TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store

Marcin Wylot
eXascale Infolab
University of Fribourg
Switzerland
marcin.wylot@unifr.ch

Philippe Cudré-Mauroux
eXascale Infolab
University of Fribourg
Switzerland
pcm@unifr.ch

Paul Groth
VU University Amsterdam
The Netherlands
p.t.groth@vu.nl

ABSTRACT

Given the heterogeneity of the data one can find on the Linked Data cloud, being able to trace back the provenance of query results is rapidly becoming a must-have feature of RDF systems. While provenance models have been extensively discussed in recent years, little attention has been given to the efficient implementation of provenance-enabled queries inside data stores. This paper introduces TripleProv: a new system extending a native RDF store to efficiently handle such queries. TripleProv implements two different storage models to physically co-locate lineage and instance data, and for each of them implements algorithms for tracing provenance at two granularity levels. In the following, we present the overall architecture of our system, its different lineage storage models, and the various query execution strategies we have implemented to efficiently answer provenance-enabled queries. In addition, we present the results of a comprehensive empirical evaluation of our system over two different datasets and workloads.

Categories and Subject Descriptors

H.2.2 [**Information Systems**]: Data Management—*Physical Design*

General Terms

Algorithms, Design, Experimentation

Keywords

Provenance Queries, Provenance Polynomials, RDF, Linked Open Data

1. INTRODUCTION

With the rapid expansion of the Linked Open Data (LOD) cloud, developers are able to query and integrate large collections of disparate online data. As the LOD cloud is rapidly growing, so is its heterogeneity. The heterogeneity of the data combined with the ability to easily integrate it—using standards such as RDF and SPARQL—mean that tracing back the provenance (or lineage) of query results becomes essential, e.g., to understand which sources were instrumental in providing results, how data sources were combined, to validate or invalidate results, and to delve deeper into data related to the results retrieved.

Within the Web community, there have been several efforts in developing models and syntaxes to interchange provenance, which resulted in the recent W3C PROV recommendation [11]. However, less attention has been given to the efficient handling of provenance data within RDF database systems. While some systems store quadruples or named graphs, to the best of our knowledge, no current high-performance triple store is able to automatically derive provenance data for the results it produces.

We aim to fill this gap. In the following, we present TripleProv, a new database system supporting the transparent and automatic derivation of detailed provenance information for arbitrary queries. TripleProv is based on a native RDF store, which we have extended with two different physical models to store provenance data on disk in a compact fashion. In addition, TripleProv supports several new query execution strategies to derive provenance information at two different levels of granularity. More specifically, we make the following contributions:

- We enable the provenance of query results to be expressed at two different granularity levels by leveraging the concept of provenance polynomials.
- We propose two new storage models to represent provenance data in a native RDF data store compactly, along with query execution strategies to derive the aforementioned provenance polynomials while executing the queries.
- Finally, we present a new system, TripleProv, implementing our approach and analyze its performance through a series of empirical experiments using two different Web-centric datasets and workloads.

The rest of this paper is structured as follows: We start below by reviewing related work and providing some use cases for provenance polynomials in Sections 2 and 3. We then give an overview or our system in Section 4. Our notion

.

of provenance polynomials is introduced in Section 5. This is followed by a discussion of two provenance storage models in Section 6, and our new query execution strategies to derive provenance polynomials in Section 7. Finally, we give the results of a detailed performance evaluation of in Section 8 before concluding.

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 [19]. Likewise, Cheney *et al.* provide a detailed review of provenance within the database community [2]. Broadly, one can categorize the work into three areas [10]: 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 [3] 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 with respect to SPARQL.

Within the Web of Data community, one focus of work has been on designing models (i.e., ontologies) for provenance information [13]. The W3C Incubator Group on provenance mapped nine different models of provenance [24] to the Open Provenance Model [20]. 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 interchanging provenance [11]. This recommendation is being increasingly adopted by both applications and data set providers - there are over 60 implementations of PROV [17].

In practice, provenance is attached to RDF data using either reification [15] or named graphs [1]. Widely used datasets such as YAGO [16] reify their entire structures to facilitate provenance annotations. Indeed, provenance is one reason for the inclusion of named graphs in the next version of RDF [29]. Both named graphs and reification lend to complex query structures especially as provenance becomes increasingly fined grained. Indeed, formally, it may be difficult to track provenance using named graphs under updates and RDFS reasoning [23].

To address these issues, a number of authors have adopted the notion of annotated RDF [26, 7]. This approach assigns annotations to each of the triples within a dataset and then tracks these annotations as they propagate through either the 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 [9]. 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 [25].[1]

[1]Note, in terms of formalization, SPARQL poses difficulties because of the OPTIONAL operator, which implies negation.

As [4] has noted, many of the annotated RDF approaches do not expose how-provenance (i.e., how a query result was constructed). The most comprehensive implementations of these approaches are [30, 26]. However, they have only been applied to small datasets (around 10 million triples) and are not aimed at reporting provenance polynomials for SPARQL query results. Annotated approaches have also been used for propagating trust values [14]. Other recent work, e.g., [8, 4], has looked at expanding the theoretical aspects of applying such a semiring based approach to capturing SPARQL. Our work instead focuses on the implementation aspects of using annotations to track provenance within the query processing pipeline. In particular, we scale to over a 100 million triples using real-world Web datasets and look at the implications of storage models on performance.

3. USE CASES FOR PROVENANCE POLYNOMIALS

There are many scenarios provenance polynomials can be applied to. [18] describes a number of use cases where storage and querying of provenance data generated by a database system could be useful. We revisit some of these here. Polynomials express the exact way through which the results were derived. As such, they can hence 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.

