Storing, Tracking, and Querying Provenance in Linked Data

Marcin Wylot*†, Philippe Cudré-Mauroux‡, Manfred Hauswirth*†, and Paul Groth¶
*Open Distributed Systems, TU Berlin / Fraunhofer FOKUS, Berlin—Germany
†m.wylot@tu-berlin.de ‡manfred.hauswirth@tu-berlin.de
§eXascale Infolab, University of Fribourg—Switzerland, pcm@unifr.ch
¶Elsevier Labs, Amsterdam—The Netherlands, p.groth@elsevier.com

Abstract—The proliferation of heterogeneous Linked Data on the Web poses new challenges to database systems. In particular, the capacity to store, track, and query provenance data is becoming a pivotal feature of modern triplestores. We present methods extending a native RDF store to efficiently handle the storage, tracking, and querying of provenance in RDF data. We describe a reliable and understandable specification of the way results were derived from the data and how particular pieces of data were combined to answer a query. Subsequently, we present techniques to tailor queries with provenance data. We empirically evaluate the presented methods and show that the overhead of storing and tracking provenance is acceptable. Finally, we show that tailoring a query with provenance information can also significantly improve the performance of query execution.

Index Terms—RDF, Linked Data, triplestores, BigData, provenance

1 Introduction

A central use-case for Resource Description Framework (RDF) data management systems is data integration [1]. Data is acquired from multiple sources either as RDF or converted to RDF; schemas are mapped; record linkage or entity resolution is performed; and, finally, integrated data is exposed. There are a variety of systems such as Karma [2] and the Linked Data Integration Framework [3] that implement this integration process. The heterogeneity of data combined with the ability to easily integrate it—using standards such as RDF and SPARQL—mean that the support of provenance within these systems is a key feature [3]. For example a user may want to trace which sources were instrumental in providing results, how data sources were combined, to validate or invalidate results, or to tailor queries specifically based on provenance information.

Within the Web community, there have been several efforts in developing models and syntaxes to specify and trace provenance, which resulted in the recent W3C PROV recommendation [4]. However, less attention has been given to the efficient handling of such provenance data in RDF database systems. The most common mechanism used within RDF data management is named graph [5]. This mechanism was recently standardized in RDF 1.1. [6]. Named graphs associate a set of triples with an URI. Using this URI, metadata including provenance can be associated with the graph. While named graphs are often used for provenance, they are also used for other purposes, for example, to track access control information. Thus, while RDF databases, i.e., triplestores, support named graphs, there has only been a relatively small number of approaches specifically focused on provenance within the triplestore itself and much of it has been focused on theoretical aspects of the problem rather than efficient implementations.

Given the prevalence of provenance in Web Data—36% of datasets contain provenance data [7]—and the use of named graphs [8], [9], this article shows how RDF databases can effectively track the lineage of queries and execute queries that originate from data scoped with provenance information (i.e., provenance-enabled queries).

In the following, we present TripleProv, a new database system supporting the transparent and automatic derivation of detailed provenance information for arbitrary queries and the execution of queries with respect to provenance data. TripleProv is based on a native RDF store [10], [11], which we have extended with two different provenance-aware storage models and co-location strategies to store provenance data in a compact fashion. In addition, TripleProv supports query execution strategies to derive provenance information at two different levels of granularity and to scope queries with provenance information.

The contribution of the work presented in this article is the integration of our previous approaches into provenance-enabled triplestore [12], [13]. The new version of the system allows the user to **execute provenance-enabled queries and at the same time obtain a provenance polynomial of the query results**. Moreover, we present **two new experimental scenarios thoroughly evaluating the scalability of our techniques**. The first scenario (Section 8.3) varies the dataset size to assess the scalability of the storage models. The second scenario (Section 8.4) measures the performance impact of the selectivity of a provenance query. A proof-of-concept of this integrated solution was presented in a demo version of the system [14]. We note that the current version of the system does not support reasoning, which can pose further difficulties in deriving provenance but which is beyond the scope of this work. In the following we describe

the key aspects of TripleProv:

1) provenance polynomials to track the lineage of RDF queries at two different granularity levels (Section 4);
2) a characterization of provenance-enabled queries to tailor the query execution process with provenance information (Section 5);
3) new provenance-aware storage models and indexing strategies to compactly store provenance data (Section 6);
4) novel provenance oriented query execution strategies to compute provenance polynomials and execute provenance-enabled queries efficiently (Section 7);
5) an experimental evaluation of our system using two different datasets and workloads (Section 8).

All the datasets and queries we used in our experiments are publicly available for further investigations [1].

2 Related Work

Data provenance has been widely studied within the database, distributed systems, and Web communities. For a comprehensive review of the provenance literature, we refer readers to the work of Luc Moreau [15]. Likewise, Cheney et al. provide a detailed review of provenance within the database community [16]. Broadly, one can categorize the work into three areas [17]: content, management, and use. Work in the content area has focused on representations and models of provenance. In management, the work has focused on collecting provenance in software ranging from scientific databases [18] to operating systems or large scale workflow systems as well as mechanisms for querying it. Finally, provenance is used for a variety of applications including debugging systems, calculating trust and checking compliance. Here, we briefly review the work on provenance with respect to the Web of Data. We also review recent results applying theoretical database results to SPARQL.

Within the Web of Data area, one focus of work has been on designing models (i.e., ontologies) for provenance information [19]. The W3C Incubator Group on provenance mapped nine different models of provenance [20] to the Open Provenance Model [21]. Given the overlap in the concepts defined by these models, a W3C standardization activity was created that has led to the development of the W3C PROV recommendations for specyfying and interchanging provenance [4]. This recommendation is being increasingly adopted by both applications and data set providers - there have been over 60 implementations of PROV [22].

In practice, provenance is attached to RDF data using either reification [23], named graphs [24], or a singleton property [25]. Widely used datasets such as YAGO [26] reify their entire structure to facilitate provenance annotation. Indeed, provenance is one reason for the inclusion of named graphs in the current version of RDF [27]. Both named graphs and reification cause complex query structures especially as provenance becomes increasingly fine-grained. Indeed, formally, it may be difficult to track provenance using named graphs under updates and RDFS reasoning [28].

To address these issues, a number of authors have adopted the notion of annotated RDF [29], [30]. This approach assigns annotations to each of the triples within a dataset and then tracks these annotations as they propagate through reasoning or query processing pipelines. Formally, these annotated relations can be represented by the algebraic structure of communicative semirings, which can take the form of polynomials with integer coefficients [31]. These polynomials represent how source tuples are combined through different relational algebra operators (e.g., UNION, JOINS). These relational approaches are now being applied to SPARQL [32].

As Damasio et al. have noted, many of the annotated RDF approaches do not expose how-provenance (i.e., how a query result was constructed) [33]. The most comprehensive implementations of these approaches were presented by Zimmermann et al. [34] and Udrea et al. [29]. However, they have only been applied to small datasets (around 10 million triples) and do not report provenance polynomials for SPARQL query results. Annotated approaches have also been used for propagating "trust values" [35]. Other recent work [33], [36] has looked at expanding the theoretical aspects of applying such a semiring-based approach to capture SPARQL. In contrast, our work focuses on the implementation aspects of using annotations to track provenance within the query processing pipeline.

The concept of a *provenance query* was defined by Simon Miles in order to only select a relevant subset of all possible results when looking up the provenance of an entity [37]. A number of authors have presented systems for specifically handling such provenance queries. Biton et al. showed how user views can be used to reduce the amount of information returned by provenance queries in a workflow system [38]. The MTCProv [39] and the RDFProv [40] systems focus on managing and enabling queries over provenance that result from scientific workflows. Similarly, the ProQL approach [41] defines a query language and proposes relational indexing techniques for speeding up provenance queries involving path traversals. Glavic and Alonso presented the Perm provenance system, which was able to compute, store and query relational provenance data [42]. Provenance is computed by using standard relational query rewriting techniques, e.g., using lazy and eager provenance computation models. Recently, Glavic and his team have built on this work to show the effectiveness of query rewriting for tracking provenance in databases that support audit logs and "time travel" [43]. Our approach is different, in that it looks at the execution of provenance queries in conjunction with standard queries within a graph database. Widom et al. presented the Trio system [44] that supports the joint management of data, uncertainty and lineage. Lineage is an integral part of the storage model in this system, i.e., it is associated with its corresponding record in the database. Trio persists lineage information in a separate lineage relation where each record corresponds to a database tuple and contains provenance-specific attributes. However, Trio is not a database for semi-structured data, which is specifically our target. TripleProv leverages the specific requirements of RDF data and queries to enable efficient tracking and querying of provenance information.

