

Scientific Workflow Provenance Metadata Management Using an RDBMS-based RDF Store

(Technical Report TR-DB-092007-CFLF, September 2007)

Artem Chebotko, Xubo Fei, Shiyong Lu, and Farshad Fotouhi

Department of Computer Science
Wayne State University
5143 Cass Avenue, Detroit, Michigan 48202, USA
{artem, xubo, shiyong, fotouhi}@wayne.edu

Abstract. Provenance management has become increasingly important to support scientific discovery reproducibility, result interpretation, and problem diagnosis in scientific workflow environments. This paper proposes an approach to provenance management that seamlessly integrates the interoperability, extensibility, and reasoning advantages of Semantic Web technologies with the storage and querying power of an RDBMS. Specifically, we propose: i) two schema mapping algorithms to map an arbitrary OWL provenance ontology to a relational database schema that is optimized for common provenance queries; ii) three efficient data mapping algorithms to map provenance RDF metadata to relational data according to the generated relational database schema, and iii) a schema-independent SPARQL-to-SQL translation algorithm that is optimized on-the-fly by using the type information of an instance available from the input provenance ontology and the statistics of the sizes of the tables in the database. While the schema mapping and query translation and optimization algorithms are applicable to general RDF storage and query systems, the data mapping algorithms are optimized for and applicable only to scientific workflow provenance metadata. Moreover, we extend SPARQL with negation, aggregation, and set operations to support additional important provenance queries. Experimental results are presented to show that our algorithms are efficient and scalable. The comparison with existing RDF stores, Jena and Sesame, showed that our optimizations result in improved performance and scalability for provenance metadata management.

Keywords: provenance, scientific workflow, metadata management, ontology, RDF, SPARQL-to-SQL translation, query optimization, RDF store.

1 Introduction

Today, many significant scientific discoveries are achieved through complex and distributed scientific computations. More and more scientists start to use work-

flow technologies to automate the steps they need to go through from raw datasets to potential scientific discovery. As a result, scientific workflow has emerged as a new field to address the new requirements from scientists [41, 44]. A scientific workflow is a formal specification of a scientific process, which represents, streamlines, and automates the steps from dataset selection and integration, computation and analysis, to final data product presentation and visualization. A Scientific Workflow Management System (SWMS) supports the specification, execution, re-run, and monitoring of scientific processes [41, 46, 22, 31, 27, 61, 30].

Provenance management is essential for scientific workflows to support scientific discovery reproducibility, result interpretation, and problem diagnosis [54, 11]; such a facility is not necessary for business workflows. Provenance metadata captures the derivation history of a data product, including the original data sources, intermediate data products, and the steps that were applied to produce the data product. The provenance management problem concerns about the efficiency and effectiveness of the recording, representation, storage, querying, and visualization of provenance metadata.

Although a general RDBMS-based RDF store (see [8] for a survey) can be used for provenance metadata management, the following provenance-specific requirements bring about several optimization strategies for schema design, data mapping, and query mapping, enabling us to develop a provenance metadata management system that is more efficient and flexible than one that is simply based on existing RDF stores.

1. As provenance metadata is generated incrementally in the progress of workflow execution, an incremental optimization strategy can be developed for data mapping. Moreover, as workflow definition metadata is generated before workflow execution metadata, a join-elimination optimization strategy can be developed to employ this ordering relationship to achieve additional performance gain for data mapping.
2. As the performance for provenance storage and that for provenance querying are often conflicting, it would be desirable for a provenance management system to provide the flexibility for performing the trade-off between the two performance goals. For example, for long-running scientific workflows, trading off data mapping performance for querying performance might be a good strategy.
3. Provenance querying performance is very important to support efficient provenance browsing, visualization, and analysis. The identification of common provenance queries has the potential to lead to an optimized database schema design.
4. Update and delete are not the concern of provenance management since it works in an append fashion, similarly to log management. Therefore, one can apply some denormalization and redundancy strategies for database schema design, leading to improved query performance.

This paper proposes an approach to provenance management that seamlessly integrates the interoperability, extensibility, and reasoning advantages of Seman-

tic Web technologies with the storage and querying power of an RDBMS. The motivation of using Semantic Web technologies is threefold. First, large-scale e-science applications can span multiple domains and can involve global distributed workflows that consist of several heterogeneous local workflows orchestrated by different workflow engines, each of which has its own provenance manager [62]. The integration of these heterogeneous provenance systems is important for global provenance analysis. A Semantic Web approach promotes interoperability and facilitates such provenance integration. Second, RDF [3] is a property-centric, extremely flexible and dynamic data model, which captures well the dynamic and heterogeneous nature of data, services, and metadata in e-science applications. Finally, we can use the inference capability of Semantic Web for deriving metadata for various provenance dependency graphs [12].

The main contributions of this paper are: i) we propose two schema mapping algorithms to map an arbitrary OWL [1] provenance ontology to a relational database schema that is optimized for common provenance queries; ii) we propose three efficient data mapping algorithms to map provenance RDF [3] metadata to relational data according to the generated relational database schema, and iii) we propose a schema-independent SPARQL-to-SQL translation algorithm that is optimized on-the-fly by using the type information of an instance available from the input provenance ontology and the statistics of the sizes of the tables in the database. While the schema mapping and query translation and optimization algorithms are applicable to general RDF storage and query systems, the data mapping algorithms are optimized for and applicable only to scientific workflow provenance metadata. Moreover, we extend SPARQL [4] with negation, aggregation, and set operations to support additional important provenance queries. Experimental results are presented to show that our algorithms are efficient and scalable. The comparison with existing general-purpose RDF stores, Jena [59] and Sesame [14], showed that our optimizations result in improved performance and scalability for provenance metadata management.

This work extends our short workshop paper [17] in several directions. First, we provide a detailed description of our provenance ontology that is used by our VIEW system [18]. Second, we propose a new data mapping algorithm, `DataMapping-TM`, that enables faster data mapping than its peer `DataMapping-T` by eliminating relational joins. Third, we conduct a number of additional experiments, specifically, data mapping performance on sequences of stored provenance datasets, disk space consumption to store provenance over different database settings, and data mapping performance with and without database indexes. Fourth, we compare our data mapping and query evaluation to two existing RDF stores. Finally, we extend SPARQL with negation, aggregation, and set operations to support additional important provenance queries and conduct experiments with new SPARQL+ queries.

Organization. The rest of the paper is organized as follows. Provenance ontology to database schema mapping is introduced in Section 2. Provenance metadata to relational data mapping is presented in Section 3. SPARQL-to-SQL query

translation is presented in Section 4. Related work and discussion are presented in Section 5. Our conclusions and future work are discussed in Section 6.

2 Provenance Ontology to Database Schema Mapping

In this section, we propose two provenance ontology to database schema mapping algorithms using our developed provenance ontology PO as a running example.

2.1 A running example of provenance ontology

In this paper, a workflow consists of a set of workflow tasks, workflow inputs, workflow input parameters, workflow outputs, and data channels that connect them. Each task represents a computational or analytical step of a scientific process. A task has input ports and output ports that provide the communication interface to other tasks. Tasks are linked together into the workflow as an acyclic graph via data channels. During workflow execution, tasks communicate with each other by passing data via their ports through data channels. Finally, a task can have arbitrary number of input parameters, which are used by an e-scientist to configure its dynamic execution behavior.

We have developed a provenance ontology called PO that not only supports the semantic, syntactic, and structural description of workflows, tasks, and data objects, but also their execution instances. Our goal is to support queries across workflow definitions, workflow runs, and data objects. Currently PO includes over 30 classes and 40 properties and is used by our VIEW system [18]. Figure 1 illustrates an excerpt of PO which sketches concepts for workflow definition, workflow execution, workflow evolution, and definition-execution relationships:

- *Workflow definition* (see Figure 1(a)). It includes classes *Workflow*, *Task*, and *DataObject*. A *DataObject* models a data object passed to a particular data channel, it can be either a workflow input (relates to *Workflow* via property *input*), workflow output (relates to *Workflow* via property *output*), workflow input parameter (relates to *Workflow* via *inputParameter*), or an intermediate data product (relates to *Workflow* via *partOf*). In the meanwhile, a data object can either be a task input (relates to *Task* via property *input*), task output (relates to *Task* via *output*), or task input parameter (relates to *Task* via *inputParameter*).
- *Workflow execution* (see Figure 1(b)). It includes classes *WorkflowRun*, *TaskRun*, and *DataObjectRun*, which can be related to each other similarly via properties *input*, *output*, *inputParameter*, and *partOf*. Additionally, task runs can be organized into a task invocation dependency graph (*directTaskDependency* and *transitiveTaskDependency*), and data object runs – into a data dependency graph (*directDataDependency* and *transitiveDataDependency*).
- *Workflow evolution* (see Figure 1(c)). A workflow that evolved from another workflow can be related to its parent via properties *directWorkflowEvolution* and *transitiveWorkflowEvolution* to model a workflow evolution graph.

- *Definition-execution relationships.* The instances of execution classes are related to the instances of corresponding definition classes via property *instanceOf* to model that each workflow run executes a particular workflow, each task run correspond to an execution of a particular workflow task, and each data object run corresponds to a particular data object produced by a particular task run.

Although the definition and execution components look similar, their purposes are totally different. A workflow definition is prospective, representing the plan for processing; a workflow run is retrospective, representing an actual execution of the workflow, which can be conditional or iterative. In addition, a workflow can be executed multiple times, each of which corresponds to a different workflow run, while the workflow definition stays the same (workflow evolution is considered as a different workflow definition).

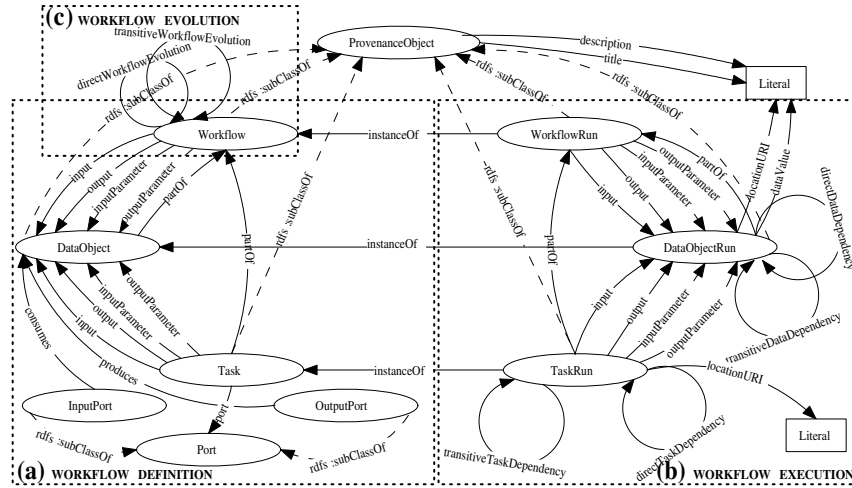


Fig. 1. An excerpt of the provenance ontology

Consider a sample workflow in Figure 2(a). It consists of three tasks, two workflow inputs, one workflow input parameter, and two workflow outputs. An RDF graph that describes this workflow is drawn in Figure 2(b), and an execution of this workflow produces the RDF graph in Figure 2(c). Both RDF graphs are stored into our provenance database.

