Modeling and Querying Collaborative Provenance

“In the long history of humankind (and animal kind, too) those who learned to collaborate and improvise most effectively have prevailed.”

– Charles Darwin

This chapter describes a relational schema for implementing the model presented in the previous chapter.

6.1 Collaborative Provenance Schema

Figure 6.1 shows the basic model we use for storing the various relationships that exist among users, datasets, workflows, and workflow runs. Data artifacts are assumed to be globally identified via the Data class, where data artifacts directly published by a user are represented in UserData and artifacts generated as a result of a workflow run are represented in RunData. Thus, all artifacts represented by the Data class represent either UserData or RunData. As shown, data artifacts can depend on other data artifacts; and similarly runs can depend on other runs. Specifically, in Figure 6.1 models that:

- a User performs one or more Runs, publishes (w-publishes) one or more workflows, and publishes (d-publishes) one or more data,
- a Run executes a Workflow (where as a workflow can be executed by one or more runs), produces one or more RunData, uses one or more Data, and depends (r-depends) on one or more other runs, and
- a Data depends (d-depends) on one or more other data (and gets used by one or more runs).

This chapter is based on Altintas et al. 2010d) and (Altintas et al. 2010f) co-authored by Altintas.
Figure 6.1: UML-based model for representing collaborative entities and their relationships (key attributes are underlined).

We consider the following relational schema based on Figure 6.1.

- **user**(*u, n*) denotes that *u* is a user with name *n*.

- **workflow**(*w, u*) denotes that *w* was a workflow published by user *u*.

- **run**(*r, w, u*) denotes that *r* was a run of workflow *w* and was executed by user *u* (i.e., *u* performed *r*).

- **data**(*d, v, t*) denotes that artifact identifier *d* had the data value *v* and type *t*, where *t* is either *rdata* (RunData) or *udata* (UserData).

- **publishes**(*d, u*) denotes that artifact id *d* was published by user *u*. If publishes(*d, u*) then we require that *d* is user data, i.e., that there is a value *v* such that data(*d, v, udata*) is true.

- **uses**(*r, d*) denotes that artifact *d* was input to run *r*. Note that *d* can be either user or run data.

- **produces**(*r, d*) denotes that artifact *d* was output by run *r*. If produces(*r, d*) then we require that *d* is run data, i.e., that there is a value *v* such that data(*d, v, rdata*) is true.

- **ddep**(*d_{from}, d_{to}*) denotes a dependency between output data *d_{from}* and input data *d_{to}*. 

- **ddep**(*d_{to}, d_{from}*) denotes the transitive closure of the ddep relation.
Note that above we represent the \textit{w-publishes} relationship of Figure 6.1 between a User and a Workflow using the \textit{workflow} relation. Similarly, we represent the \textit{performs} relationship of Figure 6.1 between a User and a Run using the \textit{run} relation. The \textit{w-publishes} relationship between User and Workflow, and \textit{d-publishes} relationship between User and Data are defined as one to many, but can be extended to many to many without a significant impact to the model in the case of co-authored workflows and data. Potential extensions of the model for co-authorship of workflows and data is discussed as a part of future work.

The schema captures the explicit dependency between outputs and inputs of a run (denoted \textit{d-depends on} in Figure 6.1) using the \textit{ddep} relation. (Most workflow engines capture this as provenance information.) The \textit{r-depends on} relationship is not defined as a base table since it can be explicitly queried using \textit{ddep}. We show below (see section 6.3.2) how run dependencies can be computed from \textit{ddep}. Typically, this information can be inferred by performing a transitive closure of dependency relations between inputs and outputs of each invocation (i.e., process execution) for a given run. We also perform pre-processing steps to compute the transitive closure of data dependencies and store the result in the \textit{ddep*} relation. This pre-computed transitive closure relation allows faster query execution, though it has expensive storage overhead. The exact storage overhead depends on how the workflow and the data dependencies are structured, and can go up to \(O(n^2)\). Although it does not impact the model discussed in this thesis, we discuss storage optimization as a part of future work. We intend to use the reduction techniques discussed in (Anand et al. 2009b) for storing both \textit{ddep} and \textit{ddep*} in reduced form.

