

Querying an integrated complex-object dataflow database

Peer-reviewed author version

KWASNIKOWSKA, Natalia & VAN DEN BUSSCHE, Jan (2013) Querying an integrated complex-object dataflow database. In: In Search of Elegance in the Theory and Practice of Computation: Essays Dedicated to Peter Buneman , p. 400-417.

DOI: 10.1007/978-3-642-41660-6_21

Handle: <http://hdl.handle.net/1942/16405>

Querying an integrated complex-object dataflow database

Natalia Kwasnikowska and Jan Van den Bussche

Hasselt University and Transnational University of Limburg, Belgium

Abstract. We consider an integrated complex-object dataflow database in which multiple dataflow specifications can be stored, together with multiple executions of these dataflows, including the complex-object data that are involved, and annotations. We focus on dataflow applications frequently encountered in the scientific community, involving the manipulation of data with a complex-object structure combined with service calls, which can be either internal or external. Internal services are dataflows acting as a subprogram of an other dataflow, whereas external services are modeled as functions with a possibly non-deterministic behavior. Dataflow specifications are expressed in a high-level programming language based on the nested relational calculus, the operators of which provide the right “glue” needed to combine different service calls into a complex-object dataflow. All entities involved, whether complex-objects, dataflow executions or dataflow specifications, are first-class citizens of the integrated database: they are all data. We discuss how such dataflow repositories can be queried in a variety of ways, including provenance queries. We show that a modern SQL platform with support for (external) routines and SQL/XML suffices to support all types of dataflow repository queries.

Dedicated to Peter Buneman.

1 Introduction

A workflow is a high-level specification of a complex and possibly long-during task, consisting of different subtasks that must be performed in a certain order. This order does not need to be linear: some tasks can be performed concurrently, or alternatively. Workflow management has its origins in business process modeling [1], but in recent years workflows have gained importance in e-science, in parallel with the rise of Grid Computing [2]. Scientific workflows are distinguished from business workflows by their placing more importance on the data flow between the subtasks, than on the synchronization of subtasks [3]. (In e-science, the data flow frequently involves collections of complex data objects.) In accordance to this focus, in this paper, we use the terms “scientific workflow” and “dataflow” interchangeably.

With the rise of scientific workflows, the need for better database support became apparent. A nice overview of relevant topics in database support for

scientific workflow management has been given in a special issue of SIGMOD Record [4]. The needs in this area go well beyond what so-called workflow management systems (WFMS) provide, even if coupled to a DBMS such as Oracle Workflow or IBM WebSphere. Such WFMSs provide support for constructing workflow specifications and guiding and monitoring workflow executions.

In an e-science environment, however, one is confronted with multiple inter-related research projects, where in each project a multitude of different dataflows are in use, each of which has been executed many times, on different input data, using different versions of external services, by different users, and so on. Standard WFMSs, although they are implemented on top of a database, lack the support for ad-hoc querying of all this information in an integrated manner. Such database support is important to manage computational experiments, to allow reproducibility, and more generally, to “enforce the scientific method” [5].

In response to this problem, in an earlier paper [6], we gave the formal specification of an integrated dataflow repository. In the current paper, we show how an implementation of this system on top of a modern but standard SQL platform, enables the querying of dataflows and dataflow executions in an integrated manner. We intensively use such features as external routines, user-defined table functions, SQL/XML and XQuery capabilities.

Of course we are not the first to address the challenge of querying dataflow executions; in the scientific workflow community such queries are known as “provenance queries”. The participants of the Provenance Challenges¹ have already intensively investigated this direction.

Our present approach focuses on the following aspects:

1. We explicitly represent the complex-data manipulations that are performed in a dataflow. We do this using the nested relational calculus (NRC [9]): an elementary functional programming language composed of all the natural manipulation operators on collection- and record-oriented data.
2. We investigate the feasibility of a 100% database solution. Using the full power of the modern SQL:2003 standard, we will see that all types of provenance queries can be solved directly in SQL. Many present solutions of the Provenance Challenge mentioned above involve coding of diverse programs outside of the database. In contrast, our approach focuses on a fixed set of user-defined functions that can then be used in SQL select statements.
3. We address querying not only of dataflow executions but also of the specifications of the dataflows. In the same vein, we address querying of executions of dataflows, the specification of which is not determined in advance. Previous approaches to dataflow provenance typically focus on querying executions of a fixed dataflow specification, given outside of the query.
4. Thanks to our explicit complex-object data model, we can support a new, finer-grained notion of provenance tracking, where we can derive connections directly among *subvalues* occurring in the result and the intermediate results of a dataflow execution.

¹ <http://twiki.ipaw.info/bin/view/Challenge/WebHome>

This paper is organized as follows. In Section 2 we describe related work. In Section 3 we briefly discuss the types of queries which illustrate the need for an integrated dataflow database. In Section 4 we use simple examples to discuss the database representation of complex data, NRC dataflows, and the integration of external services. In Section 4.4 we briefly describe the execution of dataflows, taking into account such issues as binding of input data, binding of external function calls, and binding of subdataflows. We conclude the section with the representation of past executions in the dataflow database. In Section 4.5 we briefly describe how the database can integrate annotations.