In that respect, our work is related to the work on annotated RDF [29], [34], which developed SPARQL query extensions for querying over annotation metadata (e.g.,

1. https://exascale.info/projects/tripleprov/

provenance). Halpin and Cheney have shown how to use SPARQL Update to track provenance within a triplestore with no modifications to the SPARQL specyfication [45]. Our focus is different, however, since we propose and empirically evaluate different execution strategies for running queries that take advantage of provenance metadata.

Our system partially builds upon dynamic query execution approaches, which have been studied in different contexts by database researchers. Graefe and Ward [46] focused on determining when re-optimizing a given query that is issued repeatedly is necessary. Subsequently, Colde and Graefe [47] proposed a new query optimization model, which constructs dynamic plans at compile-time and delays some of the query optimization until run-time. Kabra and DeWitt [48] proposed an approach collecting statistics during the execution of complex queries in order to dynamically correct suboptimal query execution plans. Ng et al. studied how to re-optimize suboptimal query plans on-the-fly for very long-running queries in database systems [49]. Avnur and Hellerstein proposed Eddies [50], a query processing mechanism that continuously reorders operators in a query plan as it runs, and that merges the optimization and execution phases of query processing in order to allow each tuple to have a flexible ordering of the query operators. More recently, Madden et al. have extended Eddies to continuously adapt query plans over streams of data [51]. Our work is different in the sense that we dynamically examine or drop data structures during query execution depending on provenance information.

3 Provenance Concepts & Applications

W3C PROV defines provenance as "information about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [4]. The W3C PROV Family of Documents[2] defines a model, corresponding serializations and other supporting definitions to enable the interoperable interchange of provenance information in heterogeneous environments such as the Web. In this article, we adopt the view proposed in these specifications and consider provenance attached to RDF data with named graphs (graph label) [24]. Following the RDF 1.1 N-Quads specification[3] a graph label is denoted with an URI, which can be used as a resource[4] to describe provenance of the triple to which this graph label is attached. Therefore, provenance information is stored in the form of triples, where the subject URI is the same as the graph label attached to the original triple. In this article we adopt the terminology from the earlier N-Quads specification [52], hence when we describe provenance information we refer to a graph label as a *context value*. Below we give an example illustrating how provenance is attached to a triple:

Subject Predicate Object ContextValue1 .
ContextValue1 ProvPred1 ProvObj1 GraphX.
ContextValue1 ProvPred2 ProvObj GraphX.

There are many scenarios where provenance can be applied. Karvounarakis et al. [41] describe a number of use cases where storing and querying of provenance data generated by a database system is useful. To motivate our work we revisit some of these examples. Provenance polynomials express the exact way through which results of a query were derived. As such, they can be used to calculate scores or probabilities for particular query results (e.g., for post-processing tasks such as results ranking or faceted search). Likewise, one can use polynomials to compute a trust or information quality score based on the sources used in the result. Identifying results (i.e., particular triples) with overlapping provenance is another use case. One could detect whether a particular result would still be valid when removing a source dataset. One can also use provenance to modify query execution strategies on-the-fly, that is, to improve the query execution process with provenance information. For instance, one could restrict the results to certain subsets of the sources or use provenance for access control such that only certain sources will appear in a query result. Finally, one may need the capacity to provide a specification of the data he wants to use to derive an answer.

4 Provenance Polynomials

We now describe a model encoding how query results were derived, that is, a description of the provenance of query results. The first question we tackle is how to represent provenance information we want to return to the user in addition to the results themselves. Beyond listing the various tuples involved in the query, we want to be able to characterize the specific ways in which each source tuple contributed to the query results. Lineage of a query (a provenance trace) can be modeled at different granularity levels, see below Section 4.1 for a more formal definition of *lineage*.

In the article, we leverage the notion of a *provenance polynomial* which is an algebraic structure of communicative semirings representing how different pieces of data were combined through different relational algebra operators [31]. In contrast to recent work in this space, which addressed theoretical issues, we focus on the practical realization of this model in a high performance triplestore to answer queries. Specifically, we focus on two key requirements:

1) the capability to pinpoint, for each query result, the exact lineage of the result;
2) the capability to trace back, for each query result, the complete list of tuples and how they were combined to deliver a result.

To do this we support two different provenance operators at the physical level: *pProjection*, addressing the first requirement and pinpointing the exact lineage of the result, and *pConstraint*, tracing back the full provenance of the results. As shown in Figure 1, TripleProv returns a provenance polynomial describing the exact way the results were produced in addition to the query results.

At the logical level, we use two operators introduced by Green [53] to combine the provenance polynomials: ($\oplus$) represents unions of source tuples, and ($\otimes$) represents joins between source tuples.

Unions are used in two cases when generating the polynomials: first, when a constraint or a projection can be satisfied with multiple triples; meaning that there is more

2. http://www.w3.org/TR/prov-overview/
3. https://www.w3.org/TR/n-quads/
4. https://www.w3.org/TR/rdf11-concepts/#resources-and-statements

than one triple corresponding to the particular triple pattern of the query. The following polynomial:

$$l1 \oplus l2 \oplus l3$$

for instance, encodes the fact that a given result can originate from three different lineages ($l1$, $l2$, or $l3$). Second, unions are also used when multiple entities satisfy a set of constraints or projections (like the collection *provenance* in Section 7.2, Algorithm 2).

The join operator can also be used in two ways: to express the fact that entities were joined to handle a constraint or a projection, or to handle object-subject or object-object joins between sets of constraints. The following polynomial:

$$(l1 \oplus l2) \otimes (l3 \oplus l4)$$

for example, encodes the fact that lineages $l1$ or $l2$ were joined with lineages $l3$ or $l4$ to produce the results.

4.1 Provenance Granularity Levels

TripleProv returns RDF data provenance at different granularity levels. Current approaches (see Section 2), typically, return a list of named graphs from which the answer was computed. Our system, besides generating polynomials summarizing the complete provenance of the results, also supports two levels of granularity. First, a lineage l_i (i.e., an element appearing in a polynomial) can represent the *context value* of a triple, e.g., the fourth element in a quadruple. We call this granularity level *context-level*. Alternatively, a lineage can represent a quadruple, i.e., a triple plus its corresponding *context value*. This second type of lineage produces polynomials consisting of all the pieces of data, i.e., quadruples that were used to answer the query, including all intermediate results. We call this level of granularity *triple-level*. For instance given a provenance polynomial $l1 \oplus l2 \oplus l3$ TripleProv returns: at the *context level* $ContextValue1 \oplus ContextValue12 \oplus ContextValue13$ and at the *triple-level* $(Subject1 Predicate1 Object1) \oplus (Subject2 Predicate2 Object2) \oplus (Subject3 Predicate3 Object3)$.

5 Provenance-Enabled Queries

The next question is how to incorporate provenance in the query execution process. Our goal here is to provide to a user a way to specify which pieces of data should be used to answer a query. For this we introduce the following definitions:

Definition 1. A **Workload Query** is a query producing results for a user. These results are referred to as *workload query results*.

Definition 2. A **Provenance Scope Query** is a query that defines a scope on the query results (i.e., only results within that defined scope should be returned). Specifically, a *Provenance Scope Query* returns a set of *context values* whose corresponding triples will be considered during the execution of a *Workload Query*.

Definition 3. A **Provenance-Enabled Query** is a pair consisting of a *Workload Query* and a *Provenance Scope Query*, producing results for a user (as specified by the *Workload Query*) and originating only from data pre-selected by the *Provenance Scope Query*.

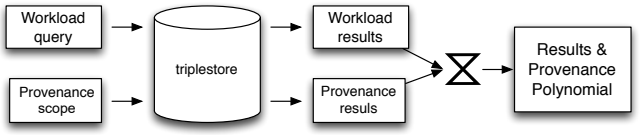


Figure 1: Executing provenance-enabled queries: both a workload and a provenance scope query are given as inputs to a triplestore, which produces results for both queries and then combines them to obtain the final results and a provenance polynomial.

As mentioned above, provenance data can be taken into account during query execution through the use of named graphs. However, those solutions are not optimized for provenance, and require rewriting all workload queries with respect to a provenance query. Our approach aims to keep workload queries unchanged and introduce provenance-driven optimization on the database system level.

We assume a strict separation of the workload query and the provenance scope query from the user perspective (as illustrated in Figure 1). That is, the user provides two separate queries: one to query the data and a second one to specify the provenance. Internally, the system combines these queries to provide the correct results. Including the provenance specification directly in to the query itself is also possible. The execution strategies and models we develop in the rest of this paper would work similarly in that case. Provenance and workload results are joined to produce a final result. A consequence of our design is that workload queries can remain unchanged, while the whole process of applying provenance filtering takes place during query execution. Both provenance scope and workload queries are delivered in the same way, preferably using the SPARQL language or a high-level API that offers a similar functionality. The body of the provenance scope query specifies the set of *context values* that identify which triples will be used when executing the workload queries.