One can also use the provenance to modify query execution strategies on the fly. For instance, one could restrict the results to certain subsets of sources or use provenance for access control such that only certain sources will appear in a query result. Identifying results (i.e., particular triples) with overlapping provenance is also another prospective use case. Finally, one could detect whether a particular result would still be valid when removing a source dataset.

4. SYSTEM OVERVIEW

In the following, we give a high-level overview of TripleProv, a native RDF store supporting the efficient generation of provenance polynomials during query execution. TripleProv is based on dipLODocus[RDF] [28], a recent and native RDF database management system, and is available as an open-source package on our Web page[2].

Figure 1 gives an overview of the architecture of our system, composed of a series of subcomponents:

a query executor responsible for parsing the incoming query, rewriting the query plans, collecting and finally returning the results along with the provenance polynomials to the client;

a key index in charge of encoding URIs and literals into compact system identifiers and of translating them back;

a type index clustering all keys based on their types;

a series of RDF *molecules* storing RDF data as very compact subgraphs;

[2]`http://exascale.info/tripleprov`

a molecule index storing for each key the list of molecules where the key can be found.

We give below an overview of the three most important subcomponents of our system in a provenance context, i.e., the key index, the molecules, and the molecule index.

The key index is responsible for encoding all URIs and literals appearing in the triples into a unique system id (key), and back. We use a tailored lexicographic tree to parse URIs and literals and assign them a unique numeric ID. The lexicographic tree we use is essentially a prefix tree splitting the URIs or literals based on their common prefixes (since many URIs share the same prefixes), such that each substring prefix is stored once and only once in the tree. A key ID is stored at every leaf, which is composed of a type prefix (encoding the type of the element, e.g., *Student* or $xsd : date$) and of an auto-incremented instance identifier. This prefix tree allows us to completely avoid potential collisions (caused for instance when applying hash functions on very large datasets), and also lets us compactly co-locate both type and instance ids into one compact key. A second structure translates the keys back into their original form. It is composed of a set of inverted indices (one per type), each relating an instance ID to its corresponding URI / literal in the lexicographic tree in order to enable efficient key look-ups.

In their simplest form, RDF molecules [6] are similar to property tables [27] and store, for each subject, the list or properties and objects related to that 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 object on disk and in main-memory to reduce disk and CPU cache latency. TripleProv stores such lists of molecules very compactly on disk or in main memory, thus making query resolution fast in many contexts.

In addition to the molecules themselves, the system also maintains a molecule index storing for each key the list of local molecules storing that key (e.g., "key 15123 [Course12] is stored in molecules 23521 [root:Student543] and 23522 [root:Student544]"). This index is particularly useful to answer triple-pattern queries as we explain below in Section 7.

TripleProv extends dipLODocus[RDF] and its native molecule storage in two important ways: i) it introduces new storage structures to store lineage data directly co-located to the instance data and ii) it supports the efficient generation of provenance polynomials during query execution. Figure 1 gives an overview of our system in action, taking as input a SPARQL query (and optionally a provenance granularity level), and returning as output the results of the query along with the provenance polynomials derived during query execution.

5. PROVENANCE POLYNOMIALS

The first question we tackle is how to represent provenance information that we want to return to the user in addition to the results themselves. Beyond listing the various sources involved in the query, we want to be able to characterize the specific ways in which each source contributed to the query results. As summarized in Section 2, there has been quite a bit of work on provenance models and languages recently. Here, we leverage the notion of *provenance polynomials*. However, in contrast to the many recent pieces of work in this space, which tackled more theoretical issues, we focus on the practical realization of this model within a high performance triple store to answer queries seen as useful in practice. Specifically, we focus on two key requirements:

1. the capability to pinpoint, for each query result, the exact source from which the result was selected;

2. the capability to trace back, for each query result, the complete list of sources and how they were combined to deliver a result.

Hence, we support two different provenance operators at the physical level, one called $pProjection$, meeting the first requirement and pinpointing to the exact sources from which the result was drawn, and a second one called $pConstraint$, tracing back the full lineage of the results.

At the logical level, we use two basic operators to express the provenance polynomials. The first one ($\oplus$) to represent unions of sources, and the second ($\otimes$) to represent joins between sources.

Unions are used in two cases when generating the polynomials. First, they are used when a constraint or a projection can be satisfied with triples coming from multiple sources (meaning that there are more than one instance of a particular triple which is used for a particular operation). The following polynomial:

$$l1 \oplus l2 \oplus l3$$

for instance, encodes the fact that a given result can originate from three different sources ($l1$, $l2$, or $l3$, see below Section 5.1 for a more formal definition of the *sources*). Second, unions are also used when multiple entities satisfy a set of constraints or projections (like the collection 'provenanceGlobal' in 7.1).

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

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

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

5.1 Provenance Granularity Levels

One can model 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 results, also supports two levels of granularity. First, a lineage l_i (i.e., an element appearing in a polynomial) can represent the source of a triple, (e.g., the fourth element in a quadruple). We call this granularity level *source-level*. Alternatively, a lineage can represent a quadruple (i.e., a triple plus its corresponding source). 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*.