An important advantage of using a formal ontology is the ability to derive new RDF descriptions using semantic inference. Out of two inference support mechanisms, *forward-chaining*, in which all inferences are precomputed and stored, and *backward-chaining*, in which inferences are computed dynamically for each query, we adopt the first one, which can be efficiently implemented outside of an RDBMS. We support: (1) *OWL semantics inference*, which uses A-Box inference

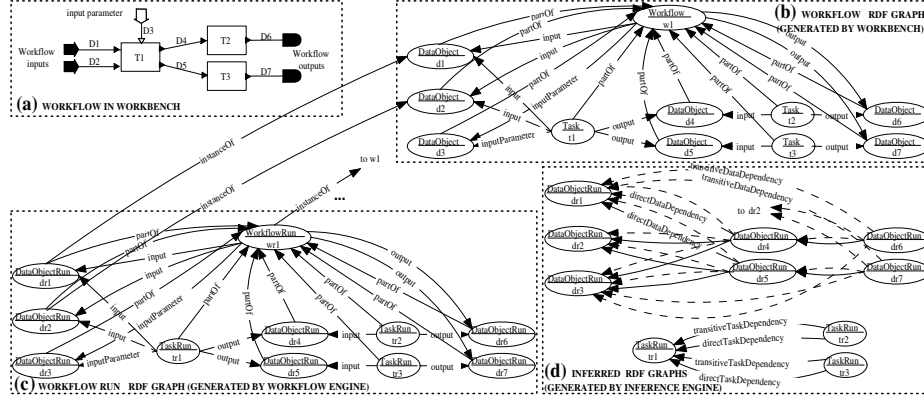


Fig. 2. A sample workflow and its RDF graphs

rules for OWL constructs, such as *rdfs:subClassOf*, *owl:TransitiveProperty*, and *owl:SymmetricProperty*, and (2) *Provenance dependency graph inference*, which uses our rules to derive various provenance graphs. We use a simple language to define inference rules, such that an antecedent and a consequent of a rule are specified as SPARQL basic graph patterns. If the antecedent matches triples in an RDF graph, then obtained variable bindings are used in the consequent to infer new RDF triples that are appended to the RDF graph. For example, our rule for deriving a data dependency graph is as follows.

$$\begin{array}{c}
 ?tr \text{ rdf:type } \textit{TaskRun} . \text{ ?tr input ?d1 . ?tr output ?d2 .} \\
 \text{ ?d1 rdf:type } \textit{DataObjectRun} . \text{ ?d2 rdf:type } \textit{DataObjectRun} . \\
 \hline
 \text{ ?d2 directDataDependency ?d1 . ?d2 transitiveDataDependency ?d1 .}
 \end{array}$$

In this rule, if task run *?tr* has input data object *?d1* and output data object *?d2*, then two triples, *?d2 directDataDependency ?d1* and *?d2 transitiveDataDependency ?d1*, are inferred, where *?d1* and *?d2* are substituted with their corresponding bindings, and the *transitiveDataDependency* property is defined as *owl:TransitiveProperty*, such that

$$\begin{array}{c}
 \text{ ?tr3 transitiveDataDependency ?tr2 . ?tr2 transitiveDataDependency ?tr1 .} \\
 \hline
 \text{ ?tr3 transitiveDataDependency ?tr1 .}
 \end{array}$$

Task invocation dependency graph can be inferred based on a similar rule:

$$\begin{array}{c}
 \text{ ?tr1 rdf:type } \textit{TaskRun} . \text{ ?tr2 rdf:type } \textit{TaskRun} . \\
 \text{ ?tr1 output ?d . ?tr2 input ?d .} \\
 \hline
 \text{ ?tr2 directTaskDependency ?tr1 . ?tr2 transitiveTaskDependency ?tr1 .}
 \end{array}$$

Finally, workflow evolution graph can be inferred using even a simpler rule, because the *directWorkflowEvolution* relationship between two different workflows is given in a provenance dataset. Then,

$$\frac{?w2 \text{ directWorkflowEvolution } ?w1 .}{?w2 \text{ transitiveWorkflowEvolution } ?w1 .}$$

For example, given the workflow run RDF graph in Figure 2(c), data dependency and task invocation dependency graphs are shown in Figure 2(d). These RDF graphs, as well as the workflow evolution graph, are precomputed by an inference engine and stored into our provenance database.

2.2 Schema mapping algorithms

Database schema design is of decisive importance to support efficient processing of provenance queries. In general, generating an optimal schema for a given set of queries under certain time and space efficiency constraints is a hard problem. Our schema design aims at supporting efficient processing of the following queries that are common in provenance retrieval and browsing: (1) retrieve RDF graph nodes of a given type, such as *Workflow*, *WorkflowRun*, and *Task*, (2) retrieve all the immediate neighboring nodes of a given node reachable through incoming or outgoing edges, (3) retrieve nodes that are directly related to a given node by some properties, such as *input*, *output*, and *partOf*, and (4) a combination of the above queries.

Our two alternative schema mapping algorithms are presented in Figure 3. **SchemaMapping-V** generates one table and four kinds of views from a given ontology with class-set \mathcal{C} and property-set \mathcal{P} : (1) a single table $Triple(s,p,o)$ that stores all RDF triples in the database, (2) for each $\$c \in \mathcal{C}$, a view $\$c(i)$ that captures all instances of class $\$c$, (3) for each $\$c \in \mathcal{C}$, a view $\$cSubject(i,p,o)$ that captures all triples whose subjects are instances of class $\$c$, (4) for each $\$c \in \mathcal{C}$, a view $\$cObject(s,p,i)$ that captures all triples whose objects are instances of class $\$c$, and (5) for each $\$p \in \mathcal{P}$, a view $\$p(s,o)$ that captures all instances of property $\$p$. **SchemaMapping-T** behaves similarly, but materializes all views as tables. The total number of generated relations (tables and views) is $1 + 3 \times |\mathcal{C}| + |\mathcal{P}|$.

In addition, we make extensive use of database indexes. For the **SchemaMapping-V** schema, we create indexes on columns (s,p,o), (s,o), (o,p), and (p) of table $Triple(s,p,o)$. For the **SchemaMapping-T** schema, we create similar indexes on columns of $Triple(s,p,o)$, $\$cSubject(i,p,o)$, and $\$cObject(s,p,i)$, indexes on columns (s,o) and (o) of $\$p(s,o)$, and single-column indexes on tables $\$c(i)$. Finally, each table has the uniqueness constraint on the tuples it stores.

The last, but not least, we consider the problem of database schema and instance data change in the context of ontology evolution. In Table 1, we outline the effects on database schemas **SchemaMapping-V** and **SchemaMapping-T** and corresponding instance data for three basic ontology-change operations, namely *add*, *delete*, and *rename*, that can be applied to classes and properties in the ontology. Consider a class/property hierarchy represented by a tree, whose nodes

```

01 Algorithm SchemaMapping-V
02 Input: ontology with class-set  $\mathcal{C}$  and property-set  $\mathcal{P}$ 
03 Output: relational database schema
04 Begin
05 Create table  $Triple(s,p,o)$ 
06 For each class  $\$c \in \mathcal{C}$  do
07   Create view  $\$c(i) = \text{Select } s \text{ From } Triple \text{ Where } p='rdf:type' \text{ And } o='\$c'$ 
08   Create view  $\$cSubject(i,p,o) = \text{Select } s,p,o \text{ From } Triple, \$c \text{ Where } s=i$ 
09   Create view  $\$cObject(s,p,i) = \text{Select } s,p,o \text{ From } Triple, \$c \text{ Where } o=i$ 
10 End For
11 For each property  $\$p \in \mathcal{P}$  do
12   Create view  $\$p(s,o) = \text{Select } s,o \text{ From } Triple \text{ Where } p='\$p'$  End For
13 End Algorithm

14 Algorithm SchemaMapping-T
15 Input: ontology with class-set  $\mathcal{C}$  and property-set  $\mathcal{P}$ 
16 Output: relational database schema
17 Begin
18 Create table  $Triple(s,p,o)$ 
19 For each class  $\$c \in \mathcal{C}$  do
20   Create table  $\$c(i)$ 
21   Create table  $\$cSubject(i,p,o)$ 
22   Create table  $\$cObject(s,p,i)$ 
23 End For
24 For each property  $\$p \in \mathcal{P}$  do Create table  $\$p(s,o)$  End For
25 End Algorithm

```

Fig. 3. Algorithms SchemaMapping-V and SchemaMapping-T

are classes/properties and edges are subclass/subproperty relationships. A new node can be added into such a tree as (1) a leaf, (2) a root, or (3) an inner node, such that it becomes a child of a node n_1 and a parent of another node n_2 , where n_1 is originally a parent of n_2 . Similarly, an existing node can be deleted from the tree at the three positions; its children become children of its parent; if the root is deleted, its each child becomes the root of an independent hierarchy. The renaming operation does not affect the tree structure. As shown in Table 1, these operations can be easily supported on the database schema level. The effect on instance data can also be implemented quite straightforwardly, but is computationally more expensive, especially for the SchemaMapping-T schema, whose materialized views must be maintained to be consistent. Some other operations, such as merge classes and split a class in several classes, can also be supported, but they are out the scope of this paper.

2.3 Experimental study

All the experiments reported in this paper were conducted on a PC with 2.4 GHz Pentium IV CPU and 1024 MB of main memory operated by MS Windows XP Professional. All algorithms were implemented in C/C++ and MySQL 5.0 Community Edition was employed as the RDBMS. Our developed provenance server communicated with MySQL using the MySQL C API.

The performance of our schema mapping algorithms on PO is presented in Table 2. The reported times include the time required to process the ontology.

