the experimental results showed satisfactory performance. For a future work, we will evaluate *OPQL* from other points of view, including expressiveness, completeness, and usability.

## REFERENCES

- [1] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, Service Provenance in QoS-Aware Web Service Runtimes, In Proc. of the IEEE International Conference on Web Services (ICWS), pages 115-122, 2009.
- [2] Y. Simmhan, B. Plale, and D. Gannon, A framework for collecting provenance in data-centric scientific workflows, In Proc. of the IEEE International Conference on Web Services (ICWS), pages 427-436, 2006.
- [3] W. Ding, J. Wang, and Y. Han, ViPen: A Model Supporting Knowledge Provenance for Exploratory Service Composition, In Proc. of the IEEE International Conference on Services Computing (SCC), pages 265-272, 2010.
- [4] A. Chebotko, S. Lu, S. Chang, F. Fotouhi, and P. Yang, Secure Abstraction Views for Scientific Workflow Provenance Querying, IEEE Transactions on Services Computing (TSC), 3(4):322-337, 2010.
- [5] S. Miles, P. T. Groth, M. Branco, and L. Moreau, The Requirements of Using Provenance in e-Science Experiments, Journal of Grid Computing (GRID), 5(1):1-25, 2007.
- [6] Y. Simmhan, B. Plale, and D. Gannon, Karma2: Provenance Management for Data Driven Workflows, International Journal of Web Services Research (IJWSR), 5(2):1-22, 2008.
- [7] H. He and A. K. Singh, Graphs-at-a-time: query language and access methods for graph databases, In Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 405-418, 2008.
- [8] 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, and J. V. Bussche, The Open Provenance Model core specification (v1.1), Future Generation Computer Systems (FGCS), 27(6):743-756, 2011.
- [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, and C. T. Silva, Tackling the Provenance Challenge one layer at a time, Concurrency and Computation: Practice and Experience, 20(5):473-483, 2008.
- [11] M. Anand, S. Bowers, and B. Ludäscher, Techniques for efficiently querying scientific workflow provenance graphs, In Proc. of the International Conference on Extending Database Technology (EDBT), pages 287-298, 2010.
- [12] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara, Querying and Managing Provenance through User Views in Scientific Workflows, In Proc. of the IEEE International Conference on Data Engineering (ICDE), pages 1072-1081, 2008.
- [13] J. Zhao, C. Goble, R. Stevens, and D. Turi, Mining Taverna's semantic web of provenance, Concurrency and Computation: Practice and Experience, 20(5):463-472, 2008.
- [14] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi, Storing, reasoning, and querying OPM-compliant scientific workflow provenance using relational databases, Future Generation Computer Systems (FGCS), 27(6):781-789, 2011.