In addition to those two provenance granularity levels, TripleProv also supports two levels of aggregation to output the results. The default level aggregates the polynomials for all results, i.e., it gives an overview of all triples/sources

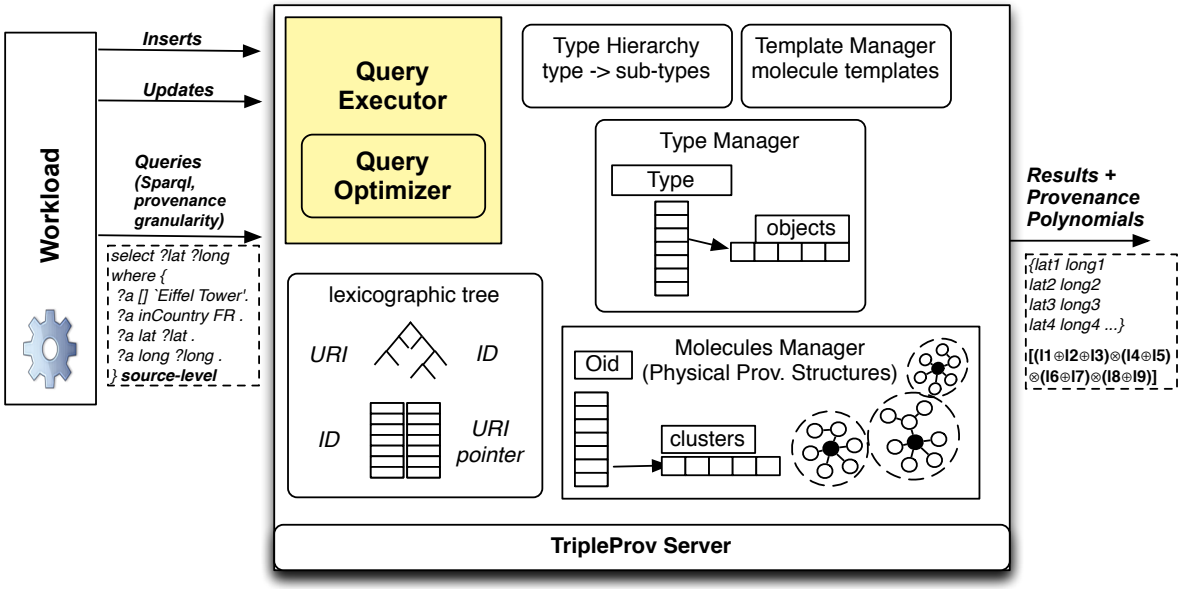


Figure 1: The architecture of TripleProv; the system takes as input queries (and optionally a provenance granularity level), and produces as output query results along with their corresponding provenance polynomials.

used during query execution. The second level provides full provenance details, explaining—for each single result—the way (polynomial) through which this particular result was constructed. Both aggregation levels typically perform similarly (since one is basically derived from the other), hence, we mainly focus on aggregated polynomial results in the following.

6. STORAGE MODELS

We now discuss the RDF storage model of TripleProv, based on our previous contribution [28], and extended with new physical storage structures to store provenance.

6.1 Native Storage Model

RDF Templates When ingesting new triples, TripleProv first identifies RDF subgraphs. It analyzes the incoming data and builds what are termed molecule *templates*. These templates act as data prototypes to create RDF molecules. Figures 2 i) gives a template example that co-locates information relating to Student instances. Once the templates have been defined, the system starts creating molecule identifiers based on the molecule roots (i.e., central molecule nodes) that it identifies in the incoming data.

While creating molecule templates and molecule identifiers, the system takes care of two additional data gathering and analysis tasks. First, it inspects both the schema and instance data to determine all subsumption (subclass) relations between the classes, and maintains this information in a compact *type hierarchy*. In case two unrelated types are assigned to a given instance, the system creates a new virtual type composed of the two types and assigns it to the instance.

RDF Molecules TripleProv stores the primary copy of the RDF data as RDF molecules, which can be seen as hybrid data structures borrowing both from property tables and from RDF subgraphs. They store, for every template defined by the template manager, a compact list of objects connected to the root of the molecule. Figure 2 (ii) gives an example of a molecule. Molecules co-locate data and are template-based, hence can store data extremely compactly. The molecule depicted in Figures 2 (ii), for instance, contains 15 triples (including type information), and would hence require 45 URIs/literals to be encoded using a standard triple-based serialization. Our molecule, on the other hand, only requires the storage 10 keys to be correctly defined, yielding a compression ratio of $1 : 4.5$.

Data (or workload) inspection algorithms can be exploited in order to *materialize* frequent joins though molecules. In addition to materializing the joins between an entity and its corresponding values (e.g., between a student and his/her firstname), one can hence materialize the joins between two semantically related entities (e.g., between a student and his/her advisor) that are frequently co-accessed by co-locating them in the same molecule.

6.2 Storage Model Variants for Provenance

We now turn to the problem of extending the physical data structures of TripleProv to support provenance queries. There are a number of ways one could implement this in our system. A first way of storing provenance data would be to simply annotate every object in the database with its corresponding source. This produces quadruple physical data structures ($SPOL$, where S is the subject of the quadruple, P its predicate, O its object, and L its source), as illustrated in Figure 3, $SPOL$). The main advantage of this variant is its ease of implementation (e.g., one simply has to extend the data structure storing the object to also store the source data). Its main disadvantage, however, is memory consumption since the source data has to be repeated for each triple.

One can try to physically co-locate the source and the triples differently, which results in a different memory consumption profile. One extreme option would be to regroup

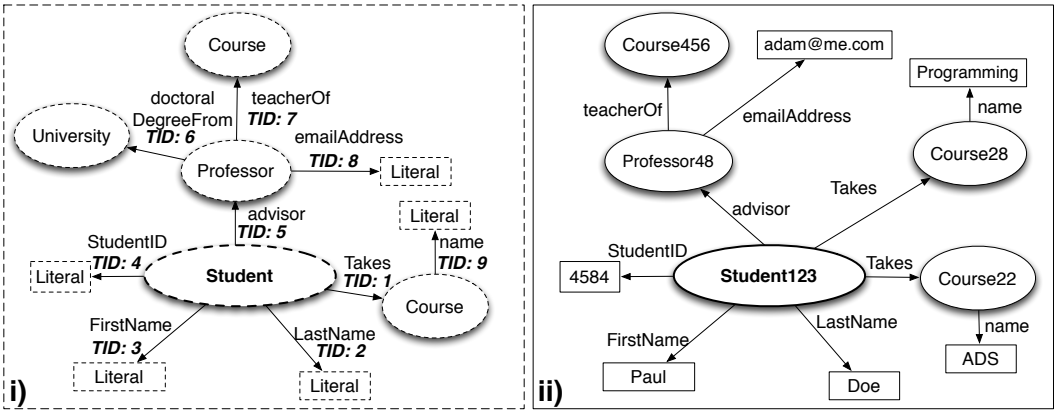


Figure 2: A molecule template (i) along with one of its RDF molecules (ii).

molecules in clusters based on their source ($LSPO$ clustering). This, however, has the negative effect of splitting our original molecules into several structures (one new structure per new source using a given subject), thus braking precomputed 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 which sources to consider during query execution (however, our goal is not to optimize for such provenance queries in the present work).