To further illustrate our approach, we present a few provenance-enabled queries that are simplified versions of use cases found in the literature. In the examples below, *context values* are denoted as *?ctx*.

Provenance-enabled queries can be used in various ways. A common case is to ensure that the data used to produce the answer comes from a set of trusted sources [54]. Given a workload query that retrieves titles of articles about "Obama":

```
SELECT ?t WHERE {
  ?a <type> <article> .
  ?a <tag> <Obama> .
  ?a <title> ?t .
}
```

One may want to ensure that the articles retrieved come from sources attributed to the government:

```
SELECT ?ctx WHERE {
  ?ctx prov:wasAttributedTo <government> .
}
```

As per the W3C definition, provenance is not only about the source of data but is also about the manner in which

the data was produced. Thus, one may want to ensure that the articles in question were edited by somebody who is a "SeniorEditor" and that articles where checked by a "Manager". Thus, we could apply the following provenance scope query while keeping the same "Obama" workload query:

```
SELECT ?ctx WHERE {
  ?ctx prov:wasGeneratedBy <articleProd>.
  <articleProd> prov:wasAssociatedWith ?ed .
  ?ed rdf:type <SeniorEdior> .
  <articleProd> prov:wasAssociatedWith ?m .
  ?m rdf:type <Manager> .
}
```

A similar example to the one above, albeit for a curated protein database, is described by Chichester et al. [55].

Another way to apply provenance-enabled queries is for scenarios in which data is integrated from multiple sources. For example, we may want to aggregate the chemical properties of a drug (e.g., its potency) provided by one database with information whether it has regulatory approval provided by another:

```
SELECT ?potency ?approval WHERE {
  ?drug <name> ''Sorafenib'' .
  ?drug ?link ?chem.
  ?chem <potency> ?potency .
  ?drug <approvalStatus> ?approval
}
```

Here, we may like to select not only the particular sources from which the workload query should be answered but also the software or approach used in establishing the links between those sources. For instance, we may want to use links generated manually or those generated through the use a of "string similarity". Such a use-case is described in detail by Batchelor et al. [56]. Below is an example of how such a provenance scope query could be written:

```
SELECT ?ctx WHERE {
{ ?ctx prov:wasGeneratedBy ?linkingActivity.
  ?linkingActivity rdf:type <StringSimilarity> }
UNION {
 ?ctx prov:wasDerivedFrom <ChemDB>}
UNION {
 ?ctx prov:wasDerivedFrom <DrugDB>}
}
```

6 Storage Models

To efficiently execute provenance-enabled queries we developped a special RDF storage model for TripleProv. We extend the molecule-based native storage model we introduced in dipLODocus$_{[RDF]}$ [10] and DiploCloud [11] with new physical storage structures to store provenance.

6.1 Native Storage Model

The basic data unit in our system is an RDF molecule [57] which is similar to a property table [58] and stores the list or properties and objects related to a subject. A molecule contains all data directly related to a subject. Molecule clusters are used in two ways: to logically group sets of relates URIs and literals (thus, pre-computing joins), and to physically co-locate information related to a given resource. A molecule co-locates together all data related to one subject. For a detailed description on the native storage model, we refer the interested reader to our previous work [10], [11] which focused on this aspect specifically.

6.2 Storage Model Variants for Provenance

To extend the physical data structures of TripleProv to store provenance data we analyzed a number of ways to implement this extras. A first option of storing provenance data would be to simply annotate every object in the database with its corresponding context value. This produces quadruple physical data structures (*SPOC*, where *S* is the subject of the quadruple, *P* its predicate, *O* its object, and *C* its context value), as illustrated in Figure 2). The main advantage of this variant is its ease of implementation: one simply has to extend the data structure storing the object to also store the context value. Its main disadvantage, however, is memory consumption since the context value has to be repeated for each triple.

One can try to physically co-locate the context values and the triples differently, which results in a different memory consumption profile. One extreme option would be to regroup molecules in clusters based on their context values (*CSPO* clustering). This, however, has the negative effect of splitting our original molecules into several structures (one new structure per new context value using a given subject), thus breaking pre-computed joins and defeating the whole purpose of storing RDF as molecules in the first place. The situation would be even worse for deeper molecules, i.e., molecules storing data at a wider scope, or considering large RDF subgraphs. On the other hand, such structures would be quite appropriate to resolve vertical provenance queries, i.e., queries that explicitly specify the lineage of triples to consider during query execution.

The last two options for co-locating context values and triples are *SCPO* and *SPCO*. *SCPO* co-locates the context values with the predicates which avoids the duplication of the same context value inside a molecule, while at the same time still co-locates all data about a given subject in one structure. The last option, *SPCO*, co-locates the context value with the predicates in the molecules. This physical structure is very similar to *SPOC* in terms of storage requirements, since it rarely happens that a given context value uses the same predicates with many values. Compared to *SPOC*, it has the disadvantage of considering a relatively complex structure (*PC*) in the middle of the physical storage structure, i.e., as the key of a hash-table mapping to the objects.

These different ways of co-locating data naturally result in different memory overheads. The exact overhead, however, is highly dependent on the dataset considered, its structure, and the homogeneity / heterogeneity of the context values involved for the different subjects. Whenever the data related to a given subject refers to many different context values, the overhead caused by repeating the predicate in the *SCPO* might not be compensated by the advantage of co-location. In such cases, models like *SPCO* or *SPOC* might be more appropriate. If, on the other hand, a large portion of the different objects attached to the predicates relate to the same context value, then the *SCPO* model might pay off. It is evident that no single provenance storage model is overall best—since the performance of such models is dependent on the queries but also on the homogeneity / heterogeneity of the considered datasets.

In case we encounter conflicting data from different sources, we store it as it arrives in the system. Considering

a quadruple *SPOC*, the *C* element (context value) uniquely identifies the provenance of the triple. During query execution, we consider all triples as valid and we execute the query as if the triples were independent pieces of data. The provenance polynomial provides information on all pieces of data, conflicting or not.

For the reasons described above, we focus below on two very different storage variants in our implementation: *SCPO*, which we refer to as *data grouped by context value* in the following (since the data is regrouped by context values inside each molecule), and *SPOC*, which we refer to as *annotated provenance* since the context value is placed like an annotation next to the last part of the triple (object).

6.3 Provenance Index

Our system support a number of vertical and horizontal data co-location structures. We propose one more way to co-locate molecules, based on the context values. This gives us the possibility to prune molecules during query execution. Figure 3 illustrates this index, which is implemented as lists of co-located molecule identifiers indexed by the context values to which the triples stored in the molecules belong. A given molecule can appear multiple times in this index. This index is created upfront, i.e., at loading time. This index allows us to early prone irrelevant molecules in our pre-filtering query execution strategy (see Section 7.4).

6.4 Provenance-Driven Full Materialization

We implemented a materialized view of pre-selected data following the provenance specification (see Figure 4). This mechanism allows us to run our full materialization query execution strategy (see Section 7.4) only on a portion of data that is compliant with the provenance query since the data was pre-selected and materialized.

6.5 Partial Materialization

Finally, we also implement a new, dedicated structure for the partial materialization strategy. In that case, we co-locate all molecule identifiers that are matching the provenance specification, i.e., that contain at least one context value compatible with the provenance query. We explored several options for this structure and in the end implemented it through a hash-set, which gives us constant time performance to both insert molecules when executing the provenance query and to query for molecules when executing workload queries.

7 Query Processing

We now turn to the question of how to take advantage of the provenance data stored in the molecules to produce provenance polynomials and to tailor the query execution process with provenance information. We have implemented specific query execution strategies in TripleProv that allow us to compute a provenance polynomial describing how different pieces of data were combined to produce the results and to limit data used to produce the results to only those matching a provenance scope query.

7.1 Query Execution Pipeline

Figure 5 depicts the query execution process. The provenance scope and workload queries are provided as an input. The query execution process can vary depending on the exact strategy chosen, but typically it starts by executing the provenance scope query and optionally pre-materializing or co-locating data. The workload queries are then rewritten—by taking into account the results of the provenance scope query—and finally we execute them and at the same time we collect information on the data used in the query execution to compute a provenance polynomial. The process returns the workload query results as output, restricted to those that are following the specification expressed in the provenance scope query, and a provenance polynomial describing how particular pieces of data were combined to derive the results. We provide more detail on this execution process below.