Table 1. Ontology-change operations and their effects on database schemas SchemaMapping-V and SchemaMapping-T and instance data

Operation	Effect on schema SchemaMapping-V	Effect on instance data (after forward-chaining inference)
Add class $\$c$	Create views $\$c$, $\$cSubject$, and $\$cObject$	If $\$c'$ is a subclass of $\$c$, then instances of $\$c'$ must be inferred as instances of $\$c$ in table $Triple$
Delete class $\$c$	Drop views $\$c$, $\$cSubject$, and $\$cObject$	Delete tuples in table $Triple$ that define instances of $\$c$. Instances of $\$c$ remain instances of the superclass of $\$c$
Rename class $\$c$ into $\$c'$	Drop corresponding views for $\$c$ and create views for $\$c'$	Update tuples in table $Triple$ that define instances of $\$c$ to become instances of $\$c'$
Add property $\$p$	Create view $\$p$	If $\$p'$ is a subproperty of $\$p$, then instances of $\$p'$ must be inferred as instances of $\$p$ in table $Triple$
Delete property $\$p$	Drop view $\$p$	Delete tuples in table $Triple$ that define instances of $\$p$. Instances of $\$p$ remain instances of the superproperty of $\$p$
Rename property $\$p$ into $\$p'$	Drop view $\$p$ and create view $\$p'$	Update tuples in table $Triple$ that define instances of $\$p$ to become instances of $\$p'$

Operation	Effect on schema SchemaMapping-T	Effect on instance data (after forward-chaining inference)
Add class $\$c$	Create tables $\$c$, $\$cSubject$, and $\$cObject$	If $\$c'$ is a subclass of $\$c$, then instances of $\$c'$ must be inferred as instances of $\$c$ in tables $Triple$ and $type$. Compute tuples for the new tables (see Section 3)
Delete class $\$c$	Drop tables $\$c$, $\$cSubject$, and $\$cObject$	Delete tuples in tables $Triple$ and $type$ that define instances of $\$c$. Instances of $\$c$ remain instances of the superclass of $\$c$
Rename class $\$c$ into $\$c'$	Rename tables for $\$c$ into $\$c'$, $\$c' Subject$, and $\$c' Object$	Update tuples in tables $Triple$ and $type$ that define instances of $\$c$ to become instances of $\$c'$
Add property $\$p$	Create table $\$p$	If $\$p'$ is a subproperty of $\$p$, then instances of $\$p'$ must be inferred as instances of $\$p$ in tables $Triple$ and $\$p$
Delete property $\$p$	Drop table $\$p$	Delete tuples in tables $Triple$, $\$cSubject$, and $\$cObject$ (for each class $\$c$) that define instances of $\$p$. Instances of $\$p$ remain instances of the superproperty of $\$p$
Rename property $\$p$ into $\$p'$	Rename table $\$p$ into $\$p'$	Update tuples in tables $Triple$, $\$cSubject$, and $\$cObject$ (for each class $\$c$) that define instances of $\$p$ to become instances of $\$p'$

In our approach, the schema mapping is only required to be performed once to store multiple provenance datasets.

3 Provenance Metadata to Relational Data Mapping

3.1 Data mapping algorithms

In this section, we present three data mapping algorithms that insert a new provenance dataset D , either of a workflow definition, a workflow run, or a workflow provenance dependency graph, into the database. The DataMapping-V algorithm that corresponds to SchemaMapping-V is trivial as all we need to do is to insert D into table $Triple$. For the database schema created by SchemaMapping-T, table $Triple$ can be populated similarly, i.e., by simply inserting D into $Triple$.

Table 2. Performance of SchemaMapping-V and SchemaMapping-T on PO

Algorithm	# of tables created	# of views created	# of indexes created	Time (s)
SchemaMapping-V	1	134	4	4.437
SchemaMapping-T	135	0	365	53.641

Let $Triple'$ be a temporary table for storing the triples of D . New tuples for $\$c$ can be calculated by $\$c'(i) \leftarrow \text{Select } s \text{ From } Triple' \text{ Where } p='rdf:type' \text{ And } o='\c' , and new tuples for $\$p$ can be calculated by $\$p'(s,o) \leftarrow \text{Select } s,o \text{ From } Triple' \text{ Where } p='\p' . The question is how we can calculate new tuples for tables $\$cSubject$ and $\$cObject$ for each class $\$c \in \mathcal{C}$.

One strategy, called *brute-force*, is to calculate $Triple'$, $\$c'$, and $\$p'$ and insert them into tables $Triple$, $\$c$, and $\$p$, respectively. Then, delete contents of $\$cSubject$ and $\$cObject$ and rematerialize these two tables as follows: $\$cSubject(i,p,o) \leftarrow \text{Select } s,p,o \text{ From } Triple, \$c \text{ Where } s=i$ and $\$cObject(s,p,i) \leftarrow \text{Select } s,p,o \text{ From } Triple, \$c \text{ Where } o=i$. However, this strategy is expensive since we have to recompute joins of $Triple$ and $\$c$, whose sizes are growing over time.

A better strategy, called *incremental*, calculates new tuples in $\$cSubject'$ and $\$cObject'$ and then inserts them into $\$cSubject$ and $\$cObject$, respectively. $\$cSubject'$ and $\$cObject'$ are calculated as follows: $\$cSubject'(i,p,o) \leftarrow (\text{Select } s,p,o \text{ From } Triple', \$c' \text{ Where } s=i) \text{ Union } (\text{Select } s,p,o \text{ From } Triple, \$c' \text{ Where } s=i) \text{ Union } (\text{Select } s,p,o \text{ From } Triple', \$c \text{ Where } s=i)$ and $\$cObject'(s,p,i) \leftarrow (\text{Select } s,p,o \text{ From } Triple', \$c' \text{ Where } o=i) \text{ Union } (\text{Select } s,p,o \text{ From } Triple, \$c' \text{ Where } o=i) \text{ Union } (\text{Select } s,p,o \text{ From } Triple', \$c \text{ Where } o=i)$. In other words, we need to compute unions of three joins: (1) $Triple' \bowtie \$c'$, (2) $Triple \bowtie \$c'$, and (3) $Triple' \bowtie \$c$. In contrast to the brute-force strategy, the incremental strategy requires to compute joins of smaller tables.

Next, our *optimized incremental* strategy is similar to the incremental one, except that we do not need to compute the join $Triple \bowtie \$c'$ when populating $\$cSubject'$ and $\$cObject'$. This simplification is possible because provenance datasets are stored in *order*, such that a workflow definition is stored at first, its workflow run is stored at second, and its provenance graphs are stored last. As a result, a to-be-stored dataset D may have an instance X whose type ($X.rdf:type$ class) is not defined in D , but is defined in the triple-set stored in the database; the other way around can never be true. Therefore, the join of $Triple$ and $\$c'$ can return only tuples that are already in the database in $\$cSubject$ or in $\$cObject$. Furthermore, since we left with only two required joins $Triple' \bowtie \$c'$ and $Triple' \bowtie \$c$, we can replace them by $Triple' \bowtie (\$c' \cup \$c)$ or by inserting $\$c'$ into $\$c$ and computing $Triple' \bowtie \$c$. This strategy substantially reduces the number of join operations.

Finally, our *optimized incremental in-memory* strategy completely eliminates relational joins by calculating new tuples for $Triple$, $\$c$, $\$cSubject$, $\$cObject$, and $\$p$ in main memory outside of the database engine. To achieve this, we introduce the notion of *type dictionary* – an in-memory data structure that, given an instance URI, returns a set of types of this instance. We model the

```

01 Algorithm DataMapping-T
02 Input: RDF dataset  $D$ , class-set  $\mathcal{C}$ , and property-set  $\mathcal{P}$ 
03 Output: database populated with relational tuples
04 Begin
05 Let  $Triple'(s,p,o)$  be a temporary table
06 Parse  $D$  and bulkload triples into  $Triple'(s,p,o)$ 
07 Insert into  $Triple(s,p,o) \leftarrow \text{Select } s,p,o \text{ From } Triple'(s,p,o)$ 
08 For each class  $\$c \in \mathcal{C}$  do
09   Insert into  $\$c(i) \leftarrow \text{Select } s \text{ From } Triple' \text{ Where } p='rdf:type' \text{ And } o='\$c'$ 
10   Insert into  $\$cSubject(i,p,o) \leftarrow (\text{Select } s,p,o \text{ From } Triple', \$c \text{ Where } s=i)$ 
11   Insert into  $\$cObject(s,p,i) \leftarrow (\text{Select } s,p,o \text{ From } Triple', \$c \text{ Where } o=i)$ 
12 End For
13 For each property  $\$p \in \mathcal{P}$  do
14   Insert into  $\$p(s,o) \leftarrow \text{Select } s,o \text{ From } Triple' \text{ Where } p='\$p'$ 
15 End For
16 Delete all tuples from  $Triple'$ 
17 End Algorithm

```

Fig. 4. Algorithm DataMapping-T

type dictionary T as a hash table whose keys are strings that represent URIs and values are sets of strings that represent classes in the ontology. If a key k is not found in T , then $T[k] = \emptyset$. Initially, T is retrieved from table $type(s,o)$ which stores instances in column s and their types in column o and, once in memory, T is maintained synchronously with table $type$. Given T and a new provenance dataset D , this strategy scans D twice. First, to update T with new instances and types found in D . Second, to calculate new tuples for the database tables as follows. Let $Triple'$, $\$c'$, $\$cSubject'$, $\$cObject'$, and $\$p'$ be empty tuple-sets, for each class $\$c \in \mathcal{C}$ and property $\$p \in \mathcal{P}$ in the ontology. For each triple t ($t.sub, t.pre, t.obj$) in D , (1) t is added to $Triple'$, (2) $t.sub$ is added to $\$c'$ if $t.pre=rdf:type$ and $t.obj = \$c$, (3) t is added to $\$cSubject'$ for each $\$c \in T[t.sub]$, (4) t is added to $\$cObject'$ for $\$c \in T[t.obj]$, and (5) $(t.sub, t.obj)$ is added to $\$p'$ if $t.pre = \$p$. After the tuple-sets are computed, they are inserted into the corresponding tables in the database. This strategy has a time complexity of $O(D)$.

Figures 4 and 5 define algorithms DataMapping-T and DataMapping-TM that implement the optimized incremental and optimized incremental in-memory strategies, respectively.

Note that the following three properties regarding the cardinalities of relations always hold: (1) $|Triple| \geq |\$cSubject| \geq |\$c|$, (2) $|Triple| \geq |\$cObject| \geq |\$c|$, and (3) $|Triple| \geq |\$p|$. This information can be used to select the smallest relation for query optimization. In addition, we cache cardinality statistics for relations $\$p$, $\$cSubject$, and $\$cObject$.

Figure 6 shows some of the relations generated and loaded with tuples that correspond to the workflow RDF graph in Figure 2(b).

