Answering Provenance-Aware Queries on RDF Data Cubes under Memory Budgets

Luis Galárraga[1,2], Kim Ahlstrøm[2], Katja Hose[2], and Torben Bach Pedersen[2]

[1]INRIA; [2]Department of Computer Science, Aalborg University
luis.galarraga@inria.fr, {kah|khose|tbp}@cs.aau.dk

Abstract. The steadily-growing popularity of semantic data on the Web and the support for aggregation queries in SPARQL 1.1 have propelled the interest in Online Analytical Processing (OLAP) and data cubes in RDF. Query processing in such settings is challenging because SPARQL OLAP queries usually contain many triple patterns with grouping and aggregation. Moreover, one important factor of query answering on Web data is its provenance, i.e., metadata about its origin. Some applications in data analytics and access control require to augment the data with provenance metadata and run queries that impose constraints on this provenance. This task is called provenance-aware query answering. In this paper, we investigate the benefit of caching some parts of an RDF cube augmented with provenance information when answering provenance-aware SPARQL queries. We propose provenance-aware caching (PAC), a caching approach based on a provenance-aware partitioning of RDF graphs, and a benefit model optimized for RDF cubes and SPARQL queries with aggregation. Our results on real and synthetic data show that PAC outperforms significantly the LRU strategy and the Jena TDB native caching in terms of hit-rate and response time.

1 Introduction

In the last years we have seen a steady increase of the amount of Linked Data available on the Web. This data spans over a wide variety of topics ranging from commonsense information to specialized domains such as governmental information, media, life sciences, etc. The data is normally published in RDF [24] and queried using SPARQL [25]. The extended capabilities of SPARQL 1.1 —notably the support for aggregation queries— have motivated the publication of multidimensional data, i.e., data cubes, in RDF [8, 12, 17, 28] . The analysis of multidimensional data is a standard task in data warehouses and is known as OLAP (Online Analytical Processing). The publication of the QB vocabulary [9] has served as a bridge between the semantic web and OLAP communities.

Imposing constraints on the provenance of query results, a task known as provenance-aware query processing, is crucial in a setting with data coming from multiple independent sources. The provenance of a piece of data is a set of assertions about its origin. Provenance metadata can be used, e.g., to restrict OLAP operations to data meeting certain quality constraints [21], or to implement access control policies [3]. The importance of provenance data management motivated the creation of the PROV ontology [20] (PROV-O), the W3C standard to represent provenance information for RDF data. PROV-O provides the data model to describe a set of provenance entities,

i.e., RDF resources, which are assigned to the triples in an RDF dataset. There exist multiple representations for provenance-augmented RDF data, such as named graphs and reification. These representations are, though, not exempt from performance issues for very complex queries [22] due to the additional complexity added by the provenance metadata. In this paper we use the named graph representation [4].

A recent formulation for provenance-aware query answering divides the query in two parts: a provenance query and an analytical query [1, 32]. The provenance query imposes constraints on the provenance entities of the triples that should be considered to answer the analytical query on the actual data. In a representation of provenance entities using named graphs, this amounts to adding a set of FROM clauses to the analytical query for each provenance entity reported by the provenance query. As shown in [1], query response time is seriously affected in frameworks, such as Jena, as the number of FROM clauses increases. This happens because the query engine has to fetch a large number of intermediate results from disk in order to answer the analytical query.

In this paper we propose to alleviate the aforementioned phenomenon by caching some *fragments* of the RDF graph in memory, so that the analytical queries benefit from fast access to the data. Our caching strategy, called provenance-aware caching (PAC), selects the most *beneficial* fragments that fit within a memory budget. Since we are interested in RDF cubes, we assume our queries are OLAP queries, i.e., SPARQL queries with aggregation, grouping and filtering. This assumption reduces the space of possible queries we optimize for, without the need of an explicit query-load. We show that for OLAP analytical queries, it suffices to cache a small percentage of the dataset in order to achieve a significant speed-up in query response time. This is particularly convenient for memory-constrained settings. In summary our contributions are:

- A fragmentation scheme tailored for provenance-augmented RDF graphs.
- The formulation of the budgeted provenance-enabled fragment selection problem: The problem of selecting a set of fragments for caching so that as many OLAP queries as possible benefit from fast access to cached data.
- A query rewriting algorithm to answer analytical queries from a set of named graphs and cached fragments.
- A study of the impact of caching on the performance of Jena TDB for provenance-aware SPARQL aggregation queries.

The remainder of this paper is structured as follows. Section 2 introduces the basic concepts of RDF cubes, SPARQL aggregation queries and provenance in RDF. In Section 3, we introduce the fragment selection problem with a memory budget for RDF cubes with provenance information. This is followed by an experimental evaluation in Section 4 and a discussion of related work in Section 5. Section 6 concludes the paper.

2 Preliminaries

2.1 RDF Cubes

In compliance with the official RDF specification [24], we define an *RDF triple* t (or simply a *triple*), as $t = \langle s, p, o \rangle \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{P} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, where s is the subject, p is the predicate, and o is the object. In this definition, $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{L}$ are countably infinite sets of IRIs, blank nodes and literals. In addition, we define the set of predicates $\mathcal{P} \subseteq \mathcal{U}$ and the set of classes $\mathcal{C} \subseteq \mathcal{U}$. An *RDF dataset* $\mathcal{K}$ is a set of RDF triples. Since RDF

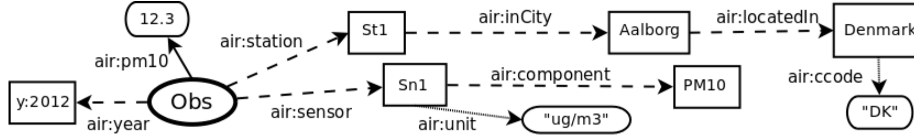

Fig. 1: Observation with 3 dimensions: Year, Station, and Sensor. Measure predicates $\mathcal{P}_{\mathcal{M}}$ are in solid line style, whereas attribute predicates $\mathcal{P}_{\mathcal{A}}$ use dotted lines. The dashed edges correspond to the dimension properties $\mathcal{P}_{\mathcal{D}}$.