7.2 Generic Query Execution Algorithm

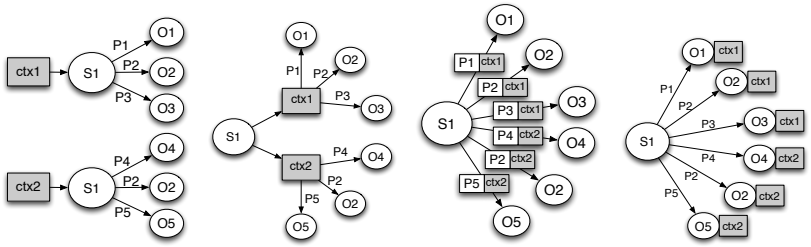
Algorithm 1 provides a simplified, generic version of the query execution algorithm. The algorithm takes as an input a provenance-enabled query, i.e., provenance scope and workload queries. We start by executing the provenance scope query, which is processed like an ordinary query (*ExecuteQuery*) but always returns sets of context values as output and itself is not restricted with provenance information (*ctxSet* = []). Subsequently, the system optionally materializes or adaptively co-locates the selected molecules containing data related to the provenance scope query. Afterwards we execute workload queries taking into account the context values returned from the previous step and we keep trace of all the triples contributing to the query evaluation. The execution starts as a standard query execution, but includes a dynamic query rewriting step to dynamically prune early in the query plan those molecules that cannot produce valid results given their provenance. We will describe different strategies to execute queries in Section 7.4.

Algorithm 1 Generic executing algorithm for provenance-enabled queries

```
Require: p: Provenance Scope Query
Require: q: workload query
1: ctxSet = ExecuteQuery(p, ctxSet=[])
2: materializedMolecules = MaterializeMolecules (ctxSet) OPTIONAL
3: collocatedMolecules = CollocateMolecules(ctxSet) OPTIONAL
4: for all workload queries do
5:     (results,polynomial) = ExecuteQuery(q, ctxSet)
6: end for
```

Algorithm 2 gives a simplified view of how simple star-like queries are answered in TripleProv (the algorithm performs only subject-subject joins). Given a SPARQL query, our system first analyzes the query to produce a physical query plan, i.e., a tree of operators that are then called iteratively to retrieve molecules likely to contain data relevant to the query (see our previous work for details [10], [11]).

During query execution, TripleProv collects all the triples that are instrumental in producing the results, i.e., molecules/sub-graphs matching the graph pattern of the query. For each molecule inspected (Algorithm 3), our system keeps tracks of all those parts of the RDF graph that



(a) CSPO Storage (b) SCPO Storage (c) SPCO Storage (d) SPOC Storage

Figure 2: The four different physical storage models identified for co-locating context values (ctx) with the triples (Subject Predicate Object) inside RDF molecules.

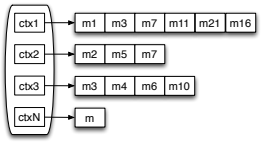


Figure 3: Provenance-driven indexing schema. Lists of molecules containing data related to a particular context value (ctx).

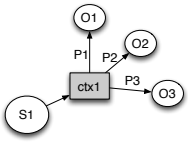


Figure 4: The molecule after materialization, driven by a provenance query returning only one context value (ctx).

have matched the evaluated query graph pattern (see our previous work [10], [11] for details of RDF query processing). TripleProv performs two kinds of graph matching operations: 1) It verifies if the triple in a molecule matches any of the triple patterns from the query and if the triple adheres to the provenance specification (*checkIfTripleExists and C ∈ ctxSet*); 2) From the matching triples it retrieves resources to compose the result set, i.e., projections (*getEntity*). The provenance track corresponding to one molecule is kept as "local provenance" (*provenanceLocal*). In case multiple molecules are used to construct the final results, the system performs a union of all the local provenance traces (*provenance.union*) and keeps the resulting trace as "global provenance" (*provenance*). To illustrate these operations and their results, below we describe the execution of two sample queries.

7.3 Example Queries

To show how the query evaluation works, we provide a few representative examples of workload queries: The first example query we consider is a simple star query, i.e., a query defining a series of triple patterns, all joined on the variable ?*a*:

```
SELECT ?t WHERE {
  ?a <type> <article> . (<- 1st constraint)
  ?a <tag> <Obama> .    (<- 2nd constraint)
  ?a <title> ?t . (<- 1st projection)
}
```

To build the corresponding provenance polynomial, TripleProv first identifies the projections and constraints from the query (see the annotated listing above). Projections

Algorithm 2 Simplified algorithm for executing a query and computing a provenance polynomial (*ExecuteQuery*).

Require: q: query
Require: ctxSet: context values; results of a provenance scope query
1: provenance ← NULL : provenance polynomial for the query (q)
2: results ← NULL : results of the query (q)
3: molecules ← q.getPhysicalPlan
4: constraints ← q.getConstraints
5: projections ← q.getProjections
6: **for all** molecules **do**
7: Filter molecules based on the provenance scope query (ctxSet) {only for the pre-filtering strategy}
8: (provenanceLocal, resultsLocal) = inspectMolecule (constraints, projections, molecule, ctxSet)
9: **if** (provenanceLocal is NOT NULL AND resultsLocal is NOT NULL) **then**
10: provenance.union(provenanceLocal)
11: results.union(resultsLocal)
12: **end if**
13: **end for**
14: **return** (results, provenance)

Algorithm 3 Algorithm for inspecting a molecule (*inspectMolecule*). The algorithm presents the way how we collect information on data provenance to compute the provenance polynomial.

Require: ctxSet: context values; results of a provenance scope query
Require: constraints, projections, molecule
1: provenanceLocal ← NULL
2: resultsLocal ← NULL
3: **for all** constrains **do**
4: **if** checkIfTripleExists(&provenanceLocal) **then**
5: provenanceLocal.join
6: **else**
7: **return** NULL
8: **end if**
9: **end for**

10: **for all** projections **do**
11: entity = getEntity(for particular projection, &provenanceLocal)
12: **if** entity is NOT EMPTY **then**
13: resultsLocal.add(entity)
14: provenanceLocal.join
15: **else**
16: **return** NULL
17: **end if**
18: **end for**

19: **return** (resultsLocal, provenanceLocal)

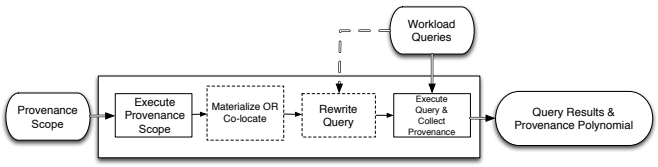


Figure 5: Provenance-enabled query execution pipeline.

correspond to these triple patterns that contain variables included in the result set (from the SELECT statement). Constraints correspond to all the remaining triple patterns defining the query graph pattern. Each molecule is inspected (Algorithm 3) to determine whenever i) the various constraints can be met i.e., the triple pattern matches any triple from the molecule (*checkIfTripleExists*), ii) the piece of data follows the provenance specification described through the provenance scope query . Following this, for the triples that match the triple pattern, the system retrieves resources that will compose the result set (*getEntity*). Both functions (*checkIfTripleExists* and *getEntity*) keep track of all the entities that were used to build the query answer besides performing their standard tasks. If they encounter a triple matching the triple pattern, they keep track of the triple in *provenanceLocal*. In case a triple pattern can by satisfied with multiple triples within one molecule both functions combine the lineage internally in *provenanceLocal* with a union operator $\oplus$. Our system keeps track of the provenance of each result, by joining ($\otimes$) the local provenance information of each triple used during the query execution to identify the result.

Finally, a provenance polynomial is issued:

$$[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5) \otimes (l6 \oplus l7)].$$

The association with the particular triple patterns from the query is expressed with the join operator ($\otimes$), and is as follows:

1st constraint	$?a < type >< article >$	$l1 \oplus l2 \oplus l3$
		$\otimes$
2nd constraint	$?a < tag >< Obama >$	$l4 \oplus l5$
		$\otimes$
1st projection	$?a < title >?t$	$l6 \oplus l7$

This particular polynomial indicates that the first constraint has been satisfied by a lineage of $l1$, $l2$, or $l3$, while the second has been satisfied by $l4$ or $l5$. It also indicates that the first projection was processed with elements having a lineage of $l6$ or $l7$. The union operator represents alternative triples that match the triple pattern. The triples involved were joined on variable $?a$, which is expressed by the join operation ($\otimes$) in the polynomial. Such a polynomial can contain lineage elements either at the source level or at the triple level, and can be returned both in an aggregated or detailed form as described in Section 4.1.

In case a query includes an OPTIONAL operator, e.g., $OPTIONAL(?a < title >?t)$, our system only includes entities that do exist in the database in the provenance polynomials. If there is no triple that matches the OPTIONAL pattern, our polynomials will not include any provenance information with respect to this triple pattern, e.g., $[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5)]$.