Finally, in Section 5, we show how the integrated complex-object dataflow database can be queried in a variety of ways.

2 Related work

Support for the complex-object structure of data flowing in a scientific workflow is present in various systems, e.g., Taverna [10, 11], Kepler/CoMaD [12, 13], and Chimera [14, 15]. The operation of applying a function on all elements of a collection is typically provided. However, that operation is just one of the many possible kinds of “glue” needed to connect different subtasks in a complex data flow together. Indeed, the NRC which we use provides exactly the natural set of operations to deal with complex-object data. It has evolved from a long tradition of complex-object data modeling in the database research literature.

Also a database-oriented approach has been advocated by many others [16–20]. However, the querying of an unbounded number of dataflow executions, as given by the database instance and not fixed in advance, as well as the querying of dataflow specifications inside the database, has not been addressed before. Our approach is based on our earlier experience with database meta-querying [21, 22]. We should also mention the topic of process mining [23, 24], although the scope of process mining is quite different from that of repository querying.

We represent a dataflow execution in the database as a “log”, i.e., a set of triples of the form (input, function, output). This representation is natural and common [25, 26, 19, 27, 28], and is often equivalently viewed as a causality graph. Specific to our approach is that we can define a finer-grained tracking of provenance not just from output to input, but also from a *subvalue* occurring in the output to a *subvalue* occurring in the input. Note that in our previous paper [29] we have given a conversion from our execution model to the proposed standard Open Provenance Model [30], which uses an explicit causality graph.

We should also mention some more distantly related work. Beer, Milo et al. have an interesting project on querying the *potential* executions of a given workflow specification [31]. That approach is mainly verification-oriented rather than repository-oriented, although they did also consider monitoring [32]. The NRC was used in the Kleisli system [33, 34] not as a dataflow specification language, but as a bioinformatics data integration query language, where the entire structure of the biological data is modeled as a complex object. We use complex objects in a different way, to model the data flow in a scientific workflow.

The NRC is also used as a framework to formalize provenance and dependency analysis for queries over annotated databases [35, 36].

We conclude by pointing out that the need for a workflow repository is also acknowledged in other fields, as shown by Blockeel and Vanschoren’s Experiment Databases for Machine Learning [37, 38].

3 Motivation

The participants of the Provenance Challenges² have already informally formulated various queries, involving both a dataflow specification and its past executions.

For example, for a specified part of a workflow output, say *out*, they have formulated queries that ask (i) which workflow inputs have contributed to the computation of *out* (Q1, Q5 from PC3); (ii) which part of the execution contributed to the computation of *out*, possibly further restricted by annotations, or only up to a specified task (Q1-Q3 from PC1, Q3 from PC3); (iii) to verify if certain tasks were involved in the computation of *out* (Q2 of PC3); (iv) to look for tasks that can be swapped during execution without affecting *out* (optional Q5 from PC3).

Queries that involve many executions of the same workflow ask (i) to find all invocations of a specified task, using a specified input, and having specified annotations (Q4); (ii) to retrieve (intermediate) results produced by a specified task and/or having specified annotations (Q8-Q9 from PC1), or even preceded by another specified task (Q6 from PC1); (iii) to find all workflow outputs produced from a specified input (Q5 from PC1); (iv) to find differences between specified past executions (Q7 from PC1). We concur that a dataflow repository should allow formulating such queries, and we illustrate in Section 5 how it can be done in our model.

In general, there are various types of queries that a dataflow repository should support, including:

- Queries involving *subvalues* of a (final) result. Indeed, in some dataflows, both intermediate values and the final result value may be huge data sets, and the user might be only interested in some part.
- Querying vast amounts of past executions, in order to identify dataflows and their executions involving a particular external service. Indeed, if that service produced erroneous results, or there is a better implementation available, such queries are necessary if we want to rerun the affected dataflows with another external service.
- Queries that allow modifying of dataflow specifications and immediate execution of the modified dataflows.

We show in Section 5 how such queries can be constructed for our integrated dataflow repository, after a description of a possible implementation in the following section.

² <http://twiki.ipaw.info/bin/view/Challenge/WebHome>, we refer to the first challenge as PC1, and to the third as PC3.

4 Complex-object dataflow database

In an earlier paper [6], we gave the formal specification of a dataflow repository. In this section we show how we can represent all aspects of that formal model on top of a modern SQL platform.

4.1 Complex data

Data objects flowing in a scientific workflow can either be atomic for the workflow, or can have a structure that is important for the workflow. The two basic data structures in databases are the set, e.g., $\{sequence_1, \dots, sequence_{76}\}$, and the tuple, e.g., $\langle organism: mouse, \dots, filename: GPZ158 \rangle$. These structures can be arbitrarily nested: we use the complex-object data model [39]. For more details on the theory, including the type system, we refer to our previous paper [6], as here we are focusing more on the implementation and use of the system.

It is important to note that an “atomic” object can be quite complex, e.g., it can be a file, it can be an XML document. However, for a dataflow that has only actions that operate on the file as a whole, it is not relevant to model the file as a set of records. On the other hand, if the structure of the file as a collection is important, because we want to apply some operation to each of its elements, then we model the file as a complex object.