defines a graph-like data model, we also refer to RDF datasets as *RDF graphs*. An *RDF cube* $\mathcal{K}_c = \{\mathcal{O}, \mathcal{D}, \mathcal{P}_{\mathcal{M}}, \mathcal{P}_{\mathcal{D}}, \mathcal{P}_{\mathcal{A}}, \Delta\}$ is an RDF graph defined in terms of:

- A set of observations $\mathcal{O} \subseteq \mathcal{U}$.
- A set of measure predicates $\mathcal{P}_{\mathcal{M}} \subseteq \mathcal{P}$, defined between observations and literal numerical values. These predicates are the target of aggregation in OLAP queries.
- A set of dimensions $\mathcal{D}$. Observations are defined by their coordinates in $\mathcal{D}$. Each dimension consists of a hierarchy of classes that describe an observation at different degrees of specificity. Each class defines a *level* in the hierarchy.
- A set of dimension predicates $\mathcal{P}_{\mathcal{D}} \subseteq \mathcal{P}$. These predicates define the coordinates of an observation in the dimensions $\mathcal{D}$.
- A set of level attributes $\mathcal{P}_{\mathcal{A}} \subseteq \mathcal{P}$. Level attributes are predicates defined on the class levels of the dimensions. They are often used for grouping and filtering.
- A function $\Delta : \mathcal{D} \to \mathcal{H}$ that assigns each dimension in $\mathcal{D}$ a *class hierarchy* from the set of hierarchies $\mathcal{H}$. A class hierarchy $H = (L, \prec_L, \gamma, \sigma) \in \mathcal{H}$ consists of a set of class levels $L \subseteq \mathcal{C}$ and a partial order $\prec_L$ on L with a single greatest element. The function $\gamma : L \to 2^{\mathcal{P}_{\mathcal{D}}}$ assigns each class level in L a set of dimension predicates, whereas the function $\sigma : L \to 2^{\mathcal{P}_{\mathcal{A}}}$ assigns each class level a set of level attributes.

Example 1. Consider an RDF cube representing a database of air pollution measurements. Each measurement corresponds to an observation in the cube model. A measurement of 12.3 $\mu g/m^3$ of the pollutant PM_{10} corresponds to the triple $\langle$*Obs, air:pm10, 12.3*$\rangle$ depicted in Figure 1. It follows that $Obs \in \mathcal{O}$ and $air{:}pm10 \in \mathcal{P}_{\mathcal{M}}$. The triple $\langle$*Obs, air:station, St1*$\rangle$ defines the coordinates of Obs in the Station dimension. There are three dimensions in our example, i.e., $\mathcal{D} = \{Year, Station, Sensor\}$. The predicates *air:year*, *air:station*, *air:sensor* $\in \mathcal{P}_{\mathcal{D}}$ are dimension predicates. The function Δ associates each dimension with a class hierarchy. For example, the dimention *Station* is mapped to a hierarchy $H = (L, \prec_L, \gamma, \sigma)$ defined by the classes $L = \{Station, City, Country\}$ and the order $Station \prec_L City \prec_L Country$. In addition, it holds that $\gamma(Country) = \{air{:}locatedIn\}$. The country's code can be modeled as a level attribute $air{:}ccode \in \mathcal{P}_{\mathcal{A}}$ of the *Country* class. Thus, $\sigma(Country) = \{air{:}ccode\}$.

2.2 SPARQL queries

Due to space constraints, we do not provide a rigorous definition of SPARQL queries; instead we resort to the formulation used in [15] to define SPARQL aggregation queries. These are the most common types of OLAP queries. We define a *triple pattern* as a triple $\hat{t} = \langle s, p, o \rangle \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{P} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$. The set $\mathcal{V}$ is a set of variables with $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}) \cap \mathcal{V} = \emptyset$. A *basic graph pattern* G_p is a set

of triple patterns. A SPARQL select query Q is an expression of the form "SELECT V F WHERE $\hat{G}_p$ GROUP BY V' HAVING c" with $V \cup F \neq \emptyset$. In this definition $V \subseteq \mathcal{V}$ is the set of projection variables and F is a set of aggregation expressions of the form $f(g(\hat{V}))$ where $f \in \{\mathit{COUNT}, \mathit{SUM}, \mathit{AVG}, \mathit{MIN}, \mathit{MAX}\}$ and $g(\hat{V})$ is a numerical expression on the set of aggregated variables $\hat{V} \subseteq \mathcal{V}$. $\hat{G}_p$ is an extended basic graph pattern, potentially containing *OPTIONAL* and *FILTER* clauses. The set of grouping variables is a superset of the projection variables ($V' \supseteq V$) whereas c is a Boolean expression on $V \cup F$. The GROUP BY and HAVING clauses are optional.

Example 2. The following SPARQL query computes the maximal concentration of PM_{10} per city in Denmark in 2012 according to the schema in Figure 1.

```
SELECT ?city (MAX(?ms) as ?max) WHERE {
 ?obs air:pm10 ?ms. ?obs air:year y:2012. ?obs air:station ?st.
 ?st air:inCity ?city. ?city air:locatedIn Denmark.
} GROUP BY ?city
```

2.3 Provenance

There exist multiple provenance models for RDF in the literature [7]; in this paper, we focus on *workflow provenance* [26]: the history of a unit of information from its sources to its current state. This provenance is modeled using RDF by assigning each triple an RDF resource, which we call its *provenance entity*. The set of statements describing the provenance entities of an RDF graph is a *provenance graph*. The PROV Ontology [20] is the W3C specification to model provenance graphs. In this model a provenance entity can represent a data resource such as a file, a web page or the intermediate result of a data transformation process. Any operation on data is modeled as an activity in PROV-O. Those activities can be directly or indirectly carried out by agents: people, organizations or even computer programs.