### 6.2 Motivating Use-case Schema

To explain an instance of the described collaborative provenance data model, we will use the example scenario shown in Figure 6.2. Note that this scenario is larger than the scenario that motivated the definition of collaborative provenance (see Figure 5.1) and involves more complicated data dependency relationships between workflow runs.

An instance of this schema is shown in Table 6.1, which corresponds to the example of Figure 6.2. The same information is also illustrated as relational tables in Figure 6.4. As mentioned before, here we assume that the data identifiers that are internal to the workflow execution has already been mapped to global data identifiers once the data resulting from workflow runs are published.

### 6.3 Generating Collaborative Provenance Views

The schema described above can be used to express a number of different provenance views (see Figure 6.3(b), Figure 6.5 and Figure 6.7) using standard relational query languages.
Figure 6.2: An example scenario for a typical scientific research project: (a) data \( \{d_1, d_2, d_3\} \) published by users in \( \{u_1, u_6\} \); (b) ready to run workflows \( \{wf_1 \ldots wf_5\} \) published by users in \( \{u_2, u_4, u_5\} \); and (c) flow graph for published workflow runs (customized through their parameters) and related data \( \{d_1 \ldots d_{10}\} \) in user spaces \( \{u_1, u_2, u_3\} \), separated by horizontal lines.

### 6.3.1 Data Dependency View

We can directly retrieve the data dependency view (DATA-DEP), e.g., shown in Figure 6.3(b), using the ddep relation.

\[
\text{DATA-DEP}(d_{\text{from}}, d_{\text{to}}) : \neg \text{ddep}(d_{\text{from}}, d_{\text{to}}).
\]
6.3 Generating Collaborative Provenance Views

Figure 6.3: (a) Combined workflow run graph that shows the flow of data through different workflow runs, and (b) the complete data dependency view, based on the scenario in Figure 6.2.

6.3.2 Run Dependency View

We can retrieve the run dependency view (RUN-DEP), e.g., as shown in Figure 6.5, by performing a join between the ddep and produces relations.
publishes := \{ (d_1, u_6), (d_2, u_3), (d_3, u_6) \}
workflow := \{ (w_1, u_4), (w_2, u_2), (w_3, u_2), (w_4, u_4), (w_5, u_5) \}
run := \{ (r_1, w_1, u_1), (r_2, w_2, u_2), (r_3, w_3, u_3), (r_4, w_4, u_1), (r_5, w_3, u_2), (r_6, w_5, u_3) \}
uses := \{ (r_1, d_1), (r_2, d_2), (r_2, d_3), (r_3, d_6), (r_4, d_4), (r_4, d_5), (r_5, d_5), (r_5, d_7), (r_6, d_7) \}
produces := \{ (r_1, d_4), (r_2, d_5), (r_2, d_6), (r_3, d_7), (r_4, d_8), (r_5, d_9), (r_6, d_10) \}
ddep := \{ (d_4, d_1), (d_5, d_2), (d_6, d_3), (d_7, d_6), (d_8, d_4), (d_8, d_5), (d_9, d_5), (d_9, d_7), (d_{10}, d_7) \}
ddep* := ddep \cup \{ (d_7, d_3), (d_8, d_1), (d_9, d_2), (d_9, d_6), (d_9, d_3), (d_{10}, d_6), (d_{10}, d_3) \}

Table 6.1: Relation instances of the provenance schema corresponding to the example in Figure 6.2.