3.2 Experimental study

Before data mapping is performed, each RDF dataset is preprocessed by our inference engine implemented outside of the relational database. The inference en-

```

01 Algorithm DataMapping-TM
02 Input: RDF dataset  $D$ , class-set  $C$ , property-set  $\mathcal{P}$ , and type dictionary  $T$ 
03 Output: database populated with relational tuples
04 Begin
05 For each triple  $t (t.sub, t.pre, t.obj)$  in  $D$  do
06   If  $t.pre = rdf:type$  then  $T[t.sub] \cup = \{t.obj\}$  End If
07 End For
08 Let  $Triple'$  be an empty tuple-set
09 Let  $\$c', \$cSubject', \$cObject'$ , and  $\$p'$  be empty tuple-sets, for each  $\$c \in C$  and  $\$p \in \mathcal{P}$ 
10 For each triple  $t (t.sub, t.pre, t.obj)$  in  $D$  do
11    $Triple' \cup = \{(t.sub, t.pre, t.obj)\}$ 
12   If  $t.pre = rdf:type$  then Let  $\$c = t.obj$ ,  $\$c' \cup = \{(t.sub)\}$  End If
13   For each class  $\$c \in T[t.sub]$ ,  $\$cSubject' \cup = \{(t.sub, t.pre, t.obj)\}$ 
14   For each class  $\$c \in T[t.obj]$ ,  $\$cObject' \cup = \{(t.sub, t.pre, t.obj)\}$ 
15   Let  $\$p = t.pre$ ,  $\$p' \cup = \{(t.sub, t.obj)\}$ 
16 End For
17 Bulkload into  $Triple(s,p,o) \leftarrow Triple'$ 
18 For each  $\$c \in C$  and  $\$p \in \mathcal{P}$  do
19   Bulkload into  $\$c(i) \leftarrow \$c'$ 
20   Bulkload into  $\$cSubject(i,p,o) \leftarrow \$cSubject'$ 
21   Bulkload into  $\$cObject(s,p,i) \leftarrow \$cObject'$ 
22   Bulkload into  $\$p(s,o) \leftarrow \$p'$ 
23 End For
24 End Algorithm

```

Fig. 5. Algorithm DataMapping-TM

(a) a single relation			(b) relations of type $\$c$			(c) relations $\$cObject$ and $\$cSubject$						(d) relations of type $\$p$												
Triple	s	p	o	Workflow	DataObject	Task	WorkflowSubject	s	p	o	WorkflowObject	s	o	partOf	input	output	inputParameter	s	o	s	o	s	o	
w1	rdf:type	Workflow		w1			w1	rdf:type	Workflow	d1	partOf	w1	w1	Workflow	d1	w1	w1	d1	w1	d6	w1	d3		
d1	rdf:type	DataObject			d1	t1	w1	input	d1	...	partOf	w1	d1	DataObject	d2	w1	w1	d2	w1	d7	t1	d3		
d1	partOf	w1			d2	t2	w1	input	d2	d7	partOf	w1	t1	d1	t1	d1	t1	d4				
w1	input	d1			...	t3	w1	inputParameter	d3	t1	partOf	w1	d7	DataObject	d7	w1	t1	d2	t1	d5				
t1	rdf:type	Task					w1	output	d6	t2	partOf	w1	t1	Task	t1	w1	t2	d4	t2	d6				
t1	input	d1			d7		w1	output	d7	t3	partOf	w1	t2	Task	t2	w1	t3	d5	t3	d7				
...					w1	output					t3	Task	t3	w1								

Fig. 6. A workflow RDF graph stored into a relational database

gine showed the performance of less than 0.1s when evaluated on sample datasets of workflow definitions with 10 tasks and workflow runs with less than 400 triples. The number of inferred triples constituted about 30% of original datasets. In the rest of our experiments, we use RDF datasets that have already been appended with inferred triples and thus reported times do not include time spent on inference.

First, algorithms DataMapping-V, DataMapping-T, and DataMapping-TM were evaluated to store sequences of workflow runs into the databases with the corresponding schemas. In particular, we stored five workflow definitions into the database and measured the times to store sequences of 1, 20, 200, 2,000, and 20,000 workflow runs. Each workflow run provenance contained 500 triples in the N-Triples [2] format. The results are shown in Figure 7, which presents two different views of the same experimental results. The algorithms showed good performance and scalability, in particular, DataMapping-V showed to be much faster than its two peers since it only populated one table. On the other hand, DataMapping-TM benefited from our in-memory data mapping strategy

and showed to be significantly faster than **DataMapping-T**. In addition, we compared our algorithms to data mapping in general-purpose RDF stores Jena 2.5.2 and Sesame 1.2.6 with the MySQL backend and inference turned off; Sesame 2 did not support a database backend at the time of comparison. Jena [59] used one table to store all triples similar to our *Triple* table, however subject, predicate, and object values were encoded, which resulted in worse performance than **DataMapping-V** and better performance than all the other approaches. Sesame [14] also used one table *Triples* to store all triples, however URIs and literals were substituted with integer IDs. The mappings from IDs to URIs and literals were stored in tables *Resources* and *Literals*, respectively. This design minimized data redundancy in the database and facilitated faster indexes on numeric values. However, the maintenance of these mappings showed to be expensive. Sesame showed the worst performance in this experiment and revealed non-linear scalability in contrast to other approaches.

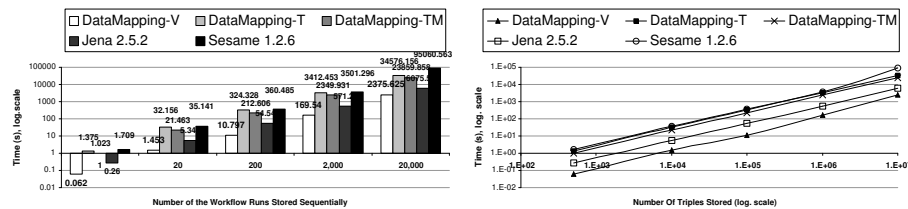


Fig. 7. Performance of **DataMapping-V**, **DataMapping-T**, **DataMapping-TM**, Jena 2.5.2, and Sesame 1.2.6 on sequences of workflow runs

Second, the algorithms were evaluated to store a single workflow run into the database for varying number of workflow runs already stored in the database. In particular, we stored five workflow definitions into the database and measured the times to store 1st, 3rd, 21st, 201st, 2,001st, and 20,001st workflow runs. The results are shown in Figure 8, where we report performance for “cold” runs with the MySQL server restarted before a trial and for “warm” runs over MySQL with “warm” cache. In both cases, **DataMapping-V** was the fastest and Jena showed the second best performance. **DataMapping-TM** was the slowest for the “cold” trials, except for 20,001st workflow run when Sesame was slower, and Sesame was the slowest for the “warm” trials. The performance of all the approaches, except for Sesame, showed to be efficient and scalable when we ran them over MySQL with “warm” cache, which is the most probable situation for real-life settings. Even though **DataMapping-V** was approximately 10 times faster than **DataMapping-T** and **DataMapping-TM**, it is reasonable to expect that a workflow environment is unlikely to produce provenance metadata with such a high rate, i.e., 500 triples in every 0.1s. **DataMapping-V** showed stable performance of about 0.1s per workflow run, Jena – about 0.2 – 0.3s per workflow run, **DataMapping-**

TM – about 1.0 – 1.2s per workflow run, DataMapping-T – about 1.3 – 1.7s per workflow run, and Sesame performed in the range of 1.7 – 5.4s per workflow run.

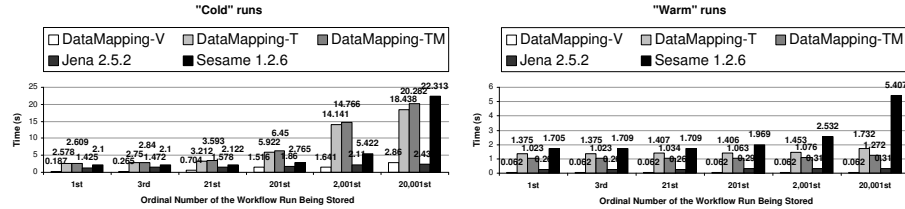


Fig. 8. Performance of DataMapping-V, DataMapping-T, DataMapping-TM, Jena 2.5.2, and Sesame 1.2.6 on a single workflow run

Third, in Figure 9, we report disk space required to store 1, 20, 200, 2,000, and 20,000 workflow runs for the database schemas generated by SchemaMapping-V, SchemaMapping-T, Jena, and Sesame. In the figure, the data length corresponds to the length of all tables in the database and the data and index length corresponds to the length of all tables and indexes in the database. Since SchemaMapping-T generated a number of materialized views, it required approximately three times larger disk space than SchemaMapping-V to store the same data. Jena and Sesame required less space than SchemaMapping-T and more space than SchemaMapping-V for our datasets. The indexes consumed much larger space than data. Overall, the data and index length for SchemaMapping-T was approximately two times larger the data and index length for SchemaMapping-V for the same datasets. Due to the extensive use of indexes, both SchemaMapping-V and SchemaMapping-T consumed more disk space than Jena and Sesame.

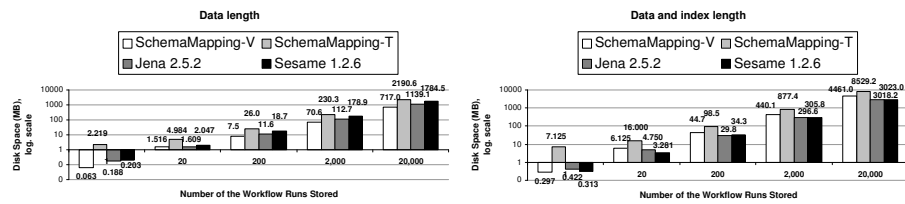


Fig. 9. Disk space required to store workflow runs in the MySQL database over schemas SchemaMapping-V, SchemaMapping-T, Jena 2.5.2, and Sesame 1.2.6

Finally, we explored the performance of our algorithms on the database with no indexes created. Such settings can be used to achieve even faster data mapping of large provenance datasets, while the indexes can be created later for

efficient query evaluation. SchemaMapping-V showed constant performance of 0.022s and 0.16s per workflow run for “warm” and “cold” trials, respectively. SchemaMapping-TM showed nearly constant performance of about 1.0s and 2.5s per workflow run for “warm” and “cold” trials, respectively. SchemaMapping-T appeared to be unsuitable for this purpose. Its performance degraded quickly with the database growth, since it required to compute relational joins, whose performance greatly depended on the availability of indexes.