The second example we examine is more complex, as it contains two sets of constraints and projections with an upper-level join to bind them:

```
select ?l ?long ?lat
where {
  (-- first set)
  ?p name ''Krebs, Emil'' .
  ?p deathPlace ?l .

  (-- second set)
  ?c ?x ?l .
  ?c featureClass P .
  ?c inCountry DE .
  ?c long ?long .
  ?c lat ?lat .
}
```

The query execution starts similarly to the first sample query. After resolving the first two triple patterns (*ExecuteQuery*), the second set of patterns is processed in the following steps: 1) replacing variable $?l$ of the second set with the results derived from the first set; 2) executing the second set of triples with the function *ExecuteQuery*; 3) joining the corresponding results and lineage elements of both sets. The system takes the two tracks of provenance from the first and the second set of triple patterns and combines them with the join operator ($\otimes$).

Processing the query TripleProv automatically generates provenance polynomials such as the following:

$$[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5)] \bigotimes [(l6 \oplus l7) \otimes (l8) \otimes (l9 \oplus l10) \otimes (l11 \oplus l12) \otimes (l13)]$$

where an upper-level join ($\bigotimes$) is performed across the lineage elements resulting from both sets.

7.4 Execution Strategies for Provenance-Enabled Queries

We now examine the question of filtering pieces of data used to deliver a query result. There are several possible ways to execute provenance-enabled queries in a triplestore. The simplest way is to execute both a workload query and a provenance scope query independently, and to join both result sets based on a provenance polynomial and context values. One also has the option of pre-materializing some of the data based on the provenance specification. Another way to execute a provenance-enabled query is through dynamic query rewriting. In that case, the workload query is rewritten using the provenance scope query and then is executed. We now introduce five different strategies for executing provenance-enabled queries.

Post-Filtering: This is the baseline strategy that executes both the workload and the provenance scope queries independently. The provenance scope and workload queries can be executed in any order or concurrently. When both the provenance scope query and the workload query have been executed, the results from the provenance scope query (i.e., a set of context values) are used to filter the results of the workload query based on their provenance polynomial afterwards (see Algorithm 4). In this strategy we leverage the fact that the provenance polynomial pinpoints a set of context values to a result tuple. Moreover, the union operator indicates that we used several alternative triples to match a triple pattern. In consequence, if this set contains

the specific context value without an alternative we eliminate the pinpointed result tuple.

Rewriting: This strategy executes the provenance scope query first. Then, when inspecting triples in a molecule, this strategy verifies if the context value of the triple is in the set of context values returned by the provenance scope query besides matching a triple pattern (see Algorithm 5). This solution is efficient from the provenance scope query execution side, though it can be suboptimal from the workload query execution side (see Section 8).

Pre-Filtering: This strategy takes advantages of a dedicated *provenance index* co-locating, for each context value, the IDs of all molecules belonging to this context (see Section 6.3). After the provenance scope query is executed, the provenance index can be looked up to retrieve the lists of molecule IDs that are compatible with the provenance specification. These lists can then be used to filter out early the intermediate and final results of the workload queries (see Algorithm 6). After filtering, this strategy executes queries in a similar way as the *Rewriting* strategy.

Full Materialization: This is a two-step strategy where a provenance scope query is first executed on the entire database, and then all molecules containing context values that match the provenance scope query are materialized. The workload queries are then simply executed on the resulting materialized view, which only contains triples that are compatible with the provenance specification. This strategy will outperform all other strategies when executing the workload queries, since they are executed only on the relevant subset of the data. However, materializing all potential molecules based on the provenance scope query can be prohibitively expensive, both in terms of storage space and latency.

Partial Materialization: This strategy introduces a trade-off between the performance of a provenance scope query and that of workload queries. The provenance scope query is executed first. While executing the provenance scope query (see Algorithm 7), the system also builds a temporary structure (e.g., a hash-table) maintaining the IDs of all molecules belonging to the context values returned by the provenance scope query. When executing the workload query, the system can then dynamically and efficiently look-up all molecules appearing as intermediate or final results, and can filter them out early in case they do not appear in the temporary structure. Further processing is similar to the *Rewriting* strategy. This strategy can achieve performance close to the Full Materialization strategy while avoiding to replicate the data, at the expense of creating and maintaining the temporary data structure.

8 Experiments

To empirically evaluate our provenance-aware storage models and query execution strategies, we implemented them in TripleProv. Our approaches to store, track, and query provenance in Linked Data were already evaluated on a few datasets and workloads in our previous work [12], [13].

Algorithm 4 Algorithm for the *Post-Filtering* strategy.

```
Require: q: workload query
Require: p: provenance scope query
(ctxSet) = ExecuteQuery(p)
(results, polynomial) = ExecuteQuery(q) {independent execution of p and q}
for all results do
    {result ∈ results}
    if ( polynomial[result].ContextValues ∉ ctxSet ) then
        remove result
    else
        keep result
    end if
end for
```

Algorithm 5 Algorithm for the *Rewriting* strategy.

```
Require: q: workload query
Require: ctxSet: context values; results of a provenance scope query
molecules = q.getPhysicalPlan
results ← NULL
polynomial ← NULL
for all molecules do
    {molecule ∈ molecules}
    for all (entities in molecule) do
        {entity ∈ entities}
        if ( entity.ContextValues ∉ ctxSet) then
            next entity
        else
            if entity.match(q) then
                results.add(entity)
                calculate provenance polynomial
            end if
        end if
    end for
end for
return (results, polynomial)
```

Algorithm 6 Algorithm for the *Pre-Filtering* strategy.

```
Require: q: workload query
Require: ctxSet: context values; results of a provenance scope query
molecules = q.getPhysicalPlan
results ← NULL
polynomial ← NULL
for all molecules do
    {molecule ∈ molecules}
    for all ctxSet do
        {ctx ∈ ctxSet}
        ctxMolecules = getMoleculesFromProvIdx(ctx)
        if ( molecule ∉ ctxMolecules) then
            next molecule
        end if
    end for
    for all (entities in molecule) do
        {entity ∈ entities}
        if ( entity.ContextValues ∉ ctxSet) then
            next entity
        else
            if entity.match(q) then
                results.add(entity)
                calculate provenance polynomial
            end if
        end if
    end for
end for
return (results, polynomial)
```

Algorithm 7 Algorithm for the *Partial Materialization* strategy.

```
Require: q: workload query
Require: ctxSet: context values; results of a provenance scope query
Require: collocatedMolecules: collection of hash values of molecules
    related to results of the provenance query (ctxSet)
 1: molecules = q.getPhysicalPlan
 2: results ← NULL
 3: polynomial ← NULL
 4: for all molecules do
 5:     {molecule ∈ molecules}
 6:     if ( molecule ∉ collocatedMolecules) then
 7:         next molecule
 8:     end if
 9:     for all (entities in molecule) do
10:         {entity ∈ entities}
11:         if ( entity.ContextValues ∉ ctxSet) then
12:             next entity
13:         else
14:             if entity.match(q) then
15:                 results.add(entity)
16:                 calculate provenance polynomial
17:             end if
18:         end if
19:     end for
20: end for
21: return (results, polynomial)
```

The goal of the following experiments is to empirically evaluate the scalability of our algorithms and data structures. Therefore, in the fist scenario (Section 8.3) we perform the experiments on datasets of varying size to assess the performance of the storage models. In the second scenario (Section 8.4), we measure the performance impact of the selectivity of a provenance query. Hence we gradually increase the number of context values resulting from the provenance query. To facilitate the potential comparison of results, we use the same datasets and queries as in our previous work on provenance-enabled query processing [13].

8.1 Experimental Environment

Hardware Platform: All experiments were run on a HP ProLiant DL385 G7 server with an AMD Opteron Processor 6180 SE (24 cores, 2 chips, 12 cores/chip), 64GB of DDR3 RAM, running Ubuntu 12.04.3 LTS (Precise Pangolin). All data were stored on a recent 3 TB Serial ATA disk.

Datasets: We used the Billion Triples Challenge (BTC)[5] dataset for our experiments: It is a collection of RDF data gathered from the Web. The Billion Triple Challenge dataset was created based on datasets provided by Falcon-S, Sindice, Swoogle, SWSE, and Watson using the MultiCrawler/SWSE framework. It represents a typical collection of data gathered from multiple and heterogeneous online sources, hence storing, tracking, and querying provenance on it precisely addresses the problem we focus on. We considered more than 40 million quadruples (almost 10GB). To sample the data, we first pre-selected quadruples satisfying the set of considered workload and provenance queries. Then, we randomly sampled additional data up to 10GB.

Following the experimental design from our previous work [13], we included provenance specific triples so that the workload query results are always the same, irrespective if the query is tailored with provenance information or not (vanilla version).

Workloads: For the workload, we used eight existing queries originally proposed by Neumann and Weikum [59]. In addition, we added two queries with the UNION and OPTIONAL clauses. We prepared a complex provenance scope query, which is conceptually similar to those presented in Section 5.