The last two options for co-locating source data and triples are $SLPO$ and $SPLO$. $SLPO$ co-locates the source data with the predicates, making it technically speaking compelling, since it avoids the duplication of the same source inside a molecule, while at the same time still co-locating all data about a given subject in one structure. $SPLO$, finally, co-locates the source data with the predicates in the molecules. In practice, this last physical structure is very similar to $SPOL$ in terms of storage requirements, since it rarely happens that a given source uses the same predicates with many values. Compared to $SPOL$, it also has the disadvantage of considering a relatively complex structure (PO) in the middle of the physical storage structure (e.g., 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 sources involved for the different subjects. Whenever the data related to a given subject comes from many different sources (e.g., when the objects related to a given predicate come from a wide variety of sources), the overhead caused by repeating the predicate in the $SLPO$ might not be compensated by the advantage of co-location. In such cases, models like $SPLO$ or $SPOL$ might be more appropriate. If, on the other hand, a large portion of the different objects attached to the predicates come from the same sources, then the $SPLO$ model might pay off (see also Section 8 for a discussion on those points). From this analysis, it is evident that no single provenance storage model is overall best—since the performance of such models is somewhat dependent on the queries, of course, but also on the homogeneity / heterogeneity of the datasets considered.

For the reasons described above, we focus below on two very different storage variants in our implementation: $SLPO$, which we refer to as *data grouped by source* in the following (since the data is regrouped by source inside each molecule), and $SPOL$, which we refer to as *annotated provenance* since the source data is placed like an annotation next to the last part of the triple (object). We note that implementing such variants at the physical layer of the database system is a significant effort, since all higher-level calls (i.e., all operators) directly depend on how the data is laid-out on disk and in memory.

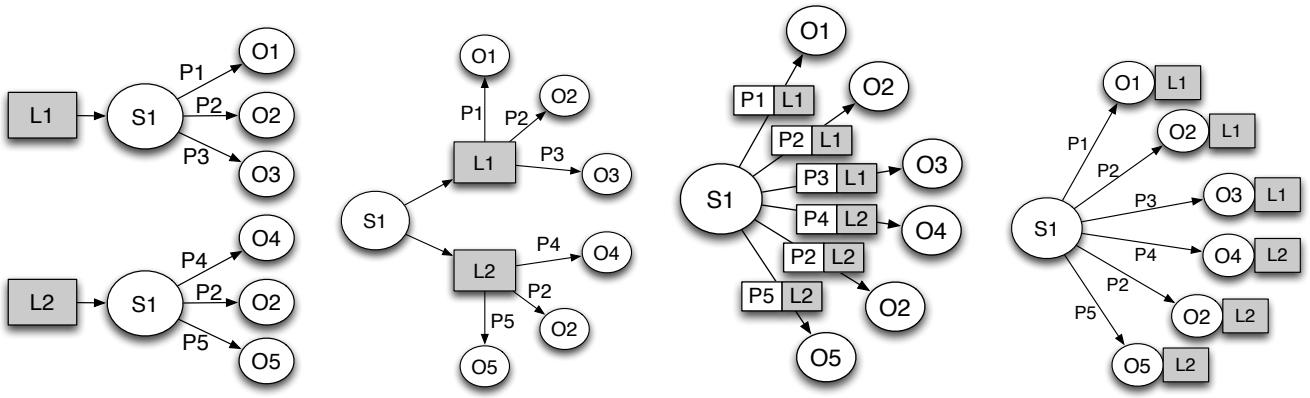
7. QUERY EXECUTION

We now turn to the way we take advantage of the source information stored in the molecules to produce provenance polynomials. We have implemented specific query execution strategies in TripleProv that allow to return a complete record of how the results were produced (including detailed information of key operations like unions and joins) in addition to the results themselves. The provenance polynomials our system produce can be generated at source-level or at triple-level, and both for detailed provenance records and for aggregated provenance records.

7.1 General Query Answering Algorithm

Algorithm 1 gives a simplified view on how simple star-like queries are answered in TripleProv. 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 susceptible of containing data relevant to the query. The molecules are retrieved by taking advantage of the lexicographic tree to translate any unbound variables in the query into keys, and then by using the molecule index to locate all molecules containing those keys (see [28] for details).

In parallel to the classical query execution process, TripleProv keeps track of the various triples and sources that have been instrumental in producing results for the query. For each molecule inspected, our system keeps track of the provenance of any triple matching the current pattern being handled (*checkIfTripleExists*). In a similar fashion, it keeps track of the provenance of all entities being retrieved in the projections (*getEntity*). In case multiple



(a) LSPO Storage (b) SLPO Storage (c) SPLO Storage (d) SPOL Storage

Figure 3: The four different physical storage models identified for co-locating source information (L) with the triples (SPO) inside RDF molecules.

molecules are used to construct the final results, the system keeps track of the local provenance of the molecules by performing a union of the local provenance data using a global provenance structure (provenanceGlobal.union). To illustrate such operations and their results, we describe below the execution of two sample queries.