4 SPARQL-to-SQL Query Translation

4.1 Query translation algorithm

In our approach, SPARQL is the primary language for provenance querying. SPARQL is used to specify queries with basic, group, optional, and alternative graph patterns that are matched over RDF provenance graphs. For example, the following SPARQL query returns information that describes workflows that require user input (have input parameters):

```
Select ?w ?p ?o
Where {?w rdf:type :Workflow . ?w ?p ?o . ?w :inputParameter ?x}
```

In the `Select` clause, it specifies three variables `?w` for the workflow, `?p` for the predicate, and `?o` for the object, whose instantiations must be returned. In the `Where` clause, the query has the basic graph pattern consisting of three triple patterns: `?w rdf:type :Workflow` to match instances of the *Workflow* type, `?w ?p ?o` to match related instances, and `?w :inputParameter ?x` to ensure that a workflow has at least one input parameter. To evaluate such a query over our relational provenance database, we design algorithms to translate a SPARQL query into an equivalent SQL query.

We first consider the problem of matching a triple pattern tp ($tp.sub$, $tp.pre$, $tp.obj$) against the database. Two questions need to be answered: (1) Which relation should be used for tp ? (2) Which relational attributes should be used for $tp.sub$, $tp.pre$, and $tp.obj$? These questions are formulated in terms of two mapping functions, ρ and α , such that $\rho(tp)$ returns the relation that stores all the triples matching tp , while $\alpha(tp, sub)$, $\alpha(tp, pre)$, and $\alpha(tp, obj)$ return the corresponding relational attributes. The two mapping functions provide a foundation for schema-independent SPARQL-to-SQL translation, such that the relational schema design, which concerns about ρ and α , is fully separated from the translation algorithm that is parameterized by ρ and α . In addition, such an abstraction enables schema design optimization and query optimization. We will not pursue the schema design optimization issue in this paper, but illustrate two query optimization techniques below by using the type information of an instance.

The first optimization is based on the following two questions: (1) Given an instance or variable X in a triple pattern tp from a SPARQL query q , can we determine the type of X based on q ? (2) If the answer to the first question is yes, let $\tau(X)$ be the type of X , then which relation should be chosen for $\rho(tp)$

among $\tau(X)$, $\tau(X)Subject$, $\tau(X)Object$, or *Triple*? Intuitively, we like to choose the relation with the smallest number of tuples.

For the first question, given an instance or variable X in a triple pattern tp from a SPARQL query q and our provenance ontology PO, $\tau(X)$ can be decided as follows:

$$\tau(X) = \begin{cases} c & \text{If } tp \text{ is of the form } X \text{ rdf:type } :c, \text{ where } c \in \mathcal{C} \text{ is a class in PO;} \\ p & \text{If } tp.pre = :p \text{ and } X = :p, \text{ where } p \in \mathcal{P} \text{ is a property in PO;} \\ c & \text{If } (\bigcap_{p_d \in q} domain(p_d)) \cap (\bigcap_{p_r \in q} range(p_r)) = \{ :c \}, \\ & \text{where } p_d \text{ and } p_r \text{ are property instances in some triple patterns} \\ & tp_d \text{ and } tp_r \text{ from } q, \text{ such that } tp_d.sub = X, tp_d.pre = p_d, \\ & tp_r.obj = X, tp_r.pre = p_r, \text{ and } p_r \text{ is of type } owl:ObjectProperty; \\ undef & \text{otherwise.} \end{cases}$$

In other words, $\tau(X)$ is defined if the type of X is explicitly stated in q via the *rdf:type* property, or X is a property instance, or it is computed as the intersection of the domain and range sets of all the properties that are predicates in q 's triple patterns and X is a subject and object, respectively, in these triple patterns. If the result of the intersection is a set with one ontology class, then $\tau(X)$ is defined; otherwise, $\tau(X)$ is undefined.

For our sample query, $\tau(?w) = Workflow$ as stated in the first triple pattern, $\tau(rdf:type) = type$, and $\tau(:inputParameter) = inputParameter$. The other instances and variables have undefined value of τ ; in particular, $\tau(?x)$ can not be computed because the range of *inputParameter* contains several classes.

The answer to the second question of choosing the best relation for $\rho(tp)$ is described in Figure 10. Algorithm Calculate- ρ - α is used to compute ρ and α for each tp in a SPARQL query, such that $\rho(tp)$ is assigned the smallest relation and α values are directly decided by the relation schema of $\rho(tp)$. The smallest relation is identified using the *min* function (line 08). When $\tau(tp.sub)$, $\tau(tp.obj)$, or $\tau(tp.pre)$ is undefined, we assign $|\tau(tp.sub)Subject|=+\infty$, $|\tau(tp.obj)Object|=+\infty$, or $|\tau(tp.pre)|=+\infty$, respectively.

```

01 Algorithm Calculate- $\rho$ - $\alpha$ 
02 Input:  $tp, \tau$ 
03 Output:  $\rho, \alpha$ 
04 Begin
05 If  $tp.pre = rdf:type$  and  $tp.obj$  is an ontology class  $c$  then
06   /*Case 1*/  $\rho(tp) = c, \alpha(tp, sub) = i, \alpha(tp, pre) = undef, \alpha(tp, obj) = undef$ 
07 Else
08   Switch [ $\min(|\tau(tp.sub)Subject|, |\tau(tp.obj)Object|, |\tau(tp.pre)|, |Triple|)$ ]
09     Case 2  $|\tau(tp.sub)Subject|$ :  $\rho(tp) = \tau(tp.sub)Subject, \alpha(tp, sub) = i, \alpha(tp, pre) = p, \alpha(tp, obj) = o$ 
10     Case 3  $|\tau(tp.obj)Object|$ :  $\rho(tp) = \tau(tp.obj)Object, \alpha(tp, sub) = s, \alpha(tp, pre) = p, \alpha(tp, obj) = i$ 
11     Case 4  $|\tau(tp.pre)|$ :  $\rho(tp) = \tau(tp.pre), \alpha(tp, sub) = s, \alpha(tp, pre) = undef, \alpha(tp, obj) = o$ 
12     Case 5  $|Triple|$ :  $\rho(tp) = Triple, \alpha(tp, sub) = s, \alpha(tp, pre) = p, \alpha(tp, obj) = o$ 
13   End Switch
14 End If
15 Return  $\rho, \alpha$ 
16 End Algorithm

```

Fig. 10. Algorithm Calculate- ρ - α

For our sample query, we have $\rho(?w \text{ rdf:type } :Workflow) = Workflow$ (Case 1), $\rho(?w ?p ?o) = WorkflowSubject$ (Case 2) because $\tau(?p)$ and $\tau(?o)$ are undefined and $|WorkflowSubject| \leq |Triple|$, and $\rho(?w :inputParameter ?x) = inputParameter$ (Case 4), assuming that $|inputParameter| \leq |WorkflowSubject|$. α is assigned accordingly.

The second optimization is the elimination of some redundant triple patterns from a basic graph pattern bgp in a SPARQL query. In particular, we eliminate tp of the form $X \text{ rdf:type } :c$, if X also appears in another triple pattern tp' , and $\rho(tp') = cSubject$ or $\rho(tp') = cObject$. This is based on the fact that $X \text{ rdf:type } :c$ restricts X to match only instances of type $:c$, however the same restriction is already in place when we match X over relational attribute $\alpha(tp'.sub)$ of $cSubject$ or $\alpha(tp'.obj)$ of $cObject$.

In our sample query, $?w \text{ rdf:type } :Workflow$ should be eliminated, because $\rho(?w \text{ rdf:type } :Workflow) = Workflow$ and $\rho(?w ?p ?o) = WorkflowSubject$.

Finally, we are ready to present our schema-independent basic graph pattern translation algorithm BGPtoSQL in Figure 11. BGPtoSQL takes bgp , ρ , and α and outputs an equivalent SQL query to evaluate bgp against the relational RDF database. Its main idea is to retrieve all possible variable instantiations from relations that correspond (based on ρ) to triple patterns in bgp , restricting (1) relational attributes that correspond (based on α) to the same variables in different triple patterns to match the same values and (2) relational attributes that correspond (based on α) to instances or literals to match the values of those instances or literals, respectively.

```

01 Algorithm BGPtoSQL
02 Input:  $bgp$ ,  $\rho$ - and  $\alpha$ - mappings
03 Output: SQL
04 Begin
05   Assign a unique alias  $a_i$  to each  $tp_i \in bgp$ 
06   Construct the SQL From clause: for each  $tp_i \in bgp$ ,  $from \ += \ "$\rho(tp_i) \$a_i"$ ,
07   Construct an inverted index (hash)  $h$  on variables in  $bgp$ :
08     for each  $tp_i \in bgp$ , for each variable  $?v \in tp_i$ ,  $h(?v) \cup = \ "$a_i.\$\alpha(tp_i, pos(?v))"$ ,
09     where  $pos(?v) \in \{sub, pre, obj\}$  encodes the position of  $?v$  in  $tp_i$ 
10   Construct the SQL Where clause:
11     for each distinct variable  $?v \in bgp$  and  $|h(?v)| > 1$ ,
12     let  $x \in h(?v)$ , for each  $y \in h(?v)$  and  $y \neq x$ ,  $where \ += \ "$x = $y \text{ And}"$ 
13     for each  $tp_i \in bgp$ , for each literal or instance  $l \in tp_i$  and  $\alpha(tp_i, pos(l)) \neq undef$ ,
14      $where \ += \ "$a_i.\$\alpha(tp_i, pos(l)) = '$l' \text{ And}"$ 
15   Construct the SQL Select clause:
16     for each distinct variable  $?v \in bgp$ , let  $?v \in tp_i$ ,  $select \ += \ "$a_i.\$\alpha(tp_i, pos(?v)) \text{ As } v,"$ 
17   Return  $"\text{Select } select \text{ From } from \text{ Where } where"$ 
18 End Algorithm

```

Fig. 11. Algorithm BGPtoSQL

For our optimized query with the basic graph pattern consisting of $?w ?p ?o$ and $?w :inputParameter ?x$, the SQL **From** clause contains two relations (lines 05-06 in the algorithm) $WorkflowSubject$ and $inputParameter$ with aliases $t1$ and $t2$, respectively. The hash h (lines 07-09) contains only one relational attribute for every variable, except for $?w$, $h(?w) = \{t1.i, t2.s\}$, and therefore, the SQL **Where**

clause should ensure their equality $t1.i = t2.s$ (lines 10-12). The *bgp* contains one instance (*:inputParameter*), however its α value is undefined (lines 13-14). The SQL `Select` clause projects each distinct variable in *bgp* (lines 15-16), resulting in the translated query (line 17):

```
Select t1.i As w, t1.p As p, t1.o As o, t2.o As x
From WorkflowSubject t1, inputParameter t2 Where t1.i=t2.s
```

Finally, the SQL equivalent of our SPARQL query is

```
Select w, p, o From (
  Select t1.i As w, t1.p As p, t1.o As o, t2.o As x
  From WorkflowSubject t1, inputParameter t2 Where t1.i=t2.s ) t3
```

Note that our system supports the translation and evaluation of arbitrary complex optional graph patterns [19], group patterns, value constraints, which are out of this paper scope. We illustrate by example some of these features in the next section.