If $\mathcal{I}$ is a set of *provenance entities* and $f : \mathcal{K} \rightarrow \mathcal{I}$ is a provenance function on the triples of an RDF graph $\mathcal{K}$, a *provenance-augmented RDF graph* $\mathcal{K}_\mathcal{I}$ is a set of pairs $\langle t, f(t) \rangle$, which can also be seen as a set of quadruples $\langle s, p, o, i \rangle$, where $i \in \mathcal{I}$. We can also model a provenance-augmented RDF graph as a set of RDF sub-graphs, each containing the triples associated to the same provenance entity. In this view $\mathcal{K}_\mathcal{I} = \{\mathcal{K}_{i_1}, \ldots, \mathcal{K}_{i_n}\}$ $(i_1, \ldots, i_n \in \mathcal{I})$ and each RDF sub-graph is a named graph with label i. We define a *provenance-augmented cube* as an RDF cube whose triples have been augmented with provenance entities.

2.4 Provenance-Aware Query Answering

Given a provenance-augmented RDF graph $\mathcal{K}_\mathcal{I} = \{\mathcal{K}_{i_1}, \ldots, \mathcal{K}_{i_n}\}$ and a provenance graph $\mathcal{G}_\mathcal{I}$ describing the set of provenance entities $i_1, \ldots, i_n \in \mathcal{I}$, a provenance-aware query is a pair of SPARQL queries $\langle q_p, q_a \rangle$ [1, 32]. q_p is known as the *provenance query* and is defined on $\mathcal{G}_\mathcal{I}$. The provenance query is designed so that it returns a set of provenance entities $I \subseteq \mathcal{I}$. Those provenance entities are used to restrict the scope of the *analytical query* q_a on the RDF graph $\mathcal{K}_\mathcal{I}$ to those subgraphs with labels in I. The problem of answering provenance-aware queries on RDF data has been studied in the last years [1, 6, 31, 32]. If provenance information is modeled using named graphs

(where the labels are provenance entities), the naive strategy is to augment the analytical query with a FROM clause for every provenance entity reported by the provenance query. In [1] it is shown that this strategy performs poorly in frameworks such as Jena for non-selective provenance queries. A strategy called *full materialization* [32] proposes to first fetch all the triples from the named graphs to main memory, and then run the analytical query on the union of those graphs. While this strategy generally outperforms the naive approach, it is not free from performance issues on memory-constrained systems. Nonetheless, we observe that both strategies require the retrieval of a large number of triples from disk. Therefore, we study the impact of keeping some parts or *fragments* of the dataset in main memory so that queries can benefit from fast access to the data.

3 The Budgeted Provenance-Enabled Fragment Selection Problem

In this section, we define the *budgeted provenance-enabled fragment selection problem*. This is the problem of selecting a set of provenance-enabled RDF data fragments for caching so that we reduce the response time of the analytical query when answering a provenance-enabled query. This is achieved by maximizing the amount of data that is retrieved from the cache. In the following, we describe the three components of our approach, namely the fragmentation strategy, the cost-benefit model, and the query rewriting algorithm. We highlight that our method is query-load agnostic, thus it aims at optimizing for as many queries as possible in the space of analytical queries.

3.1 Fragmentation strategy

A fragmentation strategy defines how to split a dataset into smaller parts, i.e., fragments. Once the dataset is fragmented, we can decide which parts to put in the cache. We start by defining a fragment for provenance-augmented RDF graphs.

Definition 1. *A fragment signature s_ϕ is a quadruple $\langle s, p, o, i \rangle$ such that each component can be a constant or a variable. We say a quadruple q in a provenance-augmented RDF graph $\mathcal{K}_\mathcal{I}$ matches a fragment signature if there exists an instantiation ρ for the variables in the signature such that $\rho(s_\phi) = q$. The set ϕ of quadruples that match s_ϕ in a provenance-augmented RDF graph $\mathcal{K}_\mathcal{I}$ is a fragment.*

Definition 2. *A provenance-aware fragment tree Φ consists of a set of fragments and a partial order $\sqsupseteq_f$ on those fragments. A fragment ϕ subsumes a fragment ϕ', denoted by $\phi \sqsupseteq_f \phi'$, iff $s_{\phi'} \Rightarrow s_\phi$. If $\phi \sqsupseteq_f \phi'$ then $\phi \supseteq \phi'$.*

Figure 2 shows a provenance-aware fragment tree describing some fragments from the cube introduced in Example 1 with two provenance entities *pr:e1* and *pr:e2*. The root fragment contains the trivial signature, i.e., the signature that matches all quadruples in the dataset. In the second level, we have signatures with restrictions on the provenance entities of the quadruples. The fragments in the third level have restrictions on both the predicate and the provenance entity. Fragments always subsume their children.

Algorithm 1 describes the method to construct a provenance-aware fragment tree given a provenance-augmented RDF graph. The algorithm first initializes the tree with the trivial signature (line 1). Then for each quadruple in the dataset, the method constructs signatures with (a) bound provenance entity, (b) bound provenance entity and

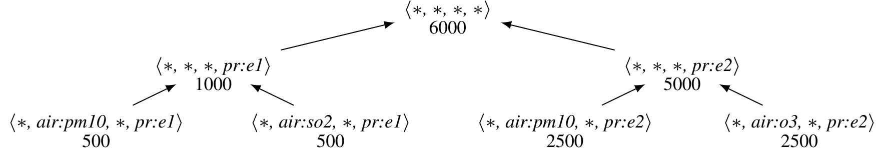

Fig. 2: A provenance-aware fragment tree. The size of each fragment is noted below.

predicate (line 3), and (c) bound provenance entity, predicate and object for the *rdf:type* predicate (lines 4-5). The latter step accounts for the typically large size of the *rdf:type* predicate. If the tree does not contain a signature, the signature is initialized (by setting its size to 1 in line 8) and added to the tree (line 9). Otherwise, the size of the signature is incremented to account for the current quadruple (line 11).