<table>
<thead>
<tr>
<th>user</th>
<th>workflow</th>
<th>data</th>
<th>publishes</th>
<th>uses</th>
<th>ddep</th>
<th>ddep*</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>id</td>
<td>value</td>
<td>type</td>
<td>id</td>
<td>ddata</td>
</tr>
<tr>
<td>u_1</td>
<td>w_1</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_1</td>
<td>u_1</td>
</tr>
<tr>
<td>u_2</td>
<td>w_1</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_2</td>
<td>u_1</td>
</tr>
<tr>
<td>u_3</td>
<td>name_3</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_3</td>
<td>u_1</td>
</tr>
<tr>
<td>u_4</td>
<td>name_3</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_4</td>
<td>u_1</td>
</tr>
<tr>
<td>u_5</td>
<td>name_3</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_5</td>
<td>u_1</td>
</tr>
<tr>
<td>u_6</td>
<td>name_3</td>
<td>d_1</td>
<td>v_1</td>
<td>udata</td>
<td>d_6</td>
<td>u_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d_7</td>
<td>v_1</td>
<td>rdata</td>
<td>d_7</td>
<td>u_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d_8</td>
<td>v_2</td>
<td>rdata</td>
<td>d_8</td>
<td>u_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d_9</td>
<td>v_3</td>
<td>rdata</td>
<td>d_9</td>
<td>u_3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d_{10}</td>
<td>v_4</td>
<td>rdata</td>
<td>d_{10}</td>
<td>u_4</td>
</tr>
</tbody>
</table>

Figure 6.4: Relational tables of the provenance schema corresponding to the example in Figure 6.2.

\[
\text{RUN-DEP}(r_{\text{from}}, r_{\text{to}}) := \text{ddep}(d_{\text{from}}, d_{\text{to}}), \text{produces}(r_{\text{from}}, d_{\text{from}}), \text{produces}(r_{\text{to}}, d_{\text{to}}).
\]

An illustration of this join corresponding the motivating scenario in Figure 6.2 is provided in Figure 6.6.
6.3 Generating Collaborative Provenance Views

Figure 6.5: Run dependency view based on the scenario in Figure 6.2.

Figure 6.6: RUN-DEP view can be generated by joining ddep and produces tables as shown.

6.3.3 User Collaboration View

Figure 6.7 shows a simple user collaboration graph that corresponds to the scenario in Figure 6.2. However, it doesn’t show the attributes for user collaborations. As described in Figure 5.3, user collaborations can be due to a published workflow (C–WF), published data by users (C–Data), or due to run data being used as inputs (C–Run). In this section, we present a set of queries to create user collaboration views ranging from least informed to most informed as shows in Figure 5.5(a). The following query can be used to generate user collaboration view, C(u_from, e, u_to), where e denotes the nature (WF, Data, Run) of the collaboration:
Figure 6.7: User collaboration view, based on the scenario in Figure 6.2.

\[
\begin{align*}
\text{C-WF}(u_{\text{from}}, WF, u_{\text{to}}) & : - \text{run}(r, w, u_{\text{from}}), \\
& \quad \text{workflow}(w, u_{\text{to}}). \\
\text{C-Data}(u_{\text{from}}, Data, u_{\text{to}}) & : - \text{run}(r, w, u_{\text{from}}), \\
& \quad \text{uses}(r,d), \\
& \quad \text{publishes}(d, u_{\text{to}}). \\
\text{C-Run}(u_{\text{from}}, Run, u_{\text{to}}) & : - \text{run}(r_1, w, u_{\text{from}}), \\
& \quad \text{uses}(r_1,d), \\
& \quad \text{produces}(r_2,d), \\
& \quad \text{run}(r_2, w, u_{\text{to}}).
\end{align*}
\]

The generation of C-WF, C-Data and C-Run tables for the example usecase schema are shown in Figure 6.8. Note that the union of all of these collaborations

\[ C(u_{\text{from}}, c, u_{\text{to}}) = \text{C-WF} \cup \text{C-Data} \cup \text{C-Run} \]
Figure 6.8: Generation of (a) C-WF, (b) C-Data, and (c) C-Run views based on the scenario in Figure 6.2.
Figure 6.9: The $C_{NSW}$ graph, based on the scenario in Figure 6.2.

gives the user collaborations with edges labelled according to these attributes, where user $u_{from}$ shares a collaboration of type $e$ with another user $u_{to}$. User collaboration $C$ captures the nature of collaboration and the “self collaboration”. This information is visualized by Figure 6.9 for the scenario in Figure 6.2. $C$ can be extended to include the nature and weight of each kinds of collaborations by performing the following operations on $C$:

1. Perform a group by operation over $C$ with concatenation of $(u_{from} || e || u_{to})$, such that $u_{from} \neq u_{to}$ becomes a column; and

2. Over the group condition generated in (1), retrieve the number of occurrences ($n$) for each unique tuple.

The number of occurrences of each type can be displayed as $c(n)$ for each kind of edge, and appended with a colon “:” to show the full collaboration label, e.g., $WF(1):Data(2)$. The “group by” and “count” operations are standard operations in many relational database query languages (e.g., SQL).