We represent atomic objects as strings. For small types of atomic objects, such as numbers, strings or dates, the string can hold the entire value of the object. For large atomic objects such as files, we could still represent them as a string by means of a path name of the file.

In many cases, however, it is more desirable to store the large atomic object in the database as a BLOB (which can contain a text file or an XML document as well as a binary file). In that case, the string representing the object is an identifier that can be used as a foreign key to the object in table `Pool(ID, object)`.

As to storing complex objects, we discuss two basic ways: *decomposition* and *XML* representation.

Decomposition of complex objects. A complex object, together with its nested subobjects, can be naturally viewed as a tree. We generate a string ID for each tuple and set node; the atomic objects, which occur as leaves in the tree, already have their string representation. We then store the tree in two tables: `Sets(ID, eID)` and `Tuples(ID, att, fID)`. Here, `eID` stands for element ID, `att` stands for attribute, and `fID` for field ID. Figure 1 shows an illustration for the following complex object:

```
{(exp: P2T42, targets: {human, mouse}, result: report123),
 (exp: P42T3, targets: {human, chimp}, result: report456)}
```

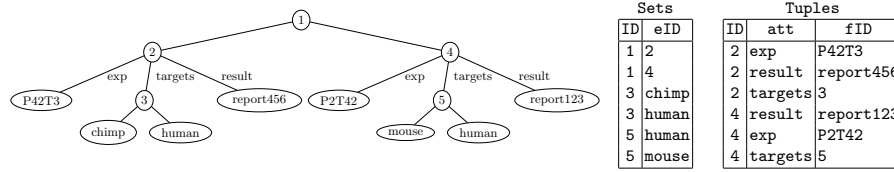


Fig. 1. Tree representation and decomposition of a complex object.

XML representation of complex objects. We can also take advantage of the XML data type supported by modern database systems, and store the complex-object tree directly as an XML value. This is illustrated in Figure 2. There is an additional choice when the complex object contains XML documents as large atomic objects at the leaves: we can just have the IDs of these objects at the leaves of the XML tree, or we can include their full XML content. For example, in Figure 2, the results are represented by IDs **report123** and **report456** referring to the **Pool** table, but alternatively we could have replaced these IDs inside the XML tree by the corresponding full XML reports.

The best choice among decomposition, intermediate XML, and full XML for complex objects depends on the application. We can provide library routines to move between the three representations; these routines can then be called in SQL statements.

4.2 NRC dataflows

In its most simple form, a dataflow is a pipeline of function applications, as illustrated in Figure 4, or expressed in the dataflow language we use in Figure 5. The function names **analyze**, **compare** and **annotate** represent the basic actions or tasks of which the dataflow is composed. In e-science and e-commerce settings, these tasks are often called *services*, so we refer to the function names in a dataflow as *abstract service names*. They are abstract in the sense that they serve only as placeholders for actions: only when the dataflow is actually executed, the abstract service names are bound to concrete actions.

Since the data objects flowing in the pipeline can have complex structure, a language with just variable definitions (the **let**-construct) and function application, as used in the above example, is not sufficient. For example, if x is a set, we want to apply **analyze** to every element of x and collect the results. We can accommodate this by adding a mapping construct $\{\mathbf{analyze}(u) \mid u \in x\}$ to the language, in the form of **for** $u \in x$ **return** **analyze**(u). In order to be able to organize the data flow, we also want the basic operations on tuples and sets: tuple formation, tuple projection, singleton set formation, set union, and big union, also known as “flatten”.³ Finally, we need an if-then-else construct. This rounds up the operations of a natural language for complex objects known as the *nested relational calculus* or NRC.

³ The flattening $\bigcup s$ of a set of sets $s = \{s_1, \dots, s_n\}$ equals $s_1 \cup \dots \cup s_n$.

```

<set>
<tuple>
  <att> exp </att>
  <atom> P2T42 </atom>
  <att> targets </att>
  <set>
    <atom> human </atom>
    <atom> mouse </atom>
  </set>
  <att> result </att>
  <atom> report123 </atom>
</tuple>
<tuple>
  <att> exp </att>
  <atom> P42T3 </atom>
  <att> targets </att>
  <set>
    <atom> human </atom>
    <atom> chimp </atom>
  </set>
  <att> result </att>
  <atom> report456 </atom>
</tuple>
</set>

```

Fig. 2. XML representation of a complex object.

```

<expr ID="0">
  <let ID="1">
    <var ID="2"> z </var>
    <for ID="3">
      <var ID="4"> u </var>
      <var ID="5"> x </var>
      <tuple ID="6">
        <att> a </att>
        <project ID="7">
          <att> a </att>
          <var ID="8"> u </var>
        </project>
        <att> b </att>
        <call ID="9">
          <name> extract </name>
          <project ID="10">
            <att> c </att>
            <var ID="11"> u </var>
          </project>
        </call>
      </tuple>
    </for>
    <call ID="12">
      <name> validate </name>
      <call ID="13">
        <name> search1 </name>
        <var ID="14"> z </var>
        <var ID="15"> y </var>
      </call>
      <call ID="16">
        <name> search2 </name>
        <var ID="17"> z </var>
        <var ID="18"> y </var>
      </call>
    </call>
  </let>
</expr>

```

Fig. 3. XML representation of the NRC expression of Figure 6.