Algorithm 1: BuildPATree

```
   Input: a provenance-augmented RDF graph: 𝒦_ℐ
   Output: a provenance-aware fragment tree: Φ
1  Φ := {⟨∗, ∗, ∗, ∗⟩}
2  foreach q := ⟨s, p, o, i⟩ ∈ 𝒦_ℐ do
3      Φ' = {⟨∗, ∗, ∗, i⟩, ⟨∗, p, ∗, i⟩}
4      if p = "rdf:type" then
5          Φ' = Φ' ∪ {⟨∗, p, o, i⟩}
6      foreach s'_φ ∈ Φ' do
7          if s'_φ ∉ Φ then
8              s'_φ.size := 1
9              Φ = Φ ∪ s'_φ
10         else
11             s'_φ.size := s'_φ.size + 1
12 return Φ
```

When clear from the context, we drop the distinction between a fragment and its signature and refer to both as ϕ. Finally, we highlight that Algorithm 1 produces fragmentation schemes with redundancy. Whether we allow redundancy or not in the set of selected fragments depends on the benefit model. We elaborate on this in the following.

3.2 Cost-Benefit Model

The cost-benefit model quantifies the price we have to pay for caching a fragment, as well as the amount of saved response time induced by using the cached fragment to answer queries. In line with approaches for view materialization [15] we use the number of quadruples matched by the fragment's signature as its cost, i.e., $cost(\phi) = |\phi|$. We say a fragment ϕ is relevant to a provenance-aware query $q = \langle q_p, q_a \rangle$ if at least one of the quadruples in ϕ can be used to answer q—and none of them could lead to a wrong answer. For example, consider a provenance query q_p with result *pr:e1* and the analytical query q_a from Example 2. In this case the analytical query is restricted by the provenance query to the quadruples with provenance *pr:e1*. We say that the fragments ϕ and ϕ' with signatures s_ϕ =⟨*, *air:pm10*, *, *pr:e1*⟩ and $s_{\phi'}$ =⟨*, *, *, *pr:e1*⟩ are

relevant to q. In contrast, the fragment ϕ'' with signature $s_{\phi''} = \langle$*, *p:unitPrice*, *, *pg:e2*$\rangle$ is not relevant, because q_p does not consider the provenance entity *pg:e2*.

Under the assumption that the cost of accessing a quadruple from main memory is insignificant compared to the cost of accessing it from disk, we define the benefit of a cached fragment ϕ as $ben(\phi) = \sum_{q_a \in \mathcal{Q}} |\phi_{q_a} \cap \phi|$ where $\mathcal{Q}$ is the space of all possible queries, and ϕ_{q_a} is the set of all quadruples required by the query engine to answer q_a. In other words, the benefit of a cached fragment is the *absolute* number of times one of its quadruples can be fetched to answer a query in the query space. It follows that the total benefit of a selection of cached fragments $\Phi' \subseteq \Phi$ from a tree Φ, is given by $ben(\Phi') = \sum_{q_a \in \mathcal{Q}} |\phi_{q_a} \cap u(\Phi')|$, with $u(\Phi') = \bigcup_{\phi \in \Phi'} \phi$. Our goal is to find a Φ' with maximal $ben(\Phi')$.

We highlight that in real-world query engines, the benefit of a fragment w.r.t a query q_a may not necessarily depend on the *absolute* number of cached relevant quadruples used to answer q_a but on the ratio w.r.t the query's relevant set, i.e., $\frac{|\phi_{q_a} \cap u(\Phi')|}{|\phi_{q_a}|}$. For example, it may be more beneficial to retrieve 1000 cached quadruples for a query with $|\phi_{q_a}| = 1000$ than for a query with $|\phi_{q_a}| = 10000$. In the absence of an explicit query load, however, we can only expect to *estimate* the term $|\phi_{q_a} \cap u(\Phi')|$ since the queries q_a and their corresponding relevant sets ϕ_{q_a} are unknown. In the following we show that the fixed structure of RDF cubes as well as our focus on OLAP queries, both provide hints to guarantee that $ben(\Phi')$ is at least large.

Observation 1 *Distance to observations. The closer to the observations a predicate lies in the schema, the larger the relevance set of its matching fragments is.*

For example, **all OLAP queries** on data cubes involve aggregation of at least one measure. This means that $|\phi_{q_a} \cap \phi| > 0$ for fragments ϕ that match measure quadruples. Furthermore and assuming connected SPARQL queries, filtering or grouping on attributes and dimensions in higher levels always requires to *pass by* the lower levels, hence it is more beneficial to cache quadruples with predicates in the lower levels.

Observation 2 *Diversity. Fragments with larger diversity of predicates have larger relevance sets, i.e., they "touch" more queries.*

Observation 3 *Duplicates. Given a selection of fragments Φ, duplicate quadruples lying in different fragments do not provide additional benefit, because they occupy extra memory without extending the set of relevant queries of Φ.*

Based on these observations and the fragmentation defined by the provenance-aware fragment tree, we devise a selection strategy given a memory budget.

3.3 Fragment selection

Given a provenance-augmented RDF cube $\mathcal{K}_c = \{\mathcal{O}, \mathcal{D}, \mathcal{P}_{\mathcal{M}}, \mathcal{P}_{\mathcal{D}}, \mathcal{P}_{\mathcal{A}}, \Delta\}$, a maximum budget W in number of quadruples, and a provenance-aware fragment tree Φ, we formulate the *budgeted provenance-enabled fragment selection problem* as an integer linear program (ILP):

$$
\begin{aligned}
\text{maximize} \quad & \sum_{\phi \in \Phi} d_o(\phi)^{-1} \times d_v(\phi) \times x_\phi \\
\text{s.t.} \quad & \sum_{\phi \in \Phi} |\phi| \times x_\phi \leq W \quad \text{(budget)} \\
& \forall p : \phi_{root} \rightarrow \cdots \rightarrow \phi_k \; : \sum_{\phi \in p} x_\phi \leq 1 \quad \text{(no replication)} \\
& \forall \phi \in \Phi : x_\phi \in \{0, 1\} \quad \text{(integrality constraints)}
\end{aligned} \tag{1}
$$