Figure 6.10 shows the $C_{NSW}$ graph generated by performing these operations. Next, using simple Datalog rules (and SQL operations when necessary), we show how these views can be used to answer the example queries in Figure 5.5(a).
Figure 6.10: The $C_{NW{S}}$ graph, based on the scenario in Figure 6.2.

6.3.4 Querying for Combinations of Collaborative Attributes

Below, we make use of SQL to provide a list of queries for combined user collaboration graphs from least informative ($C_{NW{S}}$) to most informative ($C_{NW{S}}$):

- $C_{NW{S}} = \text{SELECT distinct } u_{from}, u_{to} $
  $\text{FROM } c(u_{from}, e, u_{to})$
  $\text{WHERE } u_{from} <> u_{to}$

- $C_{NW{S}} = \text{SELECT distinct } u_{from}, e, u_{to} $
  $\text{FROM } c(u_{from}, e, u_{to})$
  $\text{WHERE } u_{from} <> u_{to}$

- $C_{NW{S}} = \text{SELECT } u_{from}, u_{to}, \text{count}(u_{from}) - ||u_{to}|| \text{ as } w$
  $\text{FROM } c(u_{from}, e, u_{to})$
  $\text{WHERE } u_{from} <> u_{to}$

- $C_{NW{S}} = \text{SELECT distinct } u_{from}, u_{to} $
  $\text{FROM } c(u_{from}, e, u_{to})$

- $C_{NW{S}} = \text{SELECT } u_{from}, u_{to}, \text{count}(u_{from}) - ||u_{to}|| \text{ as } w$
  $\text{FROM } c(u_{from}, e, u_{to})$
As demonstrated by the complexity of these queries, it is not always easy for users to write collaborative queries from scratch. Higher-level query constructs, i.e., expressions, are needed to simplify the generation and optimize the evaluation of collaborative queries. In the following section we extend the data model presented here to additionally support lineage-based path queries using Query Language for Provenance (QLP) (Anand et al. 2009c). Our approach provides a simple mechanism for filtering dependency graphs to answer provenance queries such as those for the collaborative provenance attributes.

6.4 Expressing Collaborative Queries in QLP

We use QLP (Anand et al. 2009c) for expressing lineage queries, and in particular, to filter the dependency graphs described in Section 5.2. In general, answering standard provenance questions (including those of Table 5.1) requires the generation of recursive queries over lineage graphs. Such queries are often complex to express and expensive to evaluate (Moreau et al. 2010, Anand et al. 2009a, Cohen et al. 2006, Heinis and Alonso 2008). As summarized in Section 3.3.1, QLP provides a simple, declarative, path-based language (similar, e.g., to XPath) for expressing such queries, and optimization techniques have been developed that make answering QLP queries over large provenance repositories feasible (Anand et al. 2010b). QLP queries work over sets of lineage edges, e.g., represented by the DerivedFrom relation. A QLP path query \( p \) can be viewed as a filter that selects matching paths within the lineage graph induced by the underlying edges. The result of a QLP query is the set of edges along matching paths of the induced graph. Thus, QLP is a closed language that returns a subset of a given set of lineage edges. Closed languages such as QLP have a number of benefits including the ability to construct views, “incremental” querying, and visualization (Anand et al. 2010c, Anand et al. 2010a).