So, as already seen in Figure 5, a dataflow consists of a name, a specification of its input parameters, and a specification of its behavior in the form of an NRC expression. Another example is shown in Figure 6. Actually, our system is typed [6], so input and return types, as well as service signatures should also be specified. For simplicity of presentation, however, we omit the typing system.

Storing dataflow specifications in the repository. Dataflow specifications are stored in a table **Dataflows** with attributes **ID** and **expr** in which the name and the NRC expression are stored. (There are also attributes to store type information.) Here, attribute **expr** is of type XML: we store the expressions by their syntax tree in XML format, as illustrated in Figure 3.

Note that the element nodes in the XML syntax tree have unique ID attributes. This allows us to create an index on XML column **expr** based on the XPath pattern `//*[@ID]`. This is useful to support efficient querying of stored expressions using SQL/XML. Indeed, as we will see later, some other tables in

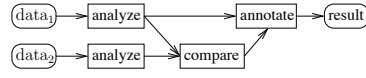


Fig. 4. A dataflow.

dataflow AFlow(x, y) returns
 let z := analyze(x)
 in annotate(compare(z, analyze(y)), z)

Fig. 5. Specification of AFlow.

dataflow BFlow(x, y) returns
 let z := for u in x return
 <a: u.a, b: extract(u.c)>
 in validate(search1(z,y),
 search2(z,y))

Fig. 6. Specification of BFlow.

```

<btree>
<entry>
  <aname>extract</aname><ename>EXTR</ename>
</entry>
<entry>
  <aname>validate</aname><ename>VAL</ename>
</entry>
<entry>
  <aname>search1</aname><sub>CFlow</sub>
  <btree>
    <entry>
      <aname>dbsearch</aname><ename>SQST</ename>
    </entry>
  </btree>
</entry>
<entry>
  <aname>search2</aname><sub>CFlow</sub>
  <btree>
    <entry>
      <aname>dbsearch</aname><ename>MSCT</ename>
    </entry>
  </btree>
</entry>
</btree>
  
```

Fig. 7. Binding tree.

the repository database contain references to these IDs, so many queries use conditions involving the above XPath pattern. We show examples in Section 5.

4.3 External services and subdataflows

The functions we want to call in a dataflow execution are called *external services*, because they represent a computation that is external to the dataflow specification, i.e., not further modeled within the dataflow specification. In e-science, services can be local programs, remote programs, Grid service calls or Web service calls, and so on.

In order to integrate external services in the dataflow database, we assume Java wrappers for them, which are registered as external routines (user-defined Java functions). These functions take XML representations of complex objects as input and output. In this way, external services can be called directly in SQL statements, but also, dataflow executions can be initiated from inside the database server.

So, before we start the execution of a dataflow, we bind some of the abstract service names occurring in the body to names of external routines. Other abstract service names, however, may be bound to names of other dataflows in the

repository. Indeed, in order to support modular programming of workflows, we want to be able to let one dataflow call another one as a subdataflow.

The specification of binding of abstract service names to external routines or subdataflows, together with the further binding of abstract service names occurring in those subdataflows, is called a *binding tree* and can be naturally represented in XML. Recall, for example, dataflow BFlow from Figure 6. To execute BFlow we might want to bind **extract** and **validate** to external routine names **EXTR** and **VAL**, and both **search₁** and **search₂** to another dataflow CFlow. Assume that CFlow calls just one abstract service name **dbsearch**. In the CFlow executions within BFlow that are called as **search₁**, we want to bind **dbsearch** to external routine **SQST**, but in the subdataflow executions called as **search₂**, we want to bind **dbsearch** to **MSCT**. The binding tree that specifies all this is shown in XML in Figure 7.

4.4 Executions

To execute a dataflow known to the repository, the system offers, as a library routine, the stored procedure *Execute*(*flowID*, *vassign*, *btree*), where *flowID* designates a dataflow from the **Dataflows** table, *btree* is a binding tree for the dataflow, and *vassign* is an assignment of input values to the input parameters of the dataflow.

This value assignment is given in XML in the following format:

```
<vassign>
  <entry> <name> x </name> <val> v </val> </entry>
  ...
</vassign>
```

Here, *x* stands for an input parameter and *v* for the input value for *x*. When using the decomposed representation of complex objects, we can give *v* as an identifier to be found in the **Sets** and **Tuples** tables. When using XML representation, *v* is itself a further XML subtree. So, formally, we have different variants of *Execute*, but we omit this here from our notation.