7.2 Example 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 an entity that has to be identified:

```
select ?lat ?long
where {
 ?a [] ''Eiffel Tower''. (<- 1st constraint)
 ?a inCountry FR .      (<- 2nd constraint)
 ?a lat ?lat .          (<- 1st projection)
 ?a long ?long .        (<- 2nd projection)
}
```

To build the corresponding provenance polynomial, TripleProv first identifies the constraints and projections from the query (see the annotated listing above). The query executor chooses the most selective pattern to start looking up molecules (in this case the first pattern), translates the bound variable ("Eiffel Tower") into a key, and retrieves all molecules containing that key. Each molecule is then inspected in turn to determine whenever both i) the various constraints can be met (*checkIfTripleExists* in the algorithm) and ii) the projections can be correctly processed (*getEntity* in the algorithm). Our system keeps track of the provenance of each result, by joining the local provenance information of each triple used during query execution to identify the result.

Finally, a provenance polynomial such as the following is issued:

$$[(l1 \oplus l2 \oplus l3) \otimes (l4 \oplus l5) \otimes (l6 \oplus l7) \otimes (l8 \oplus l9)].$$

This particular polynomial indicates that the first constraint has been satisfied with lineage $l1$, $l2$ or $l3$, while the second has been satisfied with $l4$ or $l5$. It also indicates that the

Algorithm 1 Simplified algorithm for provenance polynomials generation

Require: SPARQL query q

```
 1: results ← NULL
 2: provenanceGlobal ← NULL
 3: getMolecules ← q.getPhysicalPlan
 4: constraints ← q.getConstraints
 5: projections ← q.getProjections

 6: for all getMolecules do
 7:    provenanceLocal ← NULL

 8:    for all constrains do
 9:       if checkIfTripleExists then
10:          provenanceLocal.join
11:       else
12:          nextMolecule
13:       end if
14:    end for

15:    for all projections do
16:       entity = getEntity(for particular projection)
17:       if entity is NOT EMPTY then
18:          results.add(entity)
19:          provenanceLocal.join
20:       else
21:          nextMolecule
22:       end if
23:    end for

24:    if allConstrainsSatisfied AND allProjectionsAvailable
       then
25:       provenanceGlobal.union
26:    end if
27: end for
```

first projection was processed with elements having a lineage of $l6$ or $l7$, while the second one was processed with elements from $l8$ or $l9$. 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 aggregate or detailed form.

The second example we examine is slightly more involved, 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 [] ?l .
  ?c featureClass P .
  ?c inCountry DE .
  ?c long ?long .
  ?c lat ?lat .
}
```

The query execution starts similarly as for the first sample query. After resolving the first two patterns, the second set of patterns is processed by replacing variable $?l$ with the results derived from the first set, and by joining the corresponding lineage elements.

Processing the query in 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. More complex queries are solved similarly, by starting with the most selective patterns and iteratively joining the results and the provenance information across molecules.

8. PERFORMANCE EVALUATION

To empirically evaluate our approach, we implemented the storage models and query execution strategies described above. Specifically, we implemented two different storage models: SPOL and SLOP. For each model, we support two different levels of provenance granularity: source granularity and triple granularity. Our system does not parse SPARQL queries at this stage (adapting a SPARQL parser is currently in progress), but offers a similar, high-level and declarative API to encode queries using triple patterns. Each query is then encoded into a logical physical plan (a tree of operators), which is then optimized into a physical query plan as for any standard database system. In that sense, we follow the algorithms described above in Section 7.

In the following, we experimentally compare the vanilla version of TripleProv, i.e., the bare-metal system without provenance storage and provenance polynomials generation, to both SPOL and SLOP on two different datasets and workloads. For each provenance storage model, we report results both for generating polynomials at the source and at the triple granularity levels. We also compare our system to 4store[3], where we take advantage of 4store's quadruple storage to encode provenance data as named graphs and manually rewrite queries to return some provenance information to the user (as discussed below, such an approach cannot produce valid polynomials, but is interesting anyhow to illustrate the fundamental differences between TripleProv and standard RDF stores when it comes to provenance).

We note that the RDF storage system that TripleProv extends (i.e., the vanilla version of TripleProv) has already been compared to a number of other well-known database systems, including Postgres, AllegroGraph, BigOWLIM, Jena, Virtuoso, and RDF 3X (see [28] and [5]). The system is on average 30 times faster than the fastest RDF data management system we have considered (RDF-3X) for LUBM queries, and on average 350 times faster than the fastest system we have considered (Virtuoso) on more complex analytics.

8.1 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 and running Ubuntu 12.04.3 LTS (Precise Pangolin). All data were stored on a recent 3 TB Serial ATA disk.

8.2 Datasets

We used two different sources for our data: the Billion Triples Challenge (BTC)[4] and the Web Data Commons (WDC) [21].[5] Both datasets are collections of RDF data gathered from the Web. They represent two very different kinds of RDF data. The Billion Triple Challenge dataset was crawled based on datasets provided by Falcon-S, Sindice, Swoogle, SWSE, and Watson using the MultiCrawler/SWSE framework. The Web Data Commons project extracts all Microformat, Microdata and RDFa data from the Common Crawl Web corpus, the largest and most up-to-data Web corpus that is currently available to the public, and provides the extracted data for download in the form of RDF-quads and also in the form of CSV-tables for common entity types (e.g., products, organizations, locations, etc.).

Both datasets represent typical collections of data gathered from multiple sources, thus tracking provenance for them seems to precisely address the problem we focus on. We consider around 115 million triples for each dataset (around 25GB). To sample the data, we first pre-selected quadruples satisfying the set of considered queries. Then, we randomly sampled additional data up to 25GB. Both datasets are available for download on our website[6].

8.3 Workloads

We consider two different workloads. For BTC, we use eight existing queries originally proposed in [22]. In addition, we added two queries with UNION and OPTIONAL clauses, which we thought were missing in the original set of queries. Based on the queries used for the BTC dataset, we wrote 7 new queries for the WDC dataset, encompassing different kinds of typical query patterns for RDF, including star-queries of different sizes and up to 5 joins, object-object

[3] http://4store.org/
[4] http://km.aifb.kit.edu/projects/btc-2009/
[5] http://webdatacommons.org/
[6] http://exascale.info/tripleprov

joins, object-subject joins, and triangular joins. In addition, we included two queries with UNION and OPTIONAL clauses. As for the data, the workloads we consdiered are available on our website.

8.4 Experimental Methodology

As is typical for benchmarking database systems (e.g., for tpc-x[7]), we include 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 systems, and then repeat the process ten times (i.e., we run for each system we benchmark a total of 11 batches, each containing all the queries we consider in sequence). We report the mean values 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; We measured instead the query execution times inside the database system only.

8.5 Variants Considered

As stated above, we implemented two storage models (grouped/co-located and annotated) in TripleProv and for each model we considered two granularity levels for tracking provenance (source and triple). This gives us four different variants to compare against the vanilla version of our system. Our goal is in that sense to understand the various trade-offs of the approaches and to assess the performance penalty caused by enabling provenance. We use the following abbreviations to refer to the different variants in the following:

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

SG: source-level granularity, provenance data grouped by source;

SA: source-level granularity, annotated provenance data;

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

TA: triple-level granularity, annotated provenance data.

8.6 Comparison to 4Store

First, we start by an informal comparison with 4Store to highlight the fundamental differences between our provenance-enabled system and a quad-store supporting named graphs.

While 4Store storage takes into account quads (and thus, source data can be explicitly stored), the system does not support the generation of detailed provenance polynomials tracing back the lineage of the results. Typically, 4Store simply returns standard query results as any other RDF store. However, one can try to simulate some basic provenance capabilities by leveraging the *graph* construct in SPARQL and extensively rewriting the queries by inserting this construct for each query pattern.

As an example, rewriting the first sample query we consider above in Section 7.1 would result in the following:

[7] *http://www.tpc.org/*