Each fragment ϕ in the lattice is assigned a Boolean variable x_ϕ. If $x_\phi = 1$ the fragment is chosen for caching. Hence, the solution to the ILP produces a set of fragments $\Phi_{cached} \subset \Phi$ that will be stored in main memory. Observations 1 and 2 are implemented in the objective function. This function decreases with the distance of the fragment's predicates to the observations (d_o) and increases with the fragment's diversity (d_v). We define the distance of a predicate to the observations as the number of hops from an observation to the predicate in the schema. In Figure 1, for example, the predicates *air:pm10* and *air:unit* have distances 1 and 2 respectively. If a fragment ϕ contains quadruples with different predicates, $d(\phi)$ is the smallest distance among all predicates in ϕ. The diversity, on the other hand, is the number of different predicates in quadruples in ϕ. The cost model is encoded in the *budget* constraint. Since duplicate quadruples do not contribute with additional benefit (Observation 3), the *no replication constraint* guarantees that the resulting set Φ_{cached} has no redundancy. Because of this constraint and the partial order encoded in the tree, the solver can pick at most one fragment in a given path from the root to a leaf.

3.4 Query rewriting

In this section, we describe how to use a selection of cached fragments Φ_{cached} to answer provenance-aware queries $q = \langle q_p, q_a \rangle$. Recall from Section 2.4 that in a setting based on named graphs, a provenance-aware query can be answered by adding a FROM clause to the analytical query q_a for each provenance identifier $i \in I$ reported by q_p. Each provenance identifier corresponds to a named graph that resides in disk. In our setting we count additionally on a set of cached fragments Φ_{cached} that can be accessed from memory. We treat each fragment $\phi \in \Phi_{cached}$ as a memory named graph with label $id(\phi)$, where $id(\phi)$ returns the concatenation of the constant components of the ϕ's signature s_ϕ. Exploiting those memory named graphs to answer the analytical query is the goal of Algorithm 2. The algorithm takes as input an analytical query q_a, a provenance-aware tree Φ_{tree}, the result of the provenance query I and the set of cached fragments Φ_{cached} reported by our selection strategy in Section 3.3. The algorithm returns the graph labels that will be added as FROM clauses to the analytical query.

Line 1 initializes some intermediate variables. Lines 2-5 compute the most specific fragments in the lattice that are relevant to the analytical query, i.e., fragments whose signatures combine the provenance entities in I with the triple patterns of the analytical query. Then for each relevant fragment ϕ, the algorithm verifies whether ϕ is in the cache (line 9). If so, the fragment is added as a candidate (line 8). Otherwise, the algorithm verifies whether one of ϕ's ancestors (line 9) has been cached. There can be at most of one of such ancestors due to the redundancy constraint discussed in Section 3.3.

Algorithm 2: rewriteAnalyticalQuery

Input: Analytical query q_a, provenance-aware tree Φ_{tree}, the provenance query result I, the set of cached fragments Φ_{cached}

Output: A set of graph labels

```
1  candidates := relevant := disk := ∅
2  foreach t = ⟨s, p, o⟩ ∈ q_a do
3      foreach i ∈ I do
4          s := {⟨∗, p, o, i⟩, ⟨∗, p, ∗, i⟩} ∩ Φ_tree
5          relevant := relevant ∪ {most-specific-fragment-in(s)}
6  foreach φ ∈ relevant do
7      if φ ∈ Φ_cached then
8          candidates := candidates ∪ {φ}
9      else if ∃ φ′ ∈ Φ_cached : s_φ′ ⊒_f s_φ then
10         candidates := candidates − {φ̂ : s_φ′ ⊒_f s_φ̂} ∪ {φ′}
11     else
12         disk := disk ∪ {i : s_φ = ⟨∗, p, ∗, i⟩}
13         candidates := candidates − {φ : s_φ ≈ ⟨−, −, −, i⟩}
14 return {id(φ) : φ ∈ candidates ∪ disk)}
```

If an ancestor ϕ' is found in the cache, the algorithm adds it to the set of candidates (line 10). Since this addition turns every (possibly) selected descendant of ϕ' redundant, the algorithm removes them from the list of candidates (line 10). If neither ϕ nor any of its ancestors is in the cache, the algorithm takes as candidate the named graph labeled with the provenance identifier i in s_ϕ (line 12). This step turns any fragment with i in its signature superflous, and thus unnecessary (line 13). Once the final list of candidates have been computed, Alg. 2 generates the graph labels that will be used to rewrite the analytical query (line 14).

4 Experiments

4.1 Experimental setup

Data. We evaluated PAC on several datasets generated with the Star Schema Benchmark (*SSB* [23]) and on the QBOAirbase dataset [12]. The SSB benchmark provides a data generator for a database of line orders processed by a wholesaler. The number of line orders is an argument for the data generator. We converted the SSB dataset into an RDF cube, where each line order corresponds to an observation defined by four dimensions: supplier, part, customer, and date. We generated four datasets with four different numbers of line orders: 80k, 160k, 320k, and 640k. This resulted in 2.3m, 4.4m, 7.8m, and 14.4m triples respectively. All SSB datasets contain 68 distinct predicates. The QBOAirbase dataset, on the other hand, models air pollution measurements from 36 European countries as an RDF cube augmented with workflow provenance. A measurement corresponds to an observation with coordinates in the time, station (location), and sensor dimensions. We tested our approach on the subset of measurements of Denmark (qboairbase-dk) and Great Britain (qboairbase-gb). These datasets account for 542k and 4.3m triples respectively, both with 81 distinct predicates.