4.2 Provenance queries

A wide range of scientific queries can be answered based on our provenance ontology model enhanced with reasoning capabilities. In Table 3, we present several such queries that our e-scientist collaborators are interested in. For each query expressed in English, we construct a SPARQL query and its SQL counterpart using our translation algorithms. While queries Q1-Q9 use standard SPARQL features, in queries Q10-Q12, we use our extensions to SPARQL, denoted as SPARQL+, that are summarized in the following:

- *Negation*. The `NOT` clause in Q10 requires its graph pattern to fail (for the whole query to succeed) under the same variable instantiations as in the containing pattern. Although, negation can be expressed in pure SPARQL as shown in Q9, the explicit `NOT` greatly improves the readability of a query.
- *Aggregation*. SQL’s `COUNT`, `MAX`, `MIN`, `AVG`, and `GROUP BY` are useful constructs that SPARQL lacks. Fortunately, our approach can use the power of a relational database to implement them (see Q11).
- *Set operations*. Similarly to the `UNION` operation, a query may require other set operations like difference, intersection, containment, etc. In particular, Q12 implements the division operator.

4.3 Experimental study

Since the query API is frequently used for interactive browsing and visualization of provenance, fast query response is an essential requirement for provenance management. Our SPARQL-to-SQL translation showed to be very efficient, returning SQL equivalents of the sample provenance queries in less than 0.01s. The “cold” query response times of the SPARQL queries, Q1-Q12, for our two database schemas and two database size settings are reported in Figure 12. In addition, we compared the performance of our approach to the performance of

Jena 2.5.2 and Sesame 1.2.6 on queries Q1-Q9; since Sesame did not support SPARQL, we had to represent the queries in the SeRQL [13] query language. We do not report “warm” trials of the queries (second time, third time, and so forth), since they showed nearly 0.000s time for most queries due to caching.

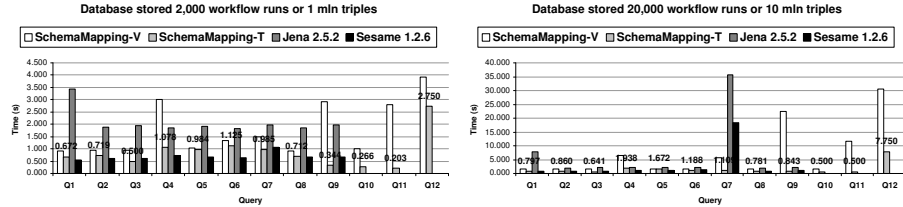


Fig. 12. Performance of provenance queries

For 2,000 workflow runs or one million triples, Sesame was the fastest for Q1, Q2, Q4, Q5, Q6, and Q8. SchemaMapping-T was the fastest for Q3, Q7, and Q9; these three queries were translated into SQL using our optimizations that reduced the number of relations and relational joins in the queries. Both Sesame and SchemaMapping-T were approximately 2-3 times faster than Jena. SchemaMapping-V was on-average faster than Jena and showed the third best performance. For the SPARQL+ queries Q10, Q11, and Q12, SchemaMapping-T was significantly faster than SchemaMapping-V.

For 20,000 workflow runs or 10 million triples, Sesame was the fastest for Q1, Q4, Q5, and Q8, while SchemaMapping-T was the fastest for Q2, Q3, Q6, Q7, and Q9. However, in this experiment, the difference between Sesame and SchemaMapping-T for Q1, Q2, Q6, and Q8 was quite small (<10%). Similar to the previous results, SchemaMapping-V and Jena showed slower performance than SchemaMapping-T and Sesame; SchemaMapping-V was on-average faster than Jena. For all the SPARQL+ queries, SchemaMapping-T was significantly faster than SchemaMapping-V.

From these two experiments, we observed that single-table approaches SchemaMapping-V, Jena, and Sesame scaled worse than SchemaMapping-T, i.e., the evaluation of Q7 suffered with the growth of the database size.

In summary, although we could not identify an absolute winner between two best performers Sesame and SchemaMapping-T, we believe that Sesame will suffer for larger datasets. Sesame’s database schema design requires to map URIs/literals in a SPARQL query to integer IDs and vice-versa, i.e., integer IDs returned in the query result to URIs/literals. If such mappings were encoded in an SQL query via relational joins of *Triples*, *Resources*, and *Literals*, the query response would have been unacceptably slow (e.g., Q1, when translated into SQL according to Sesame’s schema, took several minutes for 10 million triples). To avoid expensive joins, Sesame caches the mappings in main mem-

ory, which makes this approach memory bound. When there is a huge number of URIs/literals in a dataset, ID-to-URI/literal mapping may become problematic. In contrast, **SchemaMapping-T** only relies on the database engine for query processing rather than on in-memory data structures. While **SchemaMapping-T** provides comparable query performance with Sesame, **SchemaMapping-T** enables faster data mapping and better scalability for both data mapping and query processing.

5 Related Work and Discussion

In this section, we discuss related work on scientific workflow provenance management, as well as RDBMS-based RDF storage and querying systems, and highlight our contributions in these areas.

5.1 Scientific workflow provenance management

Provenance management has become an important functionality for most scientific workflow management systems; see [11, 54] for surveys. The Kepler system [7] implements a *provenance recorder* to record information about a workflow run, including the context, data derivation history, workflow definition, and workflow evolution. The provenance recorder is parametric and customizable, allowing the user to choose different levels of granularity of provenance data for recording. Based on the provenance information, Kepler supports efficient workflow rerun for a slightly modified workflow. In addition, the provenance recorder implements the Read-Write-State-Reset (RWS) provenance model proposed by Bowers et al. [12] for pipelined scientific workflows. The RWS model records read, write, and state-reset events for each actor in a workflow run and stores them in a relational event log. An approach is proposed to reconstruct various dependency graphs from the event log for a workflow run to support a wide range of scientific provenance queries.

The myGrid/Taverna system [60] uses Semantic Web technologies for representing provenance metadata at four levels: process, data, organization, and knowledge. Two levels of ontologies are used. A domain-independent schema ontology is used to describe the classes of resources and the properties between them that are needed to represent the four levels of provenance. A domain ontology is used to classify various types of resources such as data types, service types, and topic of interest for a particular domain.

The Chimera system [30] introduces a Virtual Data Catalog (VDC) consisting of a set of relations to store the description of executable programs as transformations, their actual invocations as derivations, and input/outputs as data objects. A workflow is constructed as a directed acyclic derivation graph via a Virtual Data Language (VDL). Chimera uses provenance for tracking the data derivation history, on-demand data generation and re-generation, and data product validation.

The VisTrails system [31] is the first system that supports provenance tracking of workflow evolution in addition to tracking the data product derivation history. In VisTrails, workflow evolution provenance is represented as a rooted tree, in which each node corresponds to a version of a workflow, and each edge corresponds to an update action that was applied to the parent workflow to create the child workflow. Therefore, a workflow evolution tree concisely represents all the workflow versions that a scientist has explored to produce the visualization products. In this way, VisTrails can support scientists to navigate through the space of workflows and parameter settings for an exploration process.

The above provenance systems are tightly coupled with their scientific workflow environments. A couple of stand-alone provenance systems have also been developed, including the PReServ system developed under the Provenance Aware Service Oriented Architecture (PASOA) project [32] and the Karma system [55]. The PReServe supports the recording of interaction provenance, actor provenance, and input provenance with the Provenance Recording Protocol (PREP), which specifies the messages that actors can asynchronously exchange with the provenance store to support provenance submission. The Karma system records provenance at four dimensions: execution, location, time, and dataflow, and use a Publish-Subscribe notification protocol for provenance collection; the Karma Provenance Browser is used for provenance presentation. Both PReServ and karma support web service interfaces.

Finally, data provenance is closely related to the data lineage problem [24, 23] studied in the database community, which determines the source data that is used to produce a data item. Buneman et al. [16] refine the concept of data lineage into why-provenance and where-provenance and propose an approach to derive them when a data product is the result of a database query from relational or XML databases. While why-provenance refers to the source data that had some influence on the existence of a data product, where-provenance refers to the locations in the source databases from which the data was extracted. Recently, Buneman et al. proposed a provenance management technique in curated databases [15]. However, in scientific workflows, datasets are not necessarily contained in a relational or XML database and data processing cannot be necessarily accomplished by a database query. Therefore, existing approaches to the data lineage problem are not sufficient for solving the data provenance problem in scientific workflows.

Although our approach is similar to Taverna's in that Semantic Web technologies are used for provenance modeling and representation, our contributions are different as we focus on the efficiency of provenance queries using an RDBMS. First, our strategy for ontology to database schema mapping results in a database schema that is optimized for common provenance queries. Second, our RDF to relational data mapping procedure employs our incremental strategy that takes advantage of the order of inserting various kinds of provenance metadata. Finally, we use SPARQL for provenance querying and propose a schema-independent SPARQL-to-SQL translation algorithm; the translation

algorithm is optimized by using the type information of an instance and the statistics of the size of the tables in the database.

5.2 Storing and querying RDF data using an RDBMS

In recent years, a number of RDBMS-based RDF stores (see [8] for a survey) have been developed to support large-scale Semantic Web applications. Database schemas employed by such systems fall into three categories [57]:

Schema-oblivious (also called *generic* or *vertical*): A single table, e.g., $Triple(s,p,o)$, is used to store RDF triples, such that attribute s stores the subject of a triple, p stores its predicate, and o stores its object. Schema-oblivious RDF stores include Jena [59, 43], Sesame [14], 3store [36, 37], KAON [58], RStar [42], and OpenLink Virtuoso [29]. This approach has no concerns of RDF schema or ontology evolution, since it employs generic database representation.

Schema-aware (also called *specific* or *binary*): This approach usually employs an RDF schema or ontology to generate so called *property tables* and *class tables*. A property table, e.g., $Property(s,o)$, is created for each property in an ontology and stores subjects s and objects o related by this property. A class table, e.g., $Class(i)$, is created for each class in an ontology and stores instances i of this class. Representatives of schema-aware RDF stores are DLDB [47], RDFSuite [6, 57], BDOWL [45], and PARKA [56]. Another research work that falls into this category, but rather stands out, is presented in [28]. In [28], database schema is generated based on patterns found in RDF data using data mining techniques. Schema evolution for this approach is quite straightforward (except for [28]): the addition or deletion of a class/property in an ontology requires the addition or deletion of a table in the database. The schema-aware approach is in general yields better query performance than the schema-oblivious approach as has been shown in several experimental studies [5, 57, 6].