QLP queries are expressed and evaluated against a selected provenance view, which can be a single workflow run, the entire repository of runs, or the provenance view resulting from a previous query. In the collaborative provenance query scenario, users can use QLP expressions to filter the various dependency views of Figure 5.2. Below we present the basic constructs of QLP and show how QLP can be used to filter dependency graphs (and subsequently answer the queries of Table 5.1; see Table 6.3).
### Lineage-preserving path queries (examples)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>* .. (e_n)</td>
<td>Lineage graph that resulted in nodes in (e_n)</td>
</tr>
<tr>
<td>(e_n) .. *</td>
<td>Lineage graph for nodes derived from nodes in (e_n)</td>
</tr>
<tr>
<td>(e_{n_1}) .. (e_{n_2})</td>
<td>Lineage graph for paths from nodes in (e_{n_1}) to nodes in (e_{n_2})</td>
</tr>
<tr>
<td>(e_{n_1}) .. (r_i) .. (e_{n_2})</td>
<td>Lineage graph for paths from nodes in (e_{n_1}) to (e_{n_2}) passing through run (r_i)</td>
</tr>
</tbody>
</table>

### Functions over lineage path queries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exists(p)</code></td>
<td>True if the selected view contains a path defined by path query (p)</td>
</tr>
<tr>
<td><code>runs(p)</code></td>
<td>The runs of the lineage graph returned by path query (p)</td>
</tr>
<tr>
<td><code>workflows(p)</code></td>
<td>The workflows of the lineage graph returned by path query (p)</td>
</tr>
<tr>
<td><code>artifacts(p)</code></td>
<td>The data nodes of the lineage graph returned by path query (p)</td>
</tr>
<tr>
<td><code>inputs(p)</code></td>
<td>The source nodes of the lineage graph returned by path query (p)</td>
</tr>
<tr>
<td><code>outputs(p)</code></td>
<td>The sink nodes of the lineage graph returned by path query (p)</td>
</tr>
<tr>
<td><code>users(x)</code></td>
<td>The users associated with entity (x); (x) is a set of workflows, runs or artifacts</td>
</tr>
</tbody>
</table>

### Views over lineage path queries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DATA-DEP(p)</code></td>
<td>Data dependencies (Figure 6.3(b)) of the lineage graph selected by path query (p)</td>
</tr>
<tr>
<td><code>RUN-DEP(p)</code></td>
<td>Run dependencies (Figure 6.5) of the lineage graph selected by path query (p)</td>
</tr>
<tr>
<td><code>COLLAB-DEP(p)</code></td>
<td>Collaborations (Figure 6.9) of the lineage graph selected by path query (p)</td>
</tr>
<tr>
<td><code>C-ATT(p, a)</code></td>
<td>Collaborations (Figure 6.9) with user selected attributes, (a), for the lineage graph selected by path query (p)</td>
</tr>
</tbody>
</table>

Table 6.2: Basic QLP constructs and functions, where \(e_n\) is a node expression comprised of either a data artifact identifier, a run identifier, a data artifact type (denoting the set of artifacts having that type), or a workflow (denoting the set of runs of the workflow). We use \(p\) to denote a QLP path query, and \(r_i\) to denote a run. \(a\) is attribute for user collaboration graph, which can be either of these \(\{\phi, N, W, S, NW, NS, WS, NWS\}\).

In the scenario illustrated by Figure 6.2, the lineage information is recorded at a “coarse-grain” level, where only the lineage relationships between inputs and outputs of a run are stored. In the following, we restrict the underlying lineage model of QLP to be over workflow runs, as opposed to the standard use of QLP that supports queries over individual processes within runs (thus modeling lineage at a “fine-grain” level).

Table 6.2 introduces some of the basic constructs and functions of QLP, together with the extensions described here, including the `users`, `DATA-DEP`, `RUN-DEP`, `COLLAB-DEP`, and `C-ATT` functions. As a simple example of a QLP path query, the expression “* .. \(d_7\)” returns lineage edges denoting paths starting from any node in the lineage graph and ending...
at node \(d_7\). Similarly, the query \"d_2 .. *\" returns lineage edges denoting paths starting at node \(d_2\) and ending at any node in the lineage graph. Both “ends” of a path can be fixed in QLP, e.g., the query \"d_5 .. d_9\" returns all edges on paths in the lineage graph that start at \(d_5\) and end at \(d_9\). QLP queries can restrict paths to include intermediate objects, e.g., the query \"#r_4 .. d_6 .. #r_5 .. *\" returns the set of lineage edges denoting paths that start at run \(r_2\), go through artifact \(d_6\) followed by (via one or more lineage edges) run \(r_5\), and end at any node.