```
select ?lat ?long ?g1 ?g2 ?g3 ?g4
where {
  graph ?g1 {?a [] "Eiffel Tower" . }
  graph ?g2 {?a inCountry FR .    }
  graph ?g3 {?a lat ?lat .        }
  graph ?g4 {?a long ?long .      }
}
```

However, such a query processed in 4store would obviously not produce full-fledged provenance polynomials. Rather, a simple list of concatenated sources would be returned, whether or not they were in the end instrumental to derive the final results of the query, as follows:

```
lat long l1 l2 l4 l4, lat long l1 l2 l4 l5,
lat long l1 l2 l5 l4, lat long l1 l2 l5 l5,
lat long l1 l3 l4 l4, lat long l1 l3 l4 l5,
lat long l1 l3 l5 l4, lat long l1 l3 l5 l5,

lat long l2 l2 l4 l4, lat long l2 l2 l4 l5,
lat long l2 l2 l5 l4, lat long l2 l2 l5 l5,
lat long l2 l3 l4 l4, lat long l2 l3 l4 l5,
lat long l2 l3 l5 l4, lat long l2 l3 l5 l5,

lat long l3 l2 l4 l4, lat long l3 l2 l4 l5,
lat long l3 l2 l5 l4, lat long l3 l2 l5 l5,
lat long l3 l3 l4 l4, lat long l3 l3 l4 l5,
lat long l3 l3 l5 l4, lat long l3 l3 l5 l5.
```

The listing above consists of all permutations of values bound to variables referring to data used to answer the original query (?*lat*, ?*long*). Additionally, all named graphs used to resolve the triple patterns from the query (relating to variables ?$g1$, ?$g2$, ?$g3$, and ?$g4$) are also integrated. Obviously, this type of outcome is insufficient for correctly tracing back the provenance of the results.

Whereas in TripleProv the answer to the original query (without the graph clauses) would be as follows:

```
lat long
```

with, in addition, the following compact provenance polynomial:

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

8.7 Query Execution Times

Figures 4 and 5 give the query execution times for the BTC dataset, while Figures 6 and 7 present similar results for WDC. We also explicitly give the overhead generated by our various approaches compared to the non-provenance-enabled (vanilla) version, in Figures 8 and 10 for BTC, and in Figures 9 and 11 for WDC, respectively.

query #	V	SG	SA	TG	TA
01	0.62	1.47	1.06	1.20	1.03
02	25.78	44.04	43.87	44.87	43.14
03	1.06	1.78	1.82	1.81	1.79
04	111.11	200.28	183.34	201.99	180.04
05	258.41	464.09	423.46	467.12	416.14
06	35.80	109.60	77.09	160.29	78.07
07	1347.44	2258.41	2327.51	2344.10	2281.88
08	4.03	5.60	4.98	5.54	4.94
09	0.0004	0.0006	0.0004	0.0006	0.0005
10	10.93	14.98	17.18	16.69	16.94

Figure 4: Query execution times (in seconds) for the BTC dataset

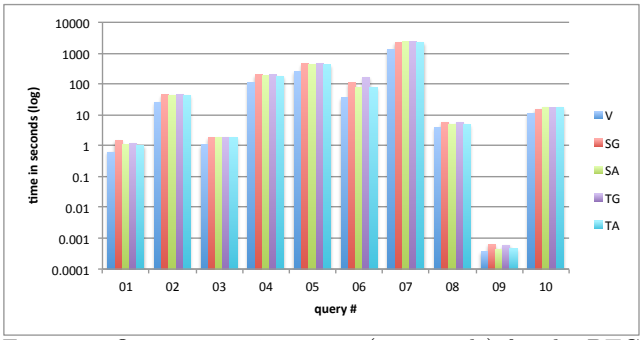


Figure 5: Query execution times (in seconds) for the BTC dataset (logarithmic scale)

query #	V	SG	SA	TG	TA
01	0.0007	0.0017	0.0014	0.0024	0.0015
02	0.0006	0.0010	0.0006	0.0009	0.0006
03	0.0015	0.0034	0.0034	0.0048	0.0034
04	4.6637	8.6492	6.9617	8.8299	7.1642
05	0.1053	0.2395	0.2604	0.3320	0.3128
06	0.0187	0.0469	0.0532	0.0749	0.0733
07	0.0021	0.0039	0.0041	0.0051	0.0040

Figure 6: Query execution times (in seconds) for the WDC dataset

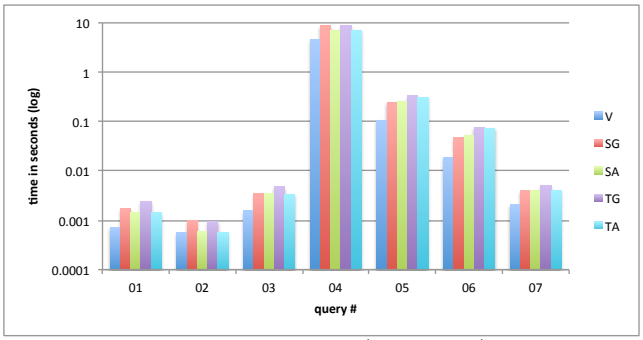


Figure 7: Query execution times (in seconds) for the WDC dataset (logarithmic scale)

query #	SG	SA	TG	TA
01	136.79%	70.29%	92.72%	66.69%
02	70.84%	70.18%	74.03%	67.32%
03	69.09%	72.68%	72.01%	69.65%
04	80.26%	65.01%	81.79%	62.04%
05	79.60%	63.87%	80.77%	61.04%
06	206.11%	115.31%	347.68%	118.05%
07	67.61%	72.74%	73.97%	69.35%
08	38.98%	23.58%	37.68%	22.69%
09	70.70%	21.36%	63.80%	24.64%
10	37.00%	57.10%	52.62%	54.88%

Figure 8: Overhead of tracking provenance compared to the vanilla version of the system for the BTC dataset

query #	SG	SA	TG	TA
01	139.98%	98.63%	238.70%	103.93%
02	74.69%	5.70%	66.35%	2.64%
03	121.07%	121.27%	206.97%	118.01%
04	85.46%	49.27%	89.33%	53.61%
05	127.40%	147.20%	215.18%	196.99%
06	150.51%	184.59%	300.12%	291.72%
07	90.77%	96.91%	146.81%	93.85%

Figure 9: Overhead of tracking provenance compared to the vanilla version of the system for the WDC dataset

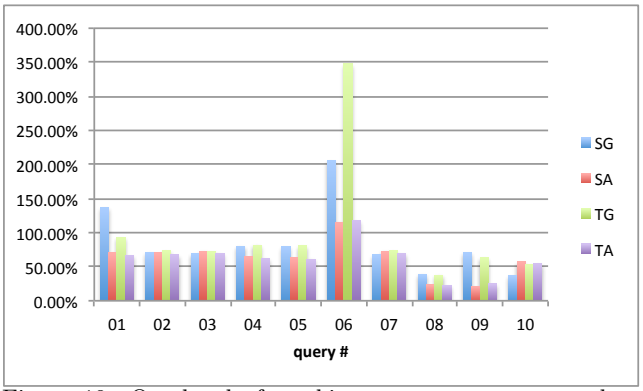


Figure 10: Overhead of tracking provenance compared to the vanilla version of the system for the BTC dataset

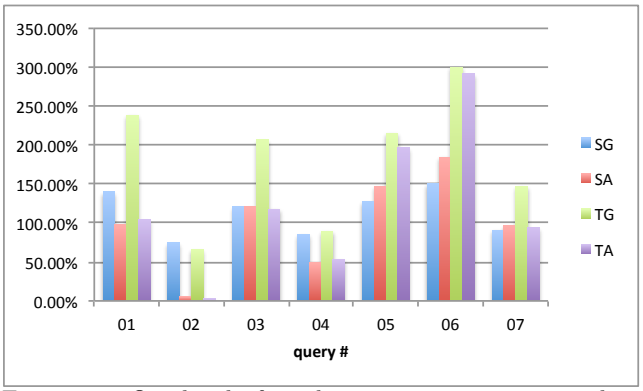


Figure 11: Overhead of tracking provenance compared to the vanilla version of the system for the WDC dataset

Overall, the performance penalty created by tracking provenance in TripleProv ranges from a few percents to almost 350%. Clearly, we observe a significant difference between the two main provenance storage models implemented (SG vs SA and TG vs TA). Retrieving data from co-located structures takes about 10%-20% more time than from simply annotated graph nodes. We experimented with various physical structures for SG and TG, but could not significantly reduce this overhead, caused by the additional look-ups and loops that have to be considered when reading from extra physical data containers.

We also notice considerable difference between the two granularity levels (SG vs TG and SA vs TA). Clearly, the more detailed triple-level provenance granularity requires more time for query execution than the simpler source-level, because of the more complete physical structures that need to be created and updated while collecting the intermediate results sets.

Also, we observe some important differences between the query execution times from the two datasets we used, even for very similar queries (01-05 map directly from one dataset onto the other; 09BTC maps to 06WDC and 10BTC maps to 07WDC). Clearly, the efficiency of our provenance polynomial generation on a given query depends upon underlying data characteristics. One important dimension in that context is the heterogeneity—in terms of number of sources providing the data—of the dataset. The more heterogeneous the data, the better the annotated storage model performs, since this model makes no attempt at co-locating data w.r.t. the sources and hence avoids additional look-ups when many sources are involved. On the other hand, the more structured the data, the better the co-located models perform.

Finally, we briefly discuss two peculiar results appearing in Figure 10 fore queries 01 and 06. For query 01, the reason behind the large disparity in performance has to do with the very short execution times (at the level of 10^{-3} second), which cannot be measured more precisely and thus introduces some noise. The performance overheard for query 06 is caused by a very large provenance record on one hand, and a high heterogeneity in terms of sources for the elements that are used to answer the query.

8.8 Loading Times & Memory Consumption

Finally, we discuss the loading times and memory consumption for the various approaches. Figure 12 reports results for the BTC dataset, while Figure 13 provides similar figures for the WDC dataset.

Referring to loading times, the more complex co-located storage model requires more computations to load the data than the simpler annotation model, which obviously increases the time needed to load data. In terms of memory consumption, the experimental results confirm our analysis from Section 6; The datasets used for our experiments are crawled from the Web, and hence consider data collated from a wide variety of sources, which results in a high diversification of the sources for each subject. As we explained in Section 6, storage structures such as *SPLO* or *SPOL* are more appropriate in such a case.