Experimental Methodology: As is typical for benchmarking database systems, e.g., for tpc-x[6] or our own OLTP-Benchmark [60], we included a warm-up phase before measuring the execution time of the queries in order to measure query execution times in a steady-state mode. We first run all the queries in sequence once to warm-up the system, and then repeated the process ten times (i.e., we ran 11 query batches for each variant we benchmark, each containing all the queries we consider in sequence). In the following, we report the average execution time of the last 10 runs for each query. In addition, we avoided the artifacts of connecting from the client to the server, of initializing the database from files, and of printing results. Instead we measured the query execution times inside the database system only.

8.2 Results of the Previous Experiments

In our previous papers, we presented two independent systems focusing on different aspects of provenance within a triplestore. Here, we briefly summarize the results of those previous experiments:

The first paper [12] focused on storing and tracking lineage in the query execution process. Overall, the performance penalty created by tracking provenance were in a range from a few percents to almost 350%. We observed a significant difference between the two main provenance storage models implemented (CG vs CA and TG vs TA). Retrieving data from co-located structures took about 10%-20% more time than from simply annotated graph nodes. We also noticed considerable differences between the two granularity levels (CG vs TG and CA vs TA). The more detailed triple-level provenance granularity required more time than the simpler source-level.

The second paper [13] focused on different query execution strategies for provenance-enabled queries, that is, queries tailored with provenance information. We found that because provenance is prevalent within Web Data and is highly selective, it can be used to improve query processing performance. By leveraging knowledge on provenance, one can execute many types of queries roughly 30x faster than a baseline store. This is a counterintuitive result as provenance is often associated with additional processing overhead. The exact optimal strategy depends on the data, on the mixture of (provenance scope and workload) queries, and of their frequencies. Based on our experiments, we believe that a partial materialization strategy provides the best trade-off for running provenance-enabled queries in general.

8.3 Storage Models and Provenance Granularity Levels

For the first experimental scenario we wanted to investigate the scalability of our storage models. Specifically we wanted

5. http://km.aifb.kit.edu/projects/btc-2009/

6. *http://www.tpc.org/*

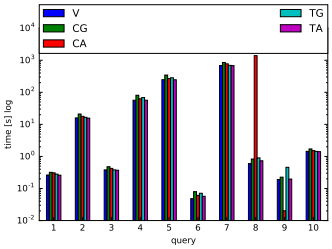


Figure 6: Query execution times (in seconds, logarithmic scale) for DS5.

Table 1: Size of the datasets used in the experiments.

	size (GB)	# quadruples
DS1	0.5	2 944 562
DS2	2.9	12 944 562
DS3	5.3	22 944 562
DS4	7.3	32 944 556
DS5	9.3	42 944 553

to measure how the performance changes with the increasing data size. We implemented two storage models (SCPO and SPOC, see Section 6.2) in TripleProv and for each model we considered two granularity levels for tracking provenance (context and triple, see Section 4.1). This gives us four different variants to compare against the vanilla version of our system. Our goal was to understand the various trade-offs of the approaches and to assess the performance penalty caused by computing a provenance polynomial. In this scenario we do not filter data based on provenance. Here we only derive a provenance polynomial of a workload query. We use the following abbreviations to refer to the different variants of storage models and granularity levels:

V: the vanilla version of our system (i.e., the version where provenance is neither stored nor looked up during query execution);

CG: context-level granularity, provenance data grouped by context values;

CA: context-level granularity, annotated provenance data;

TG: triple-level granularity, provenance data grouped by context values;

TA: triple-level granularity, annotated provenance data.

As mentioned above the main goal of these experiments was to compare the scalability of our methods. To achieve this goal in this scenario we gradually increased the size of the dataset. We started with nearly 3 million quadruples up to 43 million quadruples (see Table 1 for details about the datasets sizes).

8.3.1 Results

Figure 6 shows the query execution times for the biggest dataset we used in our experiments (DS5). Figure 7 shows the execution time for selected queries and all datasets. It exhibits how the performance of the implemented variants changes with the increasing data size.

Unsurprisingly, the more data we have to process the slower queries are executed (see Figure 7). The performance decrease varies between variants and queries. Queries with less selective constraints or introducing significant amount of intermediate results (like Q1, Q5, Q10) tend to be more sensitive to the size of data. Queries which can early prune the number of elements to inspect (e.g., Q9) exhibit lower sensitivity in that aspect. The variants grouping elements by the provenance data (CG and TG) are more sensitive to the size of data. While difference in size between the datasets is up to almost 20 times, for the query execution time we can observe a variety of the performance changes: form almost no difference (query Q9, variant CA) to 100-200 times (queries Q5, Q10, variant CG).

Finally, we briefly discuss specific results appearing in Figures 6 and 8 where the provenance-aware implementations seam to outperform the vanilla version. The reason behind the large disparity in performance has to do with the very short execution times (at the level of 10^{-3} seconds), which cannot be measured more precisely and thus introduces some noise.

Table 2: Cardinality of the context values set.

	%	# context values
full dataset	100	6 819 826
minimal	0.003	1 854
CV1	10	681 983
CV2	20	1 363 966
CV3	30	2 045 948
CV4	40	2 727 931
CV5	50	3 409 914

8.4 Query Execution Strategies

Our goal here was to investigate the scalability of the query execution strategies for provenance-enabled queries (see Section 7.4). Specifically we wanted to measure how they perform when the provenance scope query results in different selectivities, i.e., with different numbers of context values. To refer to the different query execution strategies we use the abbreviations introduced in Section 7.4.

To evaluate the scalability of the provenance-aware query execution strategies, we gradually increased the cardinality of the context values set. We started with the minimal number of elements to maintain the results of the workload queries unchanged. Following, we added elements to the set so that it consisted of 10% to 50% of context values from the entire dataset. Table 2 shows detailed statistics about the number of context values for each scenario.

8.4.1 Results

Figure 8 shows the query execution times for all queries for the biggest context values set (CV5). Figure 9 shows the execution time for selected queries and all measured sizes of the context values set (see Table 2). It shows how the query strategies are sensitive to the number of elements returned from the provenance query.

As expected, the more advanced query execution strategies (Pre-Filtering, Full Materialization, and Patrial Materialization) are significantly more sensitive to the number of results returned by the provenance scope query. Those strategies take advantage of the selectivity of the provenance information (see our previous work [13] for a detailed analysis of the provenance selectivity), so if the results of the provenance query are not selective enough, the strategies perform worse. Figure 9 shows that the Pre-Filtering strategy is the most sensitive to the output of the provenance scope query. The sensitivity varies between queries: for queries with a higher number of intermediate results (used for joins) and less selective constraints (e.g., Q2 and Q5), the difference can exceed two orders of magnitude.

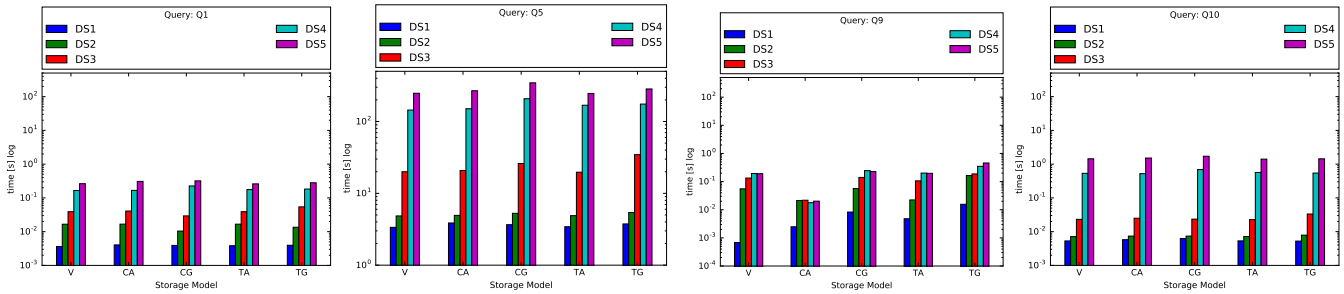


Figure 7: Query execution time (in seconds, logarithmic scale) for all datasets and the implemented storage variants, for selected queries (Q1, Q5, Q9, and Q10).

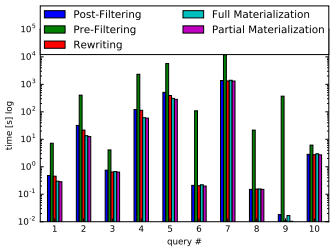


Figure 8: Query execution times (in seconds, logarithmic scale) for CV5.

The performance of queries that do not take advantage of high selectivity provenance information (e.g. Q6 and Q7) is not linked to the cardinality of the context values set.

9 Conclusions

To the best of our knowledge, we presented the first attempt to translate theoretical insight from the database provenance literature into a high-performance triplestore. Our techniques enable not only simple tracing of lineage for query results, but also considers fine-grained multilevel provenance and allow us to tailor the query execution with provenance information. We introduced two storage models and five query execution strategies for supporting provenance in Linked Data management systems.