The behavior of *Execute* is such that a log of the execution is stored in the repository. We call such a log a “run”. Runs are stored in tables **Runs**(ID, flowID, *vassign*, *btree*) and **Triples**(ID, caller, *cassign*, subexpr, *vassign*, value). Here, the ID of the run is newly generated. With this ID, a number of tuples, called “tagged triples”, are inserted in the **Triples** table: one holding the final result value, and one for each service call that has been made. This is necessary because external services need to be considered as non-deterministic functions. For example, in Bioinformatics, public search services (for genes, proteins, etc) are heavily used and called through Web interfaces, but the underlying contents change daily. So, in order to allow for querying of past executions of dataflows in the repository, it is crucial that the results returned by the service calls are stored, because we cannot simply rerun the dataflow later on the same inputs and still be certain to get the same results.

The columns of the **Triples** table have the following meaning. Let us first consider the service calls made in the main dataflow execution, so not in sub-dataflows. Column **subexpr** holds the ID of the place of the call in the syntax tree of the dataflow expression. Column **vassign** holds the values of the dataflow parameters at the time of the call. Column **value** holds the return value of the call. For the tuple holding the final result value, **subexpr** is simply the identifier of the root element of the syntax tree, and **vassign** is the original input assignment.

dataflow mapF(input) returns
for x in input return f(x)

dataflow myFlow(input) returns
<c: f(g(input.a)), d: f(g(input.b))>

Fig. 8. A simple flow.

Fig. 9. Another simple flow.

Let us illustrate the **Triples** table on the simple dataflow of Figure 8. Assume the node ID of the call $f(x)$ in the syntax tree is 4, and that of the entire expression equals 1. Assume the input value equals the set $S = \{a, b, c\}$; then a possible run could generate the following triples: (we write the value assignments in an abbreviated form, and abbreviate **input** by i)

$(1, [i = S], \{55, 66\});$
 $(4, [i = S, x = a], 55); (4, [i = S, x = b], 55); (4, [i = S, x = c], 66).$

So we see that $f(a)$ and $f(b)$ both returned 55, and $f(c)$ returned 66.

So far we have ignored the columns **caller** and **cassign**. In the triples corresponding to the execution of the main dataflow, these columns are NULL. In triples corresponding to subdataflow executions, these columns hold the ID of the call subexpression and the values of the dataflow parameters at the moment of the call.

Implementation. *Execute* can be implemented quite straightforwardly by compiling NRC into SQL/XML. Indeed, under the decomposed representation of complex objects, NRC operations can be quite simply programmed in SQL. We have already seen that external services can be called in SQL as external routines. Under the XML representation of complex objects, either decomposition can be applied first (this is the approach we take in our prototype), or a direct compilation of the NRC operations into XQuery may be performed.

4.5 Annotations

The basic set of tables **Dataflows**, **Runs**, and **Triples** that constitute the dataflow repository can, of course, be supplemented with extra tables in which extra information, known as *annotations* or *meta-data*, can be stored. These tables are application-dependent, and can refer to the IDs of the elements stored in the basic tables. Examples of meta-data can be authorship, dates and times, version information, categories of dataflows according to projects, and so on.

Annotation hooks. There is one kind of annotation that must be performed by the execution system. This is when we want to record the start or end date&time of runs, or properties of external service calls, such as date&time again, but also possible error codes and so on. In order to provide applications with a flexible annotation recording of runs, the procedure *Execute* can provide a hook that is called before and after each service call. The application developer can instantiate this hook with the code necessary to record the meta-data required by the application.

Note that for dataflows with known specification, no date&time information is necessary to determine the order of execution of (external) services. If the execution of a service depends on the result of another service, the order of execution of services can be determined from the subexpression identifiers stored in the **Triples** table and the dataflow specification in the **Dataflows** table.

5 Querying the repository

In this section we show that our approach facilitates querying in various ways:

(i) Queries involving subvalues

Apart from the obvious provenance queries, i.e., asking for the part of the run that has contributed to a certain subvalue, we can use information stored in the **Triples** table, e.g., to query the runs of relevant subdataflows if the subvalue is an element of a collection. Table **Triples** can also be used for querying multiple executions of different dataflows, even without knowing their specifications, to determine, e.g., executions that involved calls to a certain internal service. We note that with the addition of dataflow specifications stored as XML, more sophisticated queries can be formulated, to determine, e.g., executions involving certain subexpressions as well as the order in which the subexpressions were executed. For provenance queries involving dataflows which specifications are unknown, we provide function *Prov*.

(ii) Queries involving (external) services.

Table **Triples** in combination with the binding trees stored as XML in table **Runs** can be used for querying multiple executions of dataflows, to determine, e.g., executions that involved calls to a certain external service, or which external services have produced a certain subvalue.

(iii) Queries executing modified dataflow specifications.

We provide function *Eval* for the on-the-fly execution of dataflows with modified specification, with modifications in the subexpressions, as well as in binding to external services.