	V	G	A
Loading Time (min)	23.32	27.9	26.8
Memory Consumprion (GB)	36.26	53.62	39.54

Figure 12: Loading times and memory consumption for the BTC dataset

	V	G	A
Loading Time (min)	27.46	67.78	30.56
Memory Consumprion (GB)	42.53	66.22	50.29

Figure 13: Loading times and memory consumption for the WDC dataset

9. CONCLUSIONS

In this paper, we described TripleProv, an open-source and efficient system for managing RDF data while also tracking provenance. To the best of our knowledge, this is the first work that translates theoretical insights from the database provenance literature into a high-performance triple store. TripleProv not only implements simple tracing of sources for query answers, but also considers fine-grained multilevel provenance. In this paper, we implemented two possible storage models for supporting provenance in RDF data management systems. Our experimental evaluation shows that the overhead of provenance, even though considerable, is acceptable for the resulting provision of a detailed provenance trace. We note that both our query algorithms and storage models can be reused by other databases (e.g., considering property tables or subgraph storage structures) with only small modifications. As we integrate a myriad of datasets from the Web, provenance becomes a critical aspect in ascertaining trust and establishing transparency [12]. TripleProv provides the infrastructure needed for exposing and working with fine-grained provenance in RDF-based environments.

We plan to continue developing TripleProv in several directions. First, we plan to extend provenance support to the distributed version of our database system. Also, we plan to extend TripleProv with a dynamic storage model to enable further optimization between memory consumption and query execution times. We also hope to bring down the overall cost of tracing provenance within the system. In terms of provenance, we plan to extend TripleProv to output PROV, which would open the door to queries over the provenance of the query results and the data itself - merging both internal and external provenance. Such an approach would facilitate trust computations over provenance that take into account the history of the original data as well as how it was processed within the database. In addition, we aim to allow for adaptive query execution strategies based on provenance. For example, executing a query that would only consider a set of particularly trusted data sources.

Acknowledgements

This work was supported by the Swiss National Science Foundation under grant number PP00P2_128459 and by the Dutch national program COMMIT.

10. REFERENCES

[1] 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*, pages 613–622. ACM, 2005.

[2] J. Cheney, L. Chiticariu, and W.-C. Tan. *Provenance in databases: Why, how, and where*, volume 1. Now Publishers Inc, 2009.

[3] 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)*, 2(2):1534–1537, 2009.

[4] 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*, ISWC'12, pages 625–640, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudre-Mauroux. BowlognabenchâĂŤbenchmarking rdf analytics. In K. Aberer, E. Damiani, and T. Dillon, editors, *Data-Driven Process Discovery and Analysis*, volume 116 of *Lecture Notes in Business Information Processing*, pages 82–102. Springer Berlin Heidelberg, 2012.

[6] 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.

[7] 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*, ISWC '09, pages 196–212, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] 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*, ICDT '13, pages 153–164, New York, NY, USA, 2013. ACM.

[9] 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*, pages 31–40. ACM, 2007.

[10] P. Groth, Y. Gil, J. Cheney, and S. Miles. Requirements for provenance on the web. *International Journal of Digital Curation*, 7(1), 2012.

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

[12] P. T. Groth. Transparency and reliability in the data supply chain. *IEEE Internet Computing*, 17(2):69–71, 2013.

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

[14] 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*, ESWC 2009 Heraklion, pages 5–20, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[16] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013. Artificial Intelligence, Wikipedia and Semi-Structured Resources.

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

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

[19] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, Nov. 2010.

[20] 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*, 27(6):743–756, June 2011.

[21] H. Mühleisen and C. Bizer. Web data commons - extracting structured data from two large web corpora. In C. Bizer, T. Heath, T. Berners-Lee, and M. Hausenblas, editors, *LDOW*, volume 937 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[22] 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*, pages 627–640. ACM, 2009.

[23] 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.

[24] 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. Technical report, W3C, 2010.

[25] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides. On provenance of queries on semantic web data. *IEEE Internet Computing*, 15(1):31–39, Jan. 2011.

[26] O. Udrea, D. R. Recupero, and V. Subrahmanian. Annotated rdf. *ACM Transactions on Computational Logic (TOCL)*, 11(2):10, 2010.

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

[28] 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*, ISWC'11, pages 778–793, Berlin, Heidelberg, 2011. Springer-Verlag.

[29] 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.

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