From our experiment we can say that the more data we have to process the slower queries are executed and the more selective is the provenance query the more we gain in performance. Less selective workload queries are more sensitive to those aspects than queries that allow to early prune intermediate results. The more advanced query execution strategies that take advantage of the selectivity of the provenance information are more sensitive to the number elements returned by the provenance query.

It is important to remember that the performance of a system processing provenance, either by tracking or querying such meta data, highly depends on the selectivity of the provenance [12], [13]. On the one hand, the more selective provenance information is the more one can gain by scoping the query execution with provenance-enabled queries. On the other hand, with more selective data, tracking and storing provenance information becomes more expensive. Therefore such systems should be evaluated with respect to the data sampled from the targeted source to chose the most suitable storage model and a query execution strategy. A user of a system like TripleProv can easily restrict what data to be used in a query by providing a SPARQL query scoping the data provenance (Section 5). Additionally, the user will receive detailed information about exact pieces of data were used to produce the results (Section 4). Such a provenance trace can be used to compute the quality of the results or to (partially) invalidate the results in case some parts of the data collection turn out to be incorrect. Moreover, a provenance trace can be leveraged to partially reevaluate the query in case new data arrives, i.e., we can reevaluate only the part that is influenced by the new data since we know what data exactly has been used in the query.

Building on the results of this work, we see a number of possible avenues for future work: Investigating how provenance can be used to improve query execution and optimization within a data management system is one of them. Another possibility is to leverage provenance to improve the reconstruction of incomplete graph data, especially in the context of streams of Linked Data. Finally, we plan to investigate the ability to track and query provenance in dynamic Linked Data environments, linking provenance management to complex event processing and reasoning.

References

[1] T. Heath and C. Bizer, *Linked Data: Evolving the Web into a Global Data Space*, ser. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[2] C. A. Knoblock, P. Szekely, J. L. Ambite, S. Gupta, A. Goel, M. Muslea, K. Lerman, M. Taheriyan, and P. Mallick, "Semi-automatically mapping structured sources into the semantic web," in *Proceedings of the Extended Semantic Web Conference*, Crete, Greece, 2012.

[3] A. Schultz, A. Matteini, R. Isele, C. Bizer, and C. Becker, "Ldif - linked data integration framework," in *COLD*, 2011.

[4] P. Groth and L. Moreau (eds.), "PROV-Overview. An Overview of the PROV Family of Documents," World Wide Web Consortium, W3C Working Group Note NOTE-prov-overview-20130430, Apr. 2013.

[5] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *Proceedings of the 14th International Conference on World Wide Web*, ser. WWW '05. New York, NY, USA: ACM, 2005, pp. 613–622.

[6] D. W. R. Cyganiak and M. L. (Ed.), "RDF 1.1 Concepts and Abstract Syntax," W3C Recommendation, February 2014, http://www.w3.org/TR/rdf11-concepts/.

[7] M. Schmachtenberg, C. Bizer, and H. Paulheim, "Adoption of the linked data best practices in different topical domains," in *The Semantic Web, ISWC 2014*, ser. Lecture Notes in Computer Science, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandei, P. Groth, N. Noy, K. Janowicz, and C. Goble, Eds. Springer International Publishing, 2014, vol. 8796, pp. 245–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11964-9_16

[8] G. Grimnes. (2012) BTC2012 Stats. [Online]. Available: http://gromgull.net/blog/2012/07/some-basic-btc2012-stats/

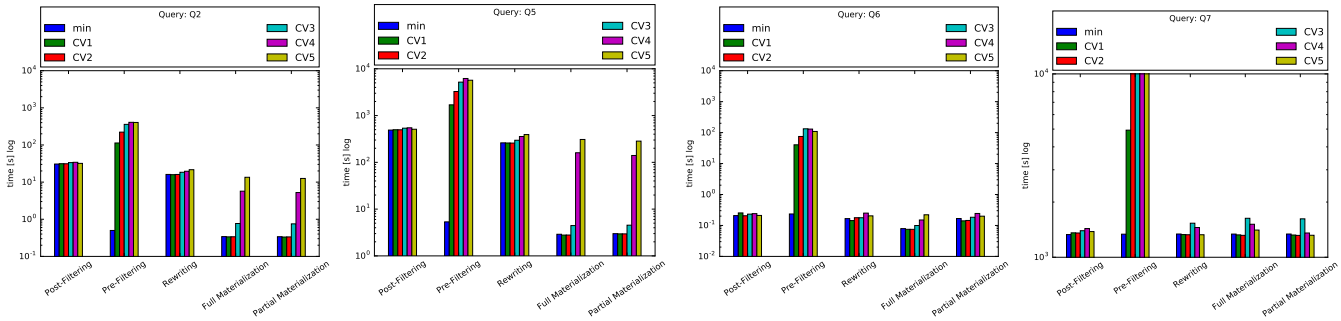


Figure 9: Query execution time (in seconds, logarithmic scale) for all sizes of the context values set and the query execution strategies, for selected queries (Q2, Q5, Q6, and Q7).

[9] P. Groth and W. Beek. (2016) Measuring PROV Provenance on the Web of Data. [Online]. Available: https://nbviewer.jupyter.org/github/pgroth/prov-wod-analysis/blob/master/MeasuringPROVProvenanceWebofData.ipynb

[10] M. Wylot, J. Pont, M. Wisniewski, and P. Cudré-Mauroux, "dipLODocus[RDF]: short and long-tail RDF analytics for massive webs of data," in *Proceedings of the 10th international conference on The semantic web - Volume Part I*, ser. ISWC'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 778–793. [Online]. Available: http://dl.acm.org/citation.cfm?id=2063016.2063066

[11] M. Wylot and P. Cudré-Mauroux, "DiploCloud: Efficient and Scalable Management of RDF Data in the Cloud," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 3, pp. 659–674, March 2016.

[12] M. Wylot, P. Cudre-Mauroux, and P. Groth, "TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2014, pp. 455–466.

[13] M. Wylot, P. Cudré-Mauroux, and P. Groth, "Executing Provenance-Enabled Queries over Web Data," in *Proceedings of the 24rd International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2015.

[14] ——, "A demonstration of TripleProv: tracking and querying provenance over web data," vol. 8, no. 12. VLDB Endowment, 2015, pp. 1992–1995.

[15] L. Moreau, "The foundations for provenance on the web," *Foundations and Trends in Web Science*, vol. 2, no. 2–3, pp. 99–241, Nov. 2010. [Online]. Available: http://eprints.ecs.soton.ac.uk/21691/

[16] J. Cheney, L. Chiticariu, and W.-C. Tan, *Provenance in databases: Why, how, and where*. Now Publishers Inc, 2009, vol. 1, no. 4.

[17] P. Groth, Y. Gil, J. Cheney, and S. Miles, "Requirements for provenance on the web," *International Journal of Digital Curation*, vol. 7, no. 1, 2012.

[18] P. Cudré-Mauroux, K. Lim, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. Zdonik, "A Demonstration of SciDB: A Science-Oriented DBMS," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2, pp. 1534–1537, 2009.

[19] O. Hartig, "Provenance information in the web of data," in *Proceedings of the 2nd Workshop on Linked Data on the Web (LDOW2009)*, 2009.

[20] S. Sahoo, P. Groth, O. Hartig, S. Miles, S. Coppens, J. Myers, Y. Gil, L. Moreau, J. Zhao, M. Panzer *et al.*, "Provenance vocabulary mappings," W3C, Tech. Rep., 2010.

[21] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van den Bussche, "The open provenance model core specification (v1.1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, Jun. 2011. [Online]. Available: http://eprints.ecs.soton.ac.uk/21449/

[22] T. D. Huynh, P. Groth, and S. Zednik (eds.), "PROV Implementation Report," World Wide Web Consortium, W3C Working Group Note NOTE-prov-implementations-20130430, Apr. 2013.

[23] P. Hayes and B. McBride, "Rdf semantics," W3C Recommendation, February 2004.

[24] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust," in *Proceedings of the 14th international conference on World Wide Web*. ACM, 2005, pp. 613–622.

[25] V. Nguyen, O. Bodenreider, and A. Sheth, "Don't like rdf reification?: making statements about statements using singleton property," in *Proceedings of the 23rd international conference on World wide web*. International World Wide Web Conferences Steering Committee, 2014, pp. 759–770.

[26] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum, "Yago2: A spatially and temporally enhanced knowledge base from wikipedia," *Artificial Intelligence*, vol. 194, no. 0, pp. 28 – 61, 2013, artificial Intelligence, Wikipedia and Semi-Structured Resources.

[27] J. Zhao, C. Bizer, Y. Gil, P. Missier, and S. Sahoo, "Provenance requirements for the next version of RDF," in *W3C Workshop RDF Next Steps*, 2010.