Queries involving subvalues. Some of the sample queries of the Provenance Challenge [7] are of the following kind. We are given a dataflow, for example, **myFlow** shown in Figure 9. Suppose we have run this flow, with run-ID **myRun**, and we observe the output value `<c: 55, d: 66>`. Consider now the query “What is

the process that produced the value 55 in the output?” From the dataflow specification we see that 55 is the output of *f* applied to the output of *g* applied to *input.a*. We can thus retrieve the two relevant triples as follows:

```
select 'input.a', R.value
  from Triples R, Dataflows D
 where R.ID='myRun' and D.ID='myFlow'
       and xmlexists('$e/*[@ID=$s][name()="project"][att="a"]',
                     passing D.expr as "e", R.subexpr as "s")
union
select 'g', R.value
  from Triples R, Dataflows D
 where R.ID='myRun' and D.ID='myFlow'
       and xmlexists('$e/*[@ID=$s][name()="call"][name="g"][project/att="a"]',
                     passing D.expr as "e", R.subexpr as "s")
```

Note the use of the SQL/XML predicate `XMLEXISTS` [40,41] to retrieve the IDs of the nodes in the syntax tree of the dataflow’s NRC expression.

Things get a bit more subtle when working with collections. Recall dataflow `mapF` from Figure 8.

Suppose the run of `mapF` with ID `myRun2` yields a final result value `{55,66}`, and we again want to know the process that produced the subvalue 55. From the dataflow specification we see that 55 is the result of *f* applied to at least one element of the input collection. We thus want to retrieve all elements *x* in *input* for which *f*(*x*) resulted in 55:

```
select xmlquery('$a/var[name="x"]/val'
               passing R.vassign as "a")
  from Triples R, Dataflows D
 where R.ID='myRun2' and D.ID='mapF'
       and xmlexists('$e/*[@ID=$s][name()=call]',
                     passing D.expr as "e", R.subexpr as "s")
       and xmlexists('$v[.="55"]', passing R.value as "v")
```

Note the use of the SQL/XML function `XMLQUERY` to extract the value of variable *x* from the value assignment of the triple.

So far we have been querying one execution of a given dataflow. However, we can as well pose queries across all dataflows, and all their executions, in the database instance. For example, queries like “List all bioinformatics dataflows in which a function named *f* is called with parameter *p* equal to 5, and the value ‘GPZ158’ appears in the result of the call.” can be expressed using similar techniques as above (assuming an annotation table that lists the IDs of bioinformatics dataflows).

The two earlier example queries over the executions `myRun` and `myRun2` are simple but typical examples of provenance queries, where we ask for the process that lead to a given value occurring as a subvalue of the output. When the dataflow specification is known in advance (in the examples, `myFlow` and `mapF`), we have seen that provenance can be directly expressed in SQL. This is no longer straightforward, however, when the dataflow specification is unknown. Our solution is to provide a generic *provenance* computation as a library routine,

which can be implemented in a programming language like Java, or even as an SQL/PSM routine. Concretely, we provide a user-defined table function *Prov* with the following signature:

```
function Prov(runID integer, subval integer)
  returns table (caller_expr XML, caller_vassign XML,
                subexpr XML, vassign XML, subval2 integer)
```

Here, *subval* is the ID of an occurrence of a subvalue in the output of run *runID*. The returned set of tagged triples is much like the set of tagged triples representing a run, but there are three important differences:

1. In the set of triples representing a run, there is one triple for each service call. We have seen that this is sufficient to reconstruct all the intermediate results of NRC operators in between the calls, but that is true only if the dataflow specification is given. Since this is not the case here, the function *Prov* returns triples for NRC operators as well as for service calls.
2. Moreover, *Prov* returns triples only for those operators and service calls that played a role in the generation of *subval*.
3. Indeed, normal run-triples contain the result values of the intermediate steps of the run. However, here we are asking for the process that lead to a *subvalue* of the final output. Accordingly, the function *Prov* returns all *subvalues* (column *subval2*) of intermediate results of the dataflow execution that lead to *subval*.

For example, the earlier query about *myRun2* can now be expressed using *Prov* without any reference to *mapF*, so that it can be applied to, say, all dataflow executions done on a given date:

```
select P.subval2
from Runs R, RunDates D,
  lateral (values xmlcast(xmlquery('$r//*[.="55"]/@ID'
                                passing R.result as "r")
                                as integer)) as I(thesubvalue),
  table(Prov(R.ID, I.thesubvalue)) as P
where D.runID=R.ID and D.when='2009-02-24'
```

For a formal specification of *Prov* we refer to our previous paper [6].

Queries involving (external) services. Consider an external service that is registered in the database as the external function *BLAST2008*. The database may contain many dataflow executions that have called this service. To retrieve them, it suffices to look in the binding tree of each execution, which is stored together with the run-ID in the *Runs* table. The following query also retrieves the abstract service name that is bound to *BLAST2008*.

```
create view B2008calls as
select U.ID, Tree.aname
from Runs U, xmltable('$tr/tree/entry'
                      passing U.btree as "tr")
```

```

        columns aname varchar(30),
        ename varchar(30)) as Tree
where Tree.ename='BLAST2008'