### 6.4.1 Filtering Collaborative Provenance Views using QLP

The DATA-DEP, RUN-DEP, and COLLAB-DEP functions construct data, run, and collaboration dependency graphs, respectively, that result from evaluating a QLP query over the current provenance view. Thus, these functions, unlike the DATA-DEP, RUN-DEP, and \(C\) relations defined in Section 6.3, create views purely out of lineage relations.

**Filtering Data Dependency Views.** We write \(v(p)\) to denote the set of lineage edges of the form \(\langle d_2, r, d_1 \rangle \in L\) returned after evaluating a QLP path query \(p\) over a set of lineage edges \(L\) (Anand et al. 2010b). We directly use this evaluation to define the DATA-DEP function as follows.

\[
\text{DATA-DEP}(p) := \{\langle d_2, d_1 \rangle \mid \exists r : \langle d_2, r, d_1 \rangle \in v(p) \land d_1 \neq d_2\}
\]

As shown in Table 6.3, we can use the DATA-DEP function to answer Q2 of Table 5.1, which returns the subset of the data-dependency graph that ends at artifact \(d\). Note that the DATA-DEP function computes a subset of the DDep relation restricted to lineage edges.

<table>
<thead>
<tr>
<th>Q1</th>
<th>Which data artifacts were used explicitly or implicitly to generate data artifact (d)?</th>
<th>artifacts( (*..d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>What is the data dependency graph that led to data artifact (d)?</td>
<td>DATA-DEP( (*..d) )</td>
</tr>
<tr>
<td>Q3</td>
<td>Which runs were used in the generation of a data artifact (d)?</td>
<td>runs( (*..d) )</td>
</tr>
<tr>
<td>Q4</td>
<td>What is the run dependency graph that led to data artifact (d)?</td>
<td>RUN-DEP( (*..d) )</td>
</tr>
<tr>
<td>Q5</td>
<td>If data artifact (d) is detected to be faulty, which runs were affected by (d)?</td>
<td>runs( (d..*) )</td>
</tr>
<tr>
<td>Q6</td>
<td>Which users depended on data artifact (d)?</td>
<td>users( (d..*) )</td>
</tr>
<tr>
<td>Q7</td>
<td>Which user collaborations were involved in the derivation of artifact (d_2) from artifact (d_1)?</td>
<td>COLLAB-DEP( (d_1..d_2) )</td>
</tr>
<tr>
<td>Q8</td>
<td>Who are the potential acknowledgements for a publication of a data artifact (d)?</td>
<td>COLLAB-DEP( (*..d) )</td>
</tr>
</tbody>
</table>

Table 6.3: Example queries expressed using the dependency functions defined for QLP.
Filtering Run Dependency Views. Similarly, to construct a filtered run-dependency graph, we again use the evaluation function as follows.

\[
\text{RUN-DEP}(p) := \{ (r_2, r_1) \mid \exists d_1, d_2, d_3 : (d_3, r_2, d_2) \in v(p) \wedge (d_2, r_1, d_1) \in v(p) \wedge d_1 \neq d_2 \neq d_3 \}
\]

Note that each output of a run within a lineage graph returned by a QLP query is required to be dependent on some input (since only derivation edges are considered). Thus, the run dependencies returned by the \text{RUN-DEP} function have the additional constraint that each output is dependent on some input (within the query result) of the run. This can be viewed as restricting the \text{RDep} relation to only selecting from \text{Produced} edges instead of \text{Output} edges. We can use the \text{RUN-DEP} function to answer Q4 of Table 5.1, as shown in Table 6.3.

Filtering User Collaboration Views. We define the \text{COLLAB-DEP} function as:

\[
\text{COLLAB-DEP}(p) := \text{C-DEP}_{WF}(p) \cup \text{C-DEP}_{DATA}(p) \cup \text{C-DEP}_{RUN}(p),
\]

where the functions \text{C-DEP}_{WF}, \text{C-DEP}_{DATA}, and \text{C-DEP}_{RUN} are defined as follows.