Provenance Data and Queries. Since the SSB benchmark does not provide provenance for the data, we augmented each RDF cube with 1000 distinct provenance entities and simulated a set of provenance queries. The provenance entities are assigned to observations in the cube according to two settings: balanced and unbalanced. In the *balanced* setting, each provenance entity is assigned the same number of observations in the cube, whereas in the *unbalanced* setting the ith provenance entity is assigned 2^i triples. We denote the resulting SSB datasets with the prefixes *b-* and *u-* followed by the number of line orders, e.g., *b-ssb-80k* contains 80k line orders with a balanced provenance assignment. We simulated our provenance queries by materializing sets of provenance entities covering from 10% to 90% of the provenance entities in the cube (at intervals of 10%). The datasets *qboairbase-dk* and *qboairbase-gb* contain 25.3k and 191.8k different provenance identifiers. For QBOAirbase we constructed a set of 5 provenance queries. These queries impose constraints (a) on whether the data has been quality checked or not (2 queries), (b) on whether we know the data provider or not (2 queries), and (c) on the observation's generation time.

Analytical Queries. The SSB benchmark provides a set of 13 standard OLAP queries [23]. For QBOAirbase [12], we used 8 of the analytical queries available at the project's website[1]. These are the queries where Jena does not time out. For all datasets, we construct provenance-aware queries by combining each analytical query with each of our provenance queries. Each provenance-aware query is executed three non-consecutive times in random order. We averaged the runtimes.

System Setup and Opponent. We used the Jena TDB physical database for the in-disk named graphs, and the Jena TDB in-memory store for the cached fragments[2]. All experiments were run in a virtual server with an AMD Opteron 6376 with 8 cores, 128 GB of RAM and 1 TB of disk space running in RAID-5. We tested our queries under two general system settings: (1) after purging the operating system cache and disabling the Jena TDB cache —which we call *cold*—, and (2) with the default TDB cache ($\sim$ 50MB) and a populated OS's cache after having run all the queries at least once. We call this setting *warm*. We compare our approach with the caching provided by Jena TDB and with the LRU caching strategy. The memory budgets are provided as percentages. For Jena TDB a budget of 20% means the engine counts on memory of size 20% the physical database. In contrast, for PAC and LRU the budgets indicate the percentage of triples in the dataset that will be cached. LRU populates the available cache space with the fragments used by the last executed query in a driven-by-size greedy fashion. Jena's standard execution plans timed out with most of the queries, thus we implemented an execution strategy on top of Jena [1,32], on which queries are executed on the merge of all relevant in-disk named graphs and cached fragments.

4.2 Evaluation

Impact of graph filtering. We disabled caching and compare PAC's graph filtering and query rewriting with the approach proposed in [1], and a naive query rewriting on the analytical queries. The naive approach rewrites the analytical query by adding a FROM clause for each of the results of the provenance query. In contrast, the approach in [1]

[1] `http://qweb.cs.aau.dk/qboairbase/`
[2] We used Jena v.3.2 available at `https://jena.apache.org/`

Dataset	PAC			Context index			Naive
	Runtime	Build time	Triples reduction	Runtime	Build time	Triples reduction	Runtime
b-ssb-80k	24.52s	17.38s	24.11%	35.97s	24.45s	24.10%	33.36s
u-ssb-80k	25.82s	17.65s	22.58%	37.48s	27.57s	22.56%	34.79s
qboairbase-gb	20.04s	42.72s	12.00%	103.41s	134.10s	-37.51%	24.85s
qboairbase-dk	1.98s	5.56s	13.72%	6.54s	19.94s	-35.31%	2.61s

Table 1: Performance of different graph filtering strategies (warm setting).

defines a *context index* that maps provenance entities to predicate paths, allowing for pruning of the graphs that do not co-occur with predicate paths in the query. In the same spirit, Alg. 2 filters irrelevant graphs by means of the provenance-aware fragment tree, which encodes the co-occurrences of predicates, object values, and provenance identifiers. Table 1 shows the average runtime and average index built time of the different strategies for four of our datasets. We observe that PAC's filtering outperforms the naive approach in query runtime, because it achieves reductions from 12% to 24% in the total number of materialized triples. While the context index and PAC achieve comparable reductions in the SSB datasets, [1] performs worse for two reasons: (a) it merges all relevant graphs in disk, and (b) it does not handle unions natively. The latter limitation implies that subqueries must be executed independently and their results merged. It also explains why this method sometimes materializes more triples than the naive approach, leading to negative reduction rates. Finally, we highlight that the context index's build time is up to 3x slower than PAC's provenance-aware fragment tree because the context index runs an expensive select query for each predicate path in the index.

Caching vs. In-memory DB. Table 2 compares the average runtime of PAC, the LRU, and the Jena TDB caching strategies –the two latter with and without PAC's filtering– at budget 20% against full PAC (budget 100%) and the Jena TDB in-memory database in a warm setting. PAC at budget 20% outperforms in total time all caching strategies and the Jena in-memory database. The bottom line is that with PAC's strategic caching, it is not necessary to store everything in main memory for speed-up. In addition, full PAC is 2x faster than the in-memory database thanks to PAC's graph filtering (Alg. 2).

Dataset	Full-PAC	Jena-mem	PAC	LRU+ PAC+F	TDB+ PAC+F	LRU	TDB
b-ssb-80k	**13.03s**	34.05s	23.43s	**20.98s**	23.78s	35.30s	33.21s
u-ssb-80k	**13.74s**	35.80s	26.58s	38.15s	**26.34s**	38.15s	35.84s
airbase-gb	**17.86s**	25.04s	**13.80s**	20.01s	17.45s	22.98s	25.56s
airbase-dk	**2.06s**	2.75s	1.65s	2.88s	**0.02s**	3.63s	2.56s
Total	**42.69s**	97.64s	**65.46s**	82.02s	67.59s	100.06s	117.17s

Table 2: Runtime of a full in-memory database vs. the caching strategies at budget=20%

Impact of the memory budget. Figures 3 and 4 show the impact of the memory budget on the average cache hit-rate and the average response time of PAC in a cold setting on

four datasets from all our families of datasets. We define the hit-rate as the ratio of graph labels returned by Alg. 2 that correspond to cached fragments. We observe a monotonically increasing behavior in the hit-rate for all datasets. On the u-ssb-80k and qboairbase datasets, the hit-rate already approaches 80% at budget 10%, contrary to the h-ssb-80k dataset where the increase is more gradual. This phenomenon is mainly caused by the fine granularity of the fragments both in u-ssb-80k and qboairbase. Fine-grained fragments give the selector more flexibility at utilizing the available budget in contrast to very large fragments as the ones found in h-ssb-80k. If a very large fragment does not fit into the remaining cache space, it will not be added, even though it may be relevant to many queries in the query space. The trends in the hit-rate are supported by the runtime behavior in Figure 4.