```

Now suppose we want to understand the effect of replacing the external function `BLAST2008` by another one, say, `BLAST2009`. We are interested, across all executions in the database, which calls to `BLAST2008` would give a different result when replaced by a call to `BLAST2009`. (We assume that the data used by all those dataflows remained unchanged.) We can find this out by the following query:

```

select O.ID, O.subexpr, O.argval, O.value, N.newvalue
from
  (select U.ID, D.subexpr, R1.value, R2.value as argval
   from Runs U, B2008calls B, Dataflows D, Triples R1, Triples R2
   where U.ID=B.ID and U.flowID=D.ID
        and U.ID=R1.runID and U.ID=R2.runID and R1.vassign=R2.vassign
        and xmlexists(
          '$e//*[ @ID=$s1 ][name()="call"][name=$b]/child::*[2][ @ID=$s2] ',
          passing D.expr as "e", R1.subexpr as "s1",
          B.aname as "b", R2.subexpr as "s2")
   ) as O,
  lateral (values BLAST2009(O.argval)) as N(newvalue)
where is_different(O.value, N.newvalue)

```

Observe how the query directly calls `BLAST2009` on the inputs of the recorded calls to `BLAST2008`. We also use a Boolean user-defined function `is_different` to compare the two resulting XML values, as a literal non-equality is not what we want.

Queries executing modified dataflow specifications. What if we want to find those dataflow executions whose *final result* would change if we replaced `BLAST2008` by `BLAST2009`? Note that a difference in an individual call might not result in a difference in the final result. To answer this query, we can no longer directly call `BLAST2009` as before, because we have to continue the process with the rest of the dataflow, which is unknown at query time. (Of course, if we are only interested in the executions of a dataflow whose specification is known in advance, we can simply rerun it, either through the repository or directly in a query, and compare the differences.)

The solution lies in the provision of dynamic dataflow execution through a library function. More specifically, we provide a user-defined table-valued function *Eval* with the following signature:

```

function Eval(expr XML, vassign XML, btree XML)
returns table (caller XML, cassign XML, subexpr integer,
              vassign XML, value XML)

```

This function returns the set of tagged triples representing the run of NRC expression `expr` on value assignment `vassign` and binding tree `btree`. So, *Eval*

is like a lightweight version of procedure *Execute*, where the run is not stored in the repository but is merely made available for ad-hoc querying.

The astute reader will note that there is an issue with the `subexpr` column in the table returned by *Eval*. In a normal execution stored in the integrated repository, this column refers to the unique ID attribute of the nodes in the syntax tree of the dataflow expression. However, here, the input to an *Eval* call is an arbitrary expression `expr`, dynamically produced in XML format during the query, where we do not want to require that every node in *expr* has a unique ID attribute. This issue is solved by letting the `subexpr` column now refer to the numbers of the nodes, in document order. To retrieve nodes, instead of `$e//[@ID=$s]`, we can use `$e/descendant-or-self::*[$s]`.

We are now able to express our query asking for those dataflow executions whose final result would change if we replaced BLAST2008 by BLAST2009.

```
select O.ID, O.result, E.value
  from (select U.result, D.expr, U.vassign, U.btree
        from Runs U, Dataflows D
       where U.flowID=D.ID
          and xmlexists('$b//ename[.="BLAST2008"]
                        passing U.btree as "b"') as O,
       lateral ( values
                xmlquery('copy $newb := $b
                          modify for $n in $newb//entry
                          where $n/ename="BLAST2008"
                          return
                            replace value of node $n/ename
                            with "BLAST2009"
                          return $newb'
                        passing O.btree as "b" ) as N.newbtree,
        table ( Eval(O.expr, O.vassign, N.newbtree ) as E
       where E.subexpr=1 and is_different(O.result, E.value)
```

The condition `E.subexpr=1` on the last line selects the top-level node so as to retrieve the final result value of each rerun. Note also the use of XQuery Update facilities. These are already supported in some SQL/XML implementations, for example, DB2 v9.5.

In the above example, we only rewrite the binding trees, not the actual NRC expressions themselves. It should be clear by now that such rewritings are equally possible. For example, we might want to see the effect of shutting out certain parts of certain dataflows. We can express such queries using the same techniques.

6 Concluding remarks

We have shown how an integrated complex-object dataflow database, implemented on top of a modern SQL platform, enables answering diverse provenance queries. (We are currently developing a prototype.)

Of course, querying such a complex database requires expression in advanced SQL and XQuery, a skill we can expect from programmers working in an e-science team. Nevertheless, it would be nice if a domain-specific query language could be designed, for example, in the field of bioinformatics dataflows. Such a language should be more intuitive, possibly graphical, and usable by the scientists themselves, who are not trained as programmers. This is an interesting direction for further research. The challenge will be to find the right balance between expressive power and ease of use.