\[
\text{C-DEP}_{WF}(p) := \{ (u_2, \text{WF}, u_1) \mid \exists r, w : \text{Run}(r, w) \wedge \text{Published}(u_1, w) \wedge \text{Performed}(u_2, r) \}
\]

\[
\text{C-DEP}_{DATA}(p) := \{ (u_2, \text{Data}, u_1) \mid \exists d_1, d_2, r : \text{Published}(u_1, d_1) \wedge \text{Performed}(u_2, r) \wedge (d_2, r, d_1) \in v(p) \}
\]

\[
\text{C-DEP}_{RUN}(p) := \{ (u_2, \text{Run}, u_1) \mid \exists d_0, d_1, d_2, r_1, r_2 : \text{Performed}(u_1, r_1) \wedge \text{Performed}(u_2, r_2) \wedge (d_1, r_1, d_0) \in v(p) \wedge (d_2, r_2, d_1) \in v(p) \}
\]

The \text{COLLAB-DEP} function can be used to answer queries Q7 and Q8 of Table 5.1, as shown in Table 6.3.

\text{COLLAB-DEP} returns a user collaboration view which associates to \text{C}(u_{to}, e, u_{from}) in Section 6.3. To support querying of this view using different collaborative provenance attributes, we extend QLP with \text{C-ATT}(p, a), where \(a\) is selected from \{“\text{WT}”, “\text{NT}”, “\text{W}”, “\text{S}”, “\text{NW}”, “\text{NS}”, “\text{WS}”, “\text{NWS}”\}.

6.5 Relation Between the Collaborative Model and OPM

The Open Provenance Model (OPM) (Moreau et al. 2010), as described in Section 3.2.1, has emerged from the e-Science community, and has evolved as a standard representation to facilitate the exchange of information between multiple provenance systems. OPM is based on a model and set of inference rules for directed acyclic provenance graphs, which represent causal dependencies between data products and processes. OPM defines three primary entities (nodes): 1) \textit{Artifacts}: immutable piece of data; 2) \textit{Processes}: actions or series of
actions performed on or caused by artifacts; and (3) Agents: entities that enable, facilitate, control, or affect execution of processes. OPM also defines five primary types of causal dependencies (edges) that comprise provenance graphs: (1) used: a process used artifact(s); (2) wasGeneratedBy: an artifact was generated by a process; (3) wasTriggeredBy: a process was triggered by another process(es); (4) wasDerivedFrom: an artifact was derived from another artifact(s); and (5) wasControlledBy: a process was controlled by an agent. In this section, we explain a mapping of the basic OPM entities to the collaborative provenance model and our extensions to the OPM model to represent these the relationships for publishing of data and workflows.

As it stands, the OPM specification focuses on the provenance for past executions of workflows. The nodes and dependencies related to past workflow runs in the collaborative provenance model can be mapped one-to-one to the basic OPM model as shown in Table 6.4. Artifact and Process nodes in OPM associate to Data and Run in the collaborative provenance model, respectively. The used dependency in OPM is mapped to the Used edge between a Run and Data in our model. Similarly, Users in the collaborative model can be viewed as a form of Agents in OPM, where Performed edges are similar to wasControlledBy edges in OPM. Produced relationship can be captured by the wasGeneratedBy edge in OPM. Note that the direction of the edges for Performed and Produced relationships change when

---

Table 6.4: Mapping of basic Open Provenance Model nodes and edges to the collaborative entities and relationships.

<table>
<thead>
<tr>
<th>Open Provenance Model</th>
<th>Collaborative Provenance Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes</strong></td>
<td></td>
</tr>
<tr>
<td>Artifact</td>
<td>Data</td>
</tr>
<tr>
<td>Process</td>
<td>Run</td>
</tr>
<tr>
<td>Agent</td>
<td>User</td>
</tr>
<tr>
<td>-</td>
<td>Workflow</td>
</tr>
<tr>
<td><strong>Dependencies</strong></td>
<td></td>
</tr>
<tr>
<td>used</td>
<td>Used</td>
</tr>
<tr>
<td>wasGeneratedBy</td>
<td>Produced</td>
</tr>
<tr>
<td>wasControlledBy</td>
<td>Performed</td>
</tr>
<tr>
<td>wasTriggeredBy</td>
<td>r-depends on</td>
</tr>
<tr>
<td>wasDerivedFrom</td>
<td>d-depends on</td>
</tr>
<tr>
<td>-</td>
<td>wasPublishedBy</td>
</tr>
<tr>
<td>-</td>
<td>wasExecutedIn</td>
</tr>
<tr>
<td><strong>Inference Rules</strong></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>hasDataCollaborationWith</td>
</tr>
<tr>
<td>-</td>
<td>hasRunCollaborationWith</td>
</tr>
<tr>
<td>-</td>
<td>hasWFCollaborationWith</td>
</tr>
</tbody>
</table>
6.5 Relation Between the Collaborative Model and OPM