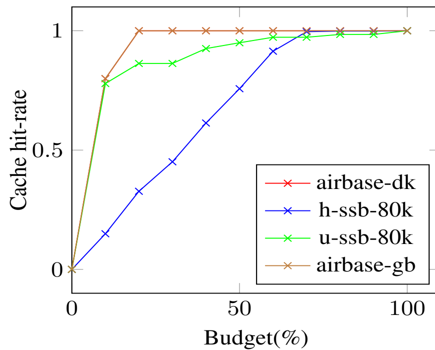

Fig. 3: Budget vs. hit-rate for PAC

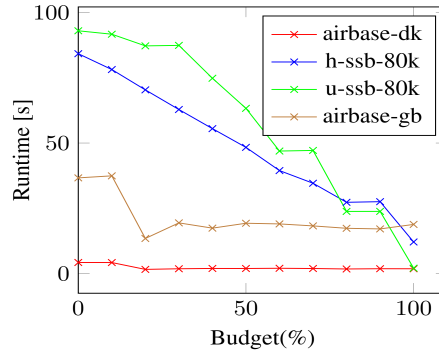

Fig. 4: Budget vs. runtime for PAC

PAC vs. LRU and TDB. We compare PAC, the Jena TDB native caching, and the LRU caching strategy for qboairbase-gb on a warm setting in Figure 5. The trend is independent of the system setting and is similar for qboairbase-dk. We first observe that PAC outperforms Jena TDB at all budgets. Only when PAC's filtering is enabled (TDB+PAC-F), TDB performs comparably to PAC. On the contrary, LRU seems inadequate for this dataset, even when PAC's filtering is enabled (LRU+PAC-F). Due to the high diversity of cached fragment signatures in the qboairbase datasets (approx. 192k), it is unlikely for two consecutive queries to require the same fragments. This hurts the performance of LRU, which delivers a hit-rate of 0 for less than 40% budget. LRU+PAC-F does slightly better, but its maximal hit-rate is no higher than 0.6. The situation is different for the u-ssb-80k dataset as shown in Figure 6. While PAC still delivers the best performance, TDB is outperformed by LRU. The trends are corroborated by the hit-rate, where PAC is between 0.26 and 0.57 ratio points better than LRU, and between 0.24 and 0.69 points better than LRU+PAC-F. Our findings in the h-ssb-80k dataset are alike: PAC is between 0.15 and 0.47 ratio points better than LRU as displayed in Figure 7 (between 0.08 and 0.28 points w.r.t. LRU+PAC-F). All in all, the synergy between graph filtering and a high hit-rate makes PAC faster than standard caching strategies.

Caching on bigger datasets. We also investigate the behavior of the different caching strategies as the number of triples increases in the u-ssb family of datasets on a warm

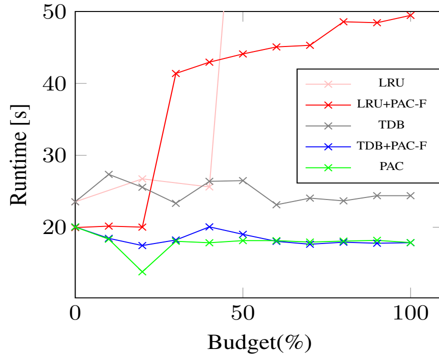

Fig. 5: Runtime on qboairbase-gb

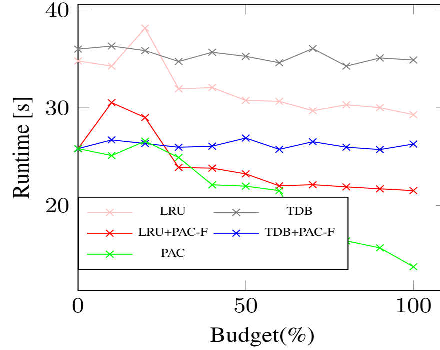

Fig. 6: Runtime on u-ssb-80k

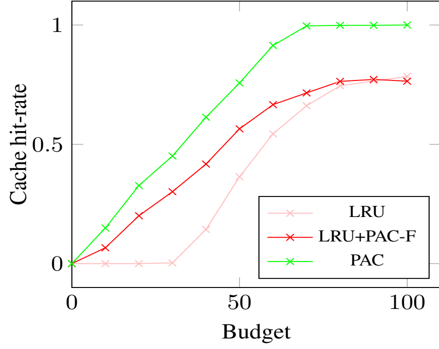

Fig. 7: Hit-rate on h-ssb-80k

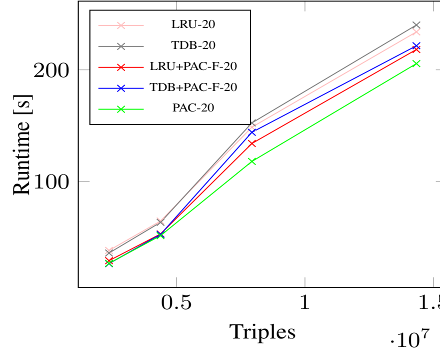

Fig. 8: Runtime on u-ssb

system setting. We set a budget of 20% and show the results in Figure 8. PAC consistently achieves better runtime than its competitors. In general, all trends observed in the h-ssb-80k and u-ssb-80k datasets remain constant as the number of triples increases.