References

1. van der Aalst, W., van Hee, K.: Workflow Management. MIT Press (2004)
2. Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. 2nd edn. Elsevier (2004)
3. Shankar, S., et al.: Integrating databases and workflow systems. *SIGMOD Record* **34**(3) (2005) 5–11
4. Ludaescher, B., Goble, C., eds.: Special Section on Scientific Workflows. Volume 34(3) of *SIGMOD Record*. ACM (2005)
5. Brown, A.: Enforcing the scientific method. (Talk at IPAW'08, Salt Lake City, June 18th, 2008)
6. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: A formal model of dataflow repositories. 4th DILS. LNCS 4544, Springer (2007) 105–121
7. Provenance challenge Wiki. (<http://twiki.ipaw.info/bin/view/Challenge/>)
8. Moreau, L., Ludäscher, B., et al.: Special issue: The first provenance challenge. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 409–597
9. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. *Theoretical Computer Science* **149**(1) (1995) 3–48
10. Turi, D., Missier, P., Goble, C., et al.: Taverna workflows: Syntax and semantics. 3rd e-Science, IEEE Computer Society (2007) 441–448
11. Missier, P., Belhajjame, K., Zhao, J., Roos, M., Goble, C.: Data lineage model for Taverna workflows with lightweight annotation requirements. 2nd IPAW. LNCS 5272, Springer (2008) 17–30.
12. McPhillips, T., Bowers, S., Ludäscher, B.: Collection-oriented scientific workflows for integrating and analyzing biological data. 3rd DILS. LNCS 4075, Springer (2006) 247–263
13. Bowers, S., McPhillips, T., Ludäscher, B.: Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 519–529
14. Foster, I., Vöckler, J., Wilde, M., Zhao, Y.: Chimera: A virtual data system for representing, querying, and automating data derivation. 14th SSDBM, IEEE Computer Society (2002) 27–46
15. Clifford, B., Foster, I., et al.: Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 519–529
16. Chen, I., Markowitz, V.: An overview of the object protocol model (OPM) and the OPM data management tools. *Information Systems* **20**(5) (1995) 393–418
17. Ailamaki, A., Ioannidis, Y., Livy, M.: Scientific workflow management by database management. 10th SSDBM, IEEE Computer Society (1998) 190–199

18. Biton, O., Cohen Boulakia, S., Davidson, S.: Querying and managing provenance through user views in scientific workflows. 24th ICDE, IEEE Computer Society (2008) 1072–1081
19. Cohen Boulakia, S., S., Biton, O., Davidson, S.: Addressing the provenance challenge using ZOOM. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 497–506
20. Chebotko, A., Fei, X., Lin, C., Lu, S., Fotouhi, F.: Storing and querying scientific workflow provenance metadata using an RDBMS. 3rd e-Science, IEEE Computer Society (2007) 611–618
21. Van den Bussche, J., Vansummeren, S., Vossen, G.: Towards practical meta-querying. *Information Systems* **30**(4) (2005) 317–332
22. Van den Bussche, J., Vansummeren, S., Vossen, G.: Meta-SQL: Towards practical meta-querying. 9th EDBT. LNCS2992, Springer (2004) 823–825
23. van der Aalst, W., Reijers, H., Weijters, A., et al.: Business process mining: An industrial application. *Information Systems* **32**(5) (2007) 713–732
24. Santos, E., Lins, L., Ahrens, J., Freire, J., Silva, C.: A first study on clustering collections of workflow graphs. 2nd IPAW. LNCS5272, Springer (2008) 160–173
25. Ludäscher, B., Podhorszki, N., et al.: From computation models to models of provenance: the RWS approach. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 507–518
26. Zao, J., et al.: Mining Taverna’s semantic web of provenance. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 463–472
27. Barga, R., Digiampietri, L.: Automatic capture and efficient storage of e-science experiment provenance. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 419–429
28. Miles, S., et al.: Extracting causal graphs from an open provenance model. *Concurrency and Computation: Practice and Experience* **20**(5) (2008) 577–586
29. Kwasnikowska, N., Van den Bussche, J.: Mapping the NRC dataflow model to the open provenance model. 2nd IPAW. LNCS 5272, Springer (2008) 3–16
30. Moreau, L., et al.: The open provenance model. Technical Report 14979, University of Southampton, School of Electronics and Computer Science (2007)
31. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes with BP-QL. *Information Systems* **33**(6) (2008) 477–507
32. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: Monitoring business processes with queries. 33rd VLDB, ACM (2007) 603–614
33. Chen, J., Chung, S.Y., Wong, L.: The Kleisli query system as a backbone for bioinformatics data integration and analysis. In Lacroix, Z., Critchlow, T., eds.: *Bioinformatics: Managing Scientific Data*. Morgan Kaufmann (2003) 147–187
34. Davidson, S., Wong, L.: The Kleisli approach to data transformation and integration. In Gray, P., Kerschberg, L., King, P., Poulouvasilis, A., eds.: *The Functional Approach to Data Management*. Springer (2004) 135–165
35. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. 11th ICDT. LNCS 4353, Springer (2007) 209–223
36. Cheney, J., Ahmed, A., Acar, U.: Provenance as dependency analysis. 11th DBPL. LNCS 4797, Springer (2007) 138–152
37. Blockeel, H.: Experiment databases: A novel methodology for experimental research. 4th KDID. LNCS 3933, Springer (2005) 72–85
38. Blockeel, H., Vanschoren, J.: Experiment databases: Towards an improved experimental methodology in machine learning. 11th PKDD. LNCS 4702, Springer (2007) 6–17

- 39. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
- 40. Eisenberg, A., Melton, J.: Advancements in SQL/XML. SIGMOD Record **33**(3) (2004) 79–86
- 41. Özcan, F., Chamberlin, D., Kulkarni, K., Michels, J.E.: Integration of SQL and XQuery in IBM DB2. IBM Systems Journal **45**(2) (2006) 245–270