Hybrid: This approach uses the mix of features of the previous two approaches. An example of the hybrid database schema is presented in [57], where schema-oblivious database representation, e.g., $Triple(s,p,o)$, is partitioned into multiple tables based on the data type of object o , and a binary relation, e.g., $Class(i,c)$, is introduced to store instances i of classes c . [57] reports comparable query performance of the hybrid and schema-aware approaches.

The inference support mechanisms employed by RDF stores can be classified as *forward-chaining* or *backward-chaining*. In forward-chaining, all inferences are precomputed and stored along with explicit triples of an RDF graph. This enables fast query response and increased result completeness [33]; however, it complicates RDF data updates and consumes more storage space. In backward-chaining, inferences are computed dynamically for each query, which simplifies updates and omits a storage overhead, but results in worse query performance and scalability. This mechanism is bound by the main memory space required to compute inferences.

The SPARQL-to-SQL query translation, SPARQL query processing and optimization literature includes [25, 37, 19, 50, 53, 39, 38, 9, 20, 40]. None of these works proposes a SPARQL-to-SQL translation algorithm that works for both schema-oblivious and schema-aware RDF stores or features query optimizations presented in this paper. Moreover, to our best knowledge, there is no published algorithm for translating basic graph patterns into SQL in a schema-aware RDF store in the literature. Such an algorithm is of high importance, since the translation of other SPARQL constructs, such as optional and alternative graph patterns, does not in general depend on storage schema information, but on the basic graph pattern translation. Cyganiak [25] presents a relational algebra for SPARQL assuming a schema-oblivious RDF store and outlines rules establishing equivalence between this algebra and SQL. Harris and Shadbolt [37] show how basic graph pattern expressions, as well as simple optional graph patterns, can be translated into relational algebra expressions over a schema-oblivious RDF store. Our work [19] presents algorithms for basic and optional graph pattern translation into SQL for a schema-oblivious RDF store; optional graph pattern semantics of SPARQL [4] has changed since then, which was triggered by the compositional semantics presented by Perez et al. [49, 48]. Polleres [50] contributes with the translation of SPARQL queries into Datalog, along with other contributions on the extensions of SPARQL and its semantics. Serfiotis et al. [53] study the containment and minimization problems of RDF query fragments using a logic framework that allows to reduce these problems into their relational equivalents. Hartig and Heese [39] propose a SPARQL query graph model and pursue query rewriting based on this model. Harth and Decker [38] propose optimized index structures for RDF that can support efficient evaluation of select-project-join queries and can be implemented in a relational database. Bernstein et al. [9] propose SPARQL query optimization techniques based on triple pattern selectivity estimation and evaluate them using an in-memory SPARQL query engine. Chong et al. [20] introduce an SQL table function into the Oracle database to query RDF data, such that the function can be combined with SQL statements for further processing. Hung et al. [40] study the problem of RDF aggregate queries by extending an RDF query language with the GROUP BY clause and several aggregate functions. Several research works [52, 51, 26, 10] focus on accessing conventional relational databases using SPARQL, which requires SPARQL-to-SQL query translation. Finally, Guo et al. [35, 34] define requirements for Semantic Web knowledge base systems benchmarks and propose a framework for developing such benchmarks.

In this paper, we develop an RDF store that supports both schema-oblivious and schema-aware database representations. In the latter case, along with property and class tables, we introduce two novel tables – *class-subject* and *class-object* tables. A class-subject table, e.g., $ClassSubject(i, p, o)$ stores triples whose subjects are instances of a particular class in an ontology. Similarly, a class-object table, e.g., $ClassObject(s, p, i)$, stores triples whose objects are instances of a particular class. Such tables are useful for queries that retrieve all information about an instance (object or subject) and for provenance queries in particular.

Furthermore, we explore several RDF-to-Relational data mapping strategies for the schema-aware representation, two of which work for RDF data in general and the other two are optimized for provenance metadata specifically. We propose a SPARQL-to-SQL translation algorithm for basic graph pattern queries. The algorithm is schema-independent, i.e., it works for schema-aware, as well as schema-oblivious, RDF stores. The algorithm is optimized, i.e., it selects smallest tables to query based on the type information of an instance and the statistics of the size of the tables in the database, as well as eliminates redundancies in basic graph patterns. These query optimization techniques are novel and can be applied in schema-aware RDF stores in general.

Finally, we explore several performance characteristics of our two alternative database schemas, such as schema mapping, data mapping, storage space consumption, and query response time, and compare them with the performance of Jena [59, 43] and Sesame [14]. Jena and Sesame were selected, in part, because of their popularity and, in part, because they have already been used for provenance metadata management in the Taverna system [60]. Unlike in other experimental studies that benchmark specific querying capabilities of RDF stores, such as extensional taxonomic queries [33], intensional taxonomic queries [21], intensional and extensional taxonomic queries [57], our study targets real-life provenance queries and does not limit the query complexity. As a result, we present the first study of SPARQL expressivity and usability for scientific workflow provenance querying. Moreover, we extend SPARQL with negation, aggregation, and set operations to support additional important provenance queries.

6 Conclusions and Future Work

In this work, we designed an RDBMS-based provenance management system for storing and querying scientific workflow provenance metadata. Our approach seamlessly integrates the interoperability, extensibility, and reasoning advantages of Semantic Web technologies with the storage and querying power of an RDBMS. Our schema mapping, data mapping, and SPARQL-to-SQL query translation algorithms are optimized to efficiently support (1) common provenance queries, (2) incremental data loading that employs the ordering of inserting various provenance metadata, and (3) schema-independent query translation that is optimized on-the-fly by using the type information of an instance and the statistics of the sizes of the tables in the database. Experimental results showed that our algorithms are efficient and scalable. The comparison with existing RDF stores, Jena and Sesame, showed that our optimizations provide improved efficiency and scalability to provenance metadata management. The approximate rankings of `SchemaMapping-V`, `SchemaMapping-T`, Jena 2.5.2, and Sesame 1.2.6 based on conducted experiments are presented in the following table, where rank 1 stands for the best performance/scalability, rank 2 stands for the second best performance/scalability, and so forth.

RDF store characteristic	SchemaMapping-V	SchemaMapping-T	Jena 2.5.2	Sesame 1.2.6
Data mapping of sequences of datasets	1	3	2	4
Data mapping of one dataset (“warm”)	1	3	2	4
Data mapping of one dataset (“cold”)	1	2	1	2
Data mapping scalability	1	1	1	2
Disk space consumption (total)	2	3	1	1
Query performance	2	1	3	1
Query processing scalability	2	1	2	2

The two alternative database representations, our schema-oblivious representation **SchemaMapping-V** and our schema-aware representation **SchemaMapping-T**, provide the flexibility to setup a provenance repository based on specific scientific workflow needs. **SchemaMapping-V** supports very fast schema and data mappings, while **SchemaMapping-T** supports very efficient query processing. Our schema-independent query translation allows transparent switching between these two representations and provides a foundation for further schema optimization. The only characteristic where our RDF store performs worse than Jena and Sesame is the disk space consumption due to extensive use of database indexes. For larger datasets, to improve this characteristic, we may chose to drop some redundant or infrequently used indexes. The last, but not least, we explored the need of SPARQL extension with negation, aggregation, and set operations to support additional important provenance queries.

In the future, we will continue to investigate further optimizations of database schema design, data loading, and querying, with the main focus on semantic query optimization.

Acknowledgements

We thank all VIEW contributors and team members, in particular, Cui Lin for implementing the provenance recording interface in the VIEW workbench, Zhaoqiang Lai and Jing Hua for their work on interactive provenance visualization, Seunghan Chang for his work on provenance access control, and many e-scientists who provide direct or indirect support to this effort, among them Otto Muzik, Jeffrey Ram, Stephen Krawetz, and Jeffrey Loeb.

References

1. OWL Web Ontology Language Reference. W3C Recommendation, 10 February 2004. M. Dean and G. Schreiber (Eds.). <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
2. RDF Test Cases. W3C Recommendation, 10 February 2004. J. Grant and D. Beckett (Eds.). <http://www.w3.org/TR/rdf-testcases/>.
3. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. G. Klyne, J. J. Carroll, and B. McBride (Eds.). <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
4. SPARQL Query Language for RDF. W3C Candidate Recommendation, 14 June 2007. E. Prud'hommeaux and A. Seaborne (Eds.). <http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/>.

5. R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *Proc. of the VLDB Conference*, pages 149–158, 2001.
6. S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On storing voluminous RDF descriptions: The case of Web portal catalogs. In *Proc. of the International Workshop on the Web and Databases (WebDB)*, 2001.
7. I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance collection support in the Kepler scientific workflow system. In *Proc. of the International Provenance and Annotation Workshop (IPAW)*, pages 118–132, 2006.
8. D. Beckett and J. Grant. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/.
9. A. Bernstein, C. Kiefer, and M. Stocker. OptARQ: A SPARQL optimization approach based on triple pattern selectivity estimation. Technical Report ifi-2007.03, March 2007. <http://www.ifi.uzh.ch/ddis/staff/goehring/btw/files/ifi-2007.03.pdf>.
10. C. Bizer and A. Seaborne. D2RQ - treating non-RDF databases as virtual RDF graphs. In *Proc. of the International Semantic Web Conference (ISWC)*, 2004. Poster presentation.
11. R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.
12. S. Bowers, T. M. McPhillips, B. Ludäscher, S. Cohen, and S. B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *Proc. of the International Provenance and Annotation Workshop (IPAW)*, 2006.
13. J. Broekstra and A. Kampman. SeRQL: A second generation RDF query language. 2003. <http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/aduna.pdf>.
14. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 54–68, 2002.
15. P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *Proc. of the SIGMOD Conference*, pages 539–550, 2006.
16. P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. of the International Conference on Database Theory (ICDT)*, pages 316–330, 2001.
17. A. Chebotko, X. Fei, C. Lin, S. Lu, and F. Fotouhi. Storing and querying scientific workflow provenance metadata using an RDBMS. In *Proc. of the International Workshop on Scientific Workflows and Business Workflow Standards in e-Science, in conjunction with IEEE International Conference on e-Science and Grid Computing*, 2007.
18. A. Chebotko, C. Lin, X. Fei, Z. Lai, S. Lu, J. Hua, and F. Fotouhi. VIEW: a Visual sciEntific Workflow management system. In *Proc. of the International Workshop on Scientific Workflows (SWF), in conjunction with IEEE International Conference on Services Computing (SCC)*, 2007.
19. A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. Tech. Rep. TR-DB-052006-CLJF. May 2006. <http://www.cs.wayne.edu/~artem/main/research/TR-DB-052006-CLJF.pdf>.
20. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. of the VLDB Conference*, pages 1216–1227, 2005.