[28] P. Pediaditis, G. Flouris, I. Fundulaki, and V. Christophides, "On Explicit Provenance Management in RDF/S Graphs." in *Workshop on the Theory and Practice of Provenance*, 2009.

[29] O. Udrea, D. R. Recupero, and V. Subrahmanian, "Annotated RDF," *ACM Transactions on Computational Logic (TOCL)*, vol. 11, no. 2, p. 10, 2010.

[30] G. Flouris, I. Fundulaki, P. Pediaditis, Y. Theoharis, and V. Christophides, "Coloring RDF Triples to Capture Provenance," in *Proceedings of the 8th International Semantic Web Conference*, ser. ISWC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 196–212.

[31] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2007, pp. 31–40.

[32] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides, "On provenance of queries on semantic web data," *IEEE Internet Computing*, vol. 15, no. 1, pp. 31–39, Jan. 2011.

[33] C. V. Damásio, A. Analyti, and G. Antoniou, "Provenance for SPARQL queries," in *Proceedings of the 11th international conference on The Semantic Web - Volume Part I*, ser. ISWC'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 625–640.

[34] A. Zimmermann, N. Lopes, A. Polleres, and U. Straccia, "A general framework for representing, reasoning and querying with annotated semantic web data," *Web Semant.*, vol. 11, pp. 72–95, Mar. 2012.

[35] O. Hartig, "Querying Trust in RDF Data with tSPARQL," in *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ser. ESWC 2009 Heraklion. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 5–20.

[36] F. Geerts, G. Karvounarakis, V. Christophides, and I. Fundulaki, "Algebraic structures for capturing the provenance of SPARQL queries," in *Proceedings of the 16th International Conference on Database Theory*, ser. ICDT '13. New York, NY, USA: ACM, 2013, pp. 153–164.

[37] L. Moreau and I. Foster, "Electronically querying for the provenance of entities," in *Provenance and Annotation of Data*, ser. Lecture Notes in Computer Science, L. Moreau and I. Foster, Eds. Springer Berlin Heidelberg, 2006, vol. 4145, pp. 184–192.

[38] O. Biton, S. Cohen-Boulakia, and S. B. Davidson, "Zoom*userviews: Querying relevant provenance in workflow systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1366–1369.

[39] L. M. Gadelha, Jr., M. Wilde, M. Mattoso, and I. Foster, "MTCProv: A Practical Provenance Query Framework for Many-task Scientific

Computing," *Distrib. Parallel Databases*, vol. 30, no. 5-6, pp. 351–370, Oct. 2012.

[40] A. Chebotko, S. Lu, X. Fei, and F. Fotouhi, "RDFProv: A Relational RDF Store for Querying and Managing Scientific Workflow Provenance," *Data Knowl. Eng.*, vol. 69, no. 8, pp. 836–865, Aug. 2010.

[41] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 951–962.

[42] B. Glavic and G. Alonso, "The Perm Provenance Management System in Action," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1055–1058.

[43] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic, "A generic provenance middleware for queries, updates, and transactions," in *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*. Cologne: USENIX Association, Jun. 2014. [Online]. Available: https://www.usenix.org/conference/tapp2014/agenda/presentation/arab

[44] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," *Technical Report*, 2004.

[45] H. Halpin and J. Cheney, "Dynamic Provenance for SPARQL Updates," in *The Semantic Web ISWC 2014*, ser. Lecture Notes in Computer Science, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandei, P. Groth, N. Noy, K. Janowicz, and C. Goble, Eds. Springer International Publishing, 2014, vol. 8796, pp. 425–440. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11964-9_27

[46] G. Graefe and K. Ward, "Dynamic query evaluation plans," *SIGMOD Rec.*, vol. 18, no. 2, pp. 358–366, Jun. 1989.

[47] R. L. Cole and G. Graefe, "Optimization of dynamic query evaluation plans," *SIGMOD Rec.*, vol. 23, no. 2, pp. 150–160, May 1994.

[48] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," *SIGMOD Rec.*, vol. 27, no. 2, pp. 106–117, Jun. 1998.

[49] K. Ng, Z. Wang, R. R. Muntz, and S. Nittel, "Dynamic query re-optimization," in *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, Aug 1999, pp. 264–273.

[50] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *SIGMOD Rec.*, vol. 29, no. 2, pp. 261–272, May 2000.

[51] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: ACM, 2002, pp. 49–60.

[52] R. Cyganiak, A. Harth, and A. Hogan, "N-quads: Extending n-triples with context," *W3C Recommendation*, 2008.

[53] T. J. Green, "Collaborative data sharing with mappings and provenance," Ph.D. dissertation, University of Pennsylvania, 2009, rubinoff Dissertation Award; honorable mention for Jim Gray Dissertation Award.

[54] luc Moreau and G. Paul, *Provenance: An Introduction to PROV*. Morgan and Claypool, September 2013. [Online]. Available: http://eprints.soton.ac.uk/356858/

[55] C. Chichester, P. Gaudet, O. Karch, P. Groth, L. Lane, A. Bairoch, B. Mons, and A. Loizou, "Querying neXtProt nanopublications and their value for insights on sequence variants and tissue expression," *Web Semantics: Science, Services and Agents on the World Wide Web*, no. 0, pp. –, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826814000432

[56] C. R. Batchelor, C. Y. A. Brenninkmeijer, C. Chichester, M. Davies, D. Digles, I. Dunlop, C. T. A. Evelo, A. Gaulton, C. A. Goble, A. J. G. Gray, P. T. Groth, L. Harland, K. Karapetyan, A. Loizou, J. P. Overington, S. Pettifer, J. Steele, R. Stevens, V. Tkachenko, A. Waagmeester, A. J. Williams, and E. L. Willighagen, "Scientific lenses to support multiple views over linked chemistry data," in *ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, Oct. 2014, pp. 98–113. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11964-9_7

[57] L. Ding, Y. Peng, P. P. da Silva, and D. L. McGuinness, "Tracking RDF Graph Provenance using RDF Molecules," in *International Semantic Web Conference*, 2005.

[58] K. Wilkinson and K. Wilkinson, "Jena property table implementation," in *International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2006.

[59] T. Neumann and G. Weikum, "Scalable join processing on very large rdf graphs," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 627–640.

[60] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013.

Marcin Wylot is a postdoctoral researcher at TU Berlin in the Open Distributed Systems group. He received his PhD at the University of Fribourg in Switzerland in 2015, with the supervision of Professor Philippe Cudré-Mauroux. He graduated the MSc in computer science at the University of Lodz in Poland in 2010, doing part of his studies at the University of Lyon in France. During his studies he was also gaining professional experience working in various industrial companies. His main research interests revolve around database systems for Semantic Web data, provenance in Linked Data, and Big Data processing. Webpage: http://mwylot.net

Philippe Cudré-Mauroux is a Swiss-NSF Professor and the director of the eXascale Infolab at the University of Fribourg in Switzerland. Previously, he was a postdoctoral associate working in the Database Systems group at MIT. He received his Ph.D. from the Swiss Federal Institute of Technology EPFL, where he won both the Doctorate Award and the EPFL Press Mention in 2007. Before joining the University of Fribourg, he worked on distributed information and media management for HP, IBM Watson Research (NY), and Microsoft Research Asia. He was Program Chair of the International Semantic Web Conference in 2012 and General Chair of the International Symposium on Data-Driven Process Discovery and Analysis in 2012 and 2013. His research interests are in next-generation, Big Data management infrastructures for non-relational data. Webpage: http://exascale.info/phil

Manfred Hauswirth is the managing director of Fraunhofer FOKUS and a full professor for Open Distributed Systems at the Technical University of Berlin, Germany. Previous career stations were EPFL and DERI. He holds an M.Sc. (1994) and a Ph.D. (1999) in computer science from the Technical University of Vienna. His research interests are on Internet-of-Everything, domain-specific Big Data and analytics, linked data streams, semantic sensor networks, sensor networks middleware, peer-to-peer systems, service-oriented architectures and distributed systems security. He has published over 170 peer-reviewed papers in these domains, he has co-authored a book on distributed software architectures and several book chapters on data management, semantics, middleware and IoT, and is an associate editor of the IEEE Transactions on Services Computing.

Paul Groth is Disruptive Technology Director at Elsevier Labs. He holds a Ph.D. in Computer Science from the University of Southampton (2007) and has done research at the University of Southern California and the VU University Amsterdam. His research focuses on dealing with large amounts of diverse contextualized knowledge with a particular focus on the web and science applications. This includes research in data provenance, data science, data integration and knowledge sharing. Paul was co-chair of the W3C Provenance Working Group that created a standard for provenance interchange. He is co-author of Provenance: an Introduction to PROV; The Semantic Web Primer: 3rd Edition as well as numerous academic articles. He blogs at http://thinklinks.wordpress.com. You can find him on twitter: @pgroth.