Figure 6.11: The entities and edges in the standard OPM model were extended by Workflow (WF) entity, and `wasPublishedBy` and `wasExecutedIn` edges in the collaborative provenance model.

Figure 6.12: An abstract model of collaborative provenance nodes and dependencies using the extended Open Provenance Model.

depicted as `wasControlledBy` and `wasTriggeredBy`. `r`-depends on and `d`-depends on (see
Figure 6.1) relationships can be captured using \textit{wasTriggeredBy} and \textit{wasDerivedFrom}. For example, \(<d_4, r_1, d_1>\) lineage relation stating that artifact \(d_1\) was \textit{used} by the run \(r_1\) to produce artifact \(d_4\) can be captured as artifact \(d_4\) \textit{wasDerivedFrom} artifact \(d_1\) and artifact \(d_4\) \textit{wasGeneratedBy} workflow run \(r_1\). Adjacent lineage relations, e.g., \(<d_7, r_3, d_4>\) and \(<d_4, r_1, d_1>\) state that run \(r_3\) \textit{wasTriggeredBy} run \(r_1\). However, to the best of our knowledge, OPM does not provide support for recording when users publish data and workflows, which is essential in the collaborative model proposed here for creating the various types of user collaborations.

Table 6.4 also shows the extensions to the OPM to represent the publishing relationships. A special Workflow node was added and is defined as “a specific kind of artifact that refers to the workflow description that is published by the user, and gets executed in one or many processes (workflow runs)”. Using a \textit{wasPublishedBy} edge between an agent and workflow or data is added to the model. We also capture the relationship between a workflow and a run (process) that executes this workflow explicitly using the \textit{wasExecutedIn} edge. Figure 6.11 illustrates the nodes and edges that were added to the OPM to complete the mapping to the collaborative model.

Finally, Figure 6.12 shows the collaborative provenance model in Figure 5.3 using the defined OPM extensions. \textit{hasWFCollaborationWith}, \textit{hasRunCollaborationWith}, and \textit{hasDataCollaborationWith} in Figure 6.12 can be inferred using the extended nodes and edges as follows:

- If process \(p_1\) \textit{wasControlledBy} agent \(a_1\), and workflow \(w_1\) \textit{wasPublishedBy} agent \(a_2\) and \textit{wasExecutedIn} \(p_1\), then we can infer that \(a_1\) \textit{hasWFCollaborationWith} \(a_2\).

- If process \(p_1\) \textit{wasControlledBy} agent \(a_1\) and \textit{used} artifact \(a_1\) that \textit{wasPublishedBy} agent \(a_2\), then we can infer that \(a_1\) \textit{hasDataCollaborationWith} \(a_2\).

- If process \(p_1\) \textit{wasControlledBy} agent \(a_1\) and \textit{used} artifact \(a_1\) that \textit{wasGeneratedBy} process \(p_2\) that \textit{wasControlledBy} agent \(a_2\), then we can infer that \(a_1\) \textit{hasRunCollaborationWith} \(a_2\).

The formal definition of the OPM extensions and the formal equations for how these inference rules are left out of this thesis since the latest OPM specification in (Moreau et al. 2010) does not include a formal definition of OPM.

Summary

We described a data model that is effective to capture collaborative provenance scenarios. Based on this data model, we presented queries to generate collaborative views and showed how it can be used to answer collaborative provenance views queries ranging from \(C_{NWS}\) to \(C_{NWS}\). We discussed extensions to the QLP for collaborative querying and a mapping of the collaborative provenance relationships to OPM. In the next section, we will present application usecases for collaborative provenance.