21. V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtonis. On labeling schemes for the Semantic Web. In *Proc. of the International World Wide Web Conference (WWW)*, pages 544–555, 2003.
22. D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.
23. Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1):41–58, 2003.
24. Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.
25. R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170. 2005. <http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html>.
26. C. Perez de Laborda and S. Conrad. Bringing relational data into the Semantic Web using SPARQL and Relational.OWL. In *Proc. of the ICDE Workshops*, page 55, 2006.
27. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.
28. L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application specific schema design for storing large RDF datasets. In *Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.
29. O. Erling. Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. OpenLink Software Virtuoso. 2001. <http://virtuoso.openlinksw.com/wiki/main/Main/VOSRDFWP>.
30. I. Foster, J. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2002.
31. J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *Proc. of the International Provenance and Annotation Workshop (IPAW)*, 2006.
32. P. Groth, S. Miles, W. Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *Proc. of the International Symposium on High Performance Distributed Computing (HPDC)*, 2005.
33. Y. Guo, J. Heflin, and Z. Pan. Benchmarking DAML+OIL repositories. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 613–627, 2003.
34. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
35. Y. Guo, A. Qasem, Z. Pan, and J. Heflin. A requirements driven framework for benchmarking Semantic Web knowledge base systems. *IEEE Trans. Knowl. Data Eng.*, 19(2):297–309, 2007.
36. S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS)*, pages 1–15, 2003.
37. S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2005.

38. A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *Proc. of the Latin American Web Congress (LA-WEB)*, pages 71–80, 2005.
39. O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *Proc. of the European Semantic Web Conference (ESWC)*, pages 564–578, 2007.
40. E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *Proc. of the ICDE Conference*, pages 717–728, 2005.
41. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.
42. L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, pages 484–491, 2004.
43. B. McBride. Jena: Implementing the RDF model and syntax specification. Hewlett Packard Laboratories. 2001. <http://www.hpl.hp.com/personal/bwm/papers/20001221-paper/>.
44. S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.
45. S. Narayanan, T. M. Kurc, and J. H. Saltz. DBOWL: Towards extensional queries on a billion statements using relational databases. Technical Report. 2006. <http://bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf>.
46. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
47. Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *Proc. of the International Workshop on Practical and Scaleable Semantic Web Systems (PSSS)*, pages 109–113, 2003.
48. J. Perez, M. Arenas, and C. Gutierrez. *Semantics of SPARQL*. http://ing.atalca.cl/~jperez/papers/sparql_semantics.pdf.
49. J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proc. of the International Semantic Web Conference (ISWC)*, 2006.
50. A. Polleres. From SPARQL to rules (and back). In *Proc. of the International World Wide Web Conference (WWW)*, pages 787–796, 2007.
51. E. Prud’hommeaux. Notes on adding SPARQL to MySQL. <http://www.w3.org/2005/05/22-SPARQL-MySQL/>.
52. E. Prud’hommeaux. Optimal RDF access to relational databases. <http://www.w3.org/2004/04/30-RDF-RDB-access/>.
53. G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of RDF/S query patterns. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 607–623, 2005.
54. Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.
55. Y. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *Proc. of the International Conference on Web Services (ICWS)*, pages 427–436, 2006.
56. K. Stoffel, M. G. Taylor, and J. A. Hendler. Efficient management for very large ontologies. In *Proc. of the American Association for Artificial Intelligence Conference (AAAI)*, 1997.

57. Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In *Proc. of the International Semantic Web Conference (ISWC)*, 2005.
58. R. Volz, D. Oberle, B. Motik, and S. Staab. KAON SERVER - a Semantic Web management system. In *Proc. of the International World Wide Web Conference (WWW), Alternate Tracks - Practice and Experience*, 2003.
59. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. of the International Workshop on Semantic Web and Databases (SWDB)*, 2003.
60. J. Zhao, C. Wroe, C. A. Goble, R. Stevens, D. Quan, and R. M. Greenwood. Using Semantic Web technologies for representing e-science provenance. In *Proc. of the International Semantic Web Conference (ISWC)*, 2004.
61. Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. of the International Workshop on Scientific Workflows (SWF), in conjunction with IEEE International Conference on Services Computing (SCC)*, 2007.
62. Z. Zhao, A. Belloum, C. de Laat, P. Adriaans, and B. Hertzberger. Distributed execution of aggregated multi domain workflows using an agent framework. In *Proc. of the International Workshop on Scientific Workflows (SWF), in conjunction with IEEE International Conference on Services Computing (SCC)*, 2007.

Table 3. Provenance queries

Q1: Return all the tasks of a workflow :w1 that require user input (have input parameters).
SPARQL: Select Distinct ?t Where {?t :partOf :w1 . ?t :inputParameter ?p .}
SQL: Select t From (Select t1.s As t, t2.o As p From partOf t1,inputParameter t2 Where t1.s=t2.s And t1.o='w1') t3
Q2: Return the data dependency graph for a workflow run :wr1.
SPARQL: Select ?d1 ?d2 Where {?d1 :partOf :wr1 . ?d1 :directDataDependency ?d2 .}
SQL: Select d1, d2 From (Select t1.s As d1, t2.o As d2 From partOf t1, directDataDependency t2 Where t1.s=t2.s And t1.o='wr1') t3
Q3: Return all information about a workflow run :wr1.
SPARQL: Select ?p ?o Where {?wr1 rdf:type :WorkflowRun . :wr1 ?p ?o .}
SQL: Select p, o From (Select t1.p As p, t1.o As o From WorkflowRunSubject t1 Where t1.i='wr1') t2
Q4: Return all the data objects that have been used to produce a data object :wr1.d5. In addition, return task runs (if any) that have produced each such data object.
SPARQL: Select ?d ?tr Where {?wr1.d5 :transitiveDataDependency ?d . Optional {?tr :output ?d} .}
SQL: Select d, tr From (Select t1.o As d From transitiveDataDependency t1 Where t1.s='wr1.d5') t3 Natural Left Outer Join (Select t2.s As tr, t2.o As d From output t2) t4
Q5: Return all the workflow runs of a workflow :w1 whose input parameter "p" has a value in the range between 5 and 10.
SPARQL: Select ?wr Where {?wr :instanceOf :w1 . ?wr :inputParameter ?p . ?p :title "p" . ?p :dataValue ?v . Filter (?v >= 5 && ?v <= 10) .}
SQL: Select wr From (Select t1.s As wr, t2.o As p, t4.o As v From instanceOf t1, inputParameter t2, title t3, dataValue t4 Where t1.s=t2.s And t2.o=t3.s And t1.o='w1' And t3.o='p' And (t4.o >= 5 And t4.o <=10)) t5
Q6: Return all the workflow runs of workflows that directly evolved from a workflow :w1 and these runs have used an input dataset located at a URL url.
SPARQL: Select ?wr Where {?wr :instanceOf ?w . ?w :directWorkflowEvolution :w1 . ?wr :input ?d . ?d :locationURI "url"}
SQL: Select wr From (Select t1.s As wr, t1.o As w, t3.o As d From instanceOf t1, directWorkflowEvolution t2, input t3, locationURI t4 Where t1.s=t3.s And t1.o=t2.s And t3.o=t4.s And t2.o='w1' And t4.o='url') t5
Q7: Return all the workflow runs of a workflow :w1 that have used the results (final or intermediate) that have been generated by a run of a workflow :w2.
SPARQL: Select Distinct ?wr1 Where {?wr1 :instanceOf :w1 . :w1 rdf:type :Workflow . ?wr1 :input ?d . ?wr2 :instanceOf :w2 . :w2 rdf:type :Workflow . ?tr :partOf ?wr2 . ?tr :output ?d .}
SQL: Select Distinct wr1 From (Select t1.s As wr1, t2.o As d, t3.s As wr2, t4.s As tr From WorkflowObject t1, input t2, WorkflowObject t3, partOf t4, output t5 Where t1.i=t2.s And t2.o=t5.o And t3.s=t4.o And t4.s=t5.s And t1.p='instanceOf' And t1.i='w1' And t3.p='instanceOf' And t3.i='w2') t6
Q8: Return all the datasets that have been used or produced by a task run :wr1.t1.
SPARQL: Select ?d Where {{{:wr1.t1 :input ?d .} Union {:wr1.t1 :output ?d .}}
SQL: Select d From (Select t1.o As d From input t1 Where t1.s='wr1.t1') t3 Union Select d From (Select t2.o As d From output t2 Where t2.s='wr1.t1') t4
Q9: Return all the tasks of a workflow :w1 that require no user interaction.
SPARQL: Select ?t Where {?t rdf:type :Task . ?t :partOf :w1 . Optional {?t :inputParameter ?p} . Filter (!bound(?p)) .}
SQL: Select t From (Select t1.i As t From TaskSubject t1 Where t1.p='partOf' And t1.o='w1') t3 Natural Left Outer Join (Select t2.s As t, t2.o As p From inputParameter t2) t4 Where p Is Null
Q10: Return all the tasks of a workflow :w1 that require no user interaction.
SPARQL+: Select ?t Where {?t rdf:type :Task . ?t :partOf :w1 . NOT {?t :inputParameter ?p} .}
SQL: Select t From (Select t1.i As t From TaskSubject t1 Where t1.p='partOf' And t1.o='w1' And Not Exists (Select t2.s As t, t2.o As p From inputParameter t2 Where t2.s = t1.i)) t3
Q11: Return the number of the workflow runs in the system.
SPARQL+: Select COUNT(?wr) Where {?wr rdf:type :WorkflowRun .}
SQL: Select COUNT(wr) From (Select t1.i As wr From WorkflowRun t1) t2
Q12: Return all the datasets that have been used by all the workflow runs of a workflow :w1.
SPARQL+: Select ?d ?wr1 Where {{{?wr1 :instanceOf :w1 . ?wr1 :input ?d .} DIVIDE {?wr1 :instanceOf :w1 .}}}
SQL: Select Distinct d From (Select t1.s As wr, t2.o As d From instanceOf t1, input t2 Where t1.s=t2.s And t1.o='w1') r1 Where Not Exists (Select * From (Select t3.s As wr From instanceOf t3 Where t3.o='w1') r2 Where Not Exists (Select * From (Select t1.s As wr, t2.o As d From instanceOf t1, input t2 Where t1.s=t2.s And t1.o='w1') r3 Where r1.d=r3.d And r3.wr=r2.wr))