Impact of caching on queries. We also study the impact of the different caching strategies on the response time of the individual analytical queries. For this purpose we compute the area under the curve of response time vs. budget for each analytical query under the different strategies on the h-ssb-80k, u-ssb-80k, qboairbase-gb, and qboairbase-dk datasets. The runtimes were averaged across all provenance queries. Table 3 shows the number of queries where each strategy wins, that is, the strategy achieves the smallest area under the curve until budgets 20%, 50%, and 100%. We notice that TDB becomes insensitive to the budget argument after a value of 20%. By looking at the winning strategies in each query, we observe that PAC has an almost stable behavior: the set of benefited queries grows monotonically as the budget increases. Despite of being query-load oblivious, PAC with budget 20% wins in 50% of the analytical queries in the SSB datasets, and in 100% of the analytical queries in the QBOAirbase datasets.

5 State of the Art

This paper studies the impact of caching fragments of an RDF dataset for query processing on provenance-augmented RDF cubes. Therefore, we present the state of the art

	budget 20%					budget 50%					budget 100%				
Dataset	PAC	TDB+ PAC+F	LRU+ PAC+F	TDB	LRU	PAC	TDB+ PAC+F	LRU+ PAC+F	TDB	LRU	PAC	TDB+ PAC+F	LRU+ PAC+F	TDB	LRU
b-ssb-80k	4	**5**	4	0	0	**6**	1	**6**	0	0	**7**	0	6	0	0
u-ssb-80k	4	**5**	1	1	2	**8**	2	1	1	1	**7**	2	3	1	0
qboairbase-gb	**9**	0	0	0	0	**9**	0	0	0	0	**9**	0	0	0	0
qboairbase-dk	**9**	0	0	0	0	**9**	0	0	0	0	**9**	0	0	0	0

Table 3: Number of queries where each strategy wins (warm system setting)

in terms of three axes: caching in SPARQL and OLAP, query answering on SPARQL aggregation queries, and provenance management.

Caching in SPARQL and OLAP. Caching data to speed up query answering is a standard technique in databases and has been also applied in the context of RDF/SPARQL and OLAP. Caching can be implemented at different levels. For example, the Jena TDB engine relies on the file caching provided by the Java Virtual Machine to speed up subsequent access to recently-used parts of the RDF store. When implemented at the application level, e.g., in a client-server setting, caching is often concerned with the reutilization of query results [18, 19, 30]. In contrast, we aim at caching fragments of the RDF dataset that are used by multiple queries and unlike [18], we do not count on an explicit query load. Caching has also been implemented for data fragments and intermediate query results. In the framework of Linked Data Fragments (LDF) [29], the server can return cached data fragments, leaving the query processing to the client. While PAC's notion of fragments is similar to that of LDF, [29] does not consider provenance and focuses more on reducing the server's load for the sake of availability rather than on minimizing response time. Caching has also been applied in the context of OLAP [10, 16]. The system PeerOLAP [16], for example, relies on a P2P network to answer OLAP queries. PeerOLAP reuses the results of queries executed by neighbour peers as data sources. As PeerOLAP, most systems focus on caching recently queried results [2, 11]. In [27] a hybrid query engine is proposed; it combines live results with cached data as a trade-off between precision and speed. In our approach, we use heuristics to pre-cache strategically important parts of the data.

SPARQL Aggregation Queries. The interest on optimizing SPARQL queries with aggregation [5, 15] started with the publication of SPARQL 1.1 [25]. MARVEL [15] proposes to answer SPARQL aggregation queries on RDF cubes by rewriting the query in terms of a set of views. These views are structured according to a partial order, and selected for query answering based on a cost model as in PAC. Unlike PAC, MARVEL is not a caching approach, it cannot handle provenance, and it is based on data aggregates materialized as views rather than on actual RDF fragments. Conversely, the approach in [1] handles provenance and proposes an index for graph filtering. The index stores the co-occurence of provenance identifiers and predefined predicate paths. The approach also assumes that some of the quadruples produced by the ETL process are all assigned a single hard-coded provenance identifier. Albeit not equivalent, our provenance-aware fragment tree serves for graph filtering at a better performance without additional assumptions. Besides [1] is not a caching approach.

Provenance management. The management of provenance is a crucial task for Linked Data and RDF given the decentralized nature of the Web. There are several approaches to encode provenance in RDF, such as reification [24], named graphs [4], singleton properties [22], and embedded triples [14]. In this work we focus on workflow provenance [26]. Other approaches study provenance in terms of the lineage of the query results [13, 31]: expressions (e.g., a polynomial) that encode the origin of a result w.r.t the triples in the dataset. The TripleProv engine [31] allows for native calculation of lineage for the results of SPARQL queries. Our setting is significantly different, because provenance is encoded as provenance entities described using RDF and the PROV-O ontology [20]. Compared to the notion of lineage for query results, a provenance entity can be seen as the identifier of a precomputed lineage.

6 Conclusions

In this paper, we have presented provenance-aware caching (PAC), an approach to cache fragments of a provenance-augmented RDF graph in order to speed up provenance-aware OLAP queries. We propose a fragmentation scheme for provenance-augmented RDF data, and an approximative benefit model tailored for RDF cubes and OLAP queries under memory constraints. Our techniques are query-load agnostic and our experimental evaluation shows that PAC outperforms the Jena TDB native cache and the standard LRU caching strategy in real and synthetic data. The PAC principle can be applied in scenarios where the query-load is unknown, e.g., to bootstrap the cache, or when the workload changes constantly. It is also applicable in settings characterized by locations of "fast" and "slow" access, such as a hybrid drives or remote storage servers. We have also shown how to efficiently answer provenance-aware queries in an engine based on named graphs. As future work, we envision to integrate explicit dynamic query workloads into our framework, and to extend the fragment definitions beyond equality constraints on the quadruples by, for example, using the provenance graph. All the data and experimental results are available at `http://qweb.cs.aau.dk/pac/`.

Acknowledgments

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-4093-00301.

References

1. Kim Ahlstrøm, Katja Hose, and Torben Bach Pedersen. Towards Answering Provenance-Enabled SPARQL Queries Over RDF Data Cubes. In *JIST*, 2016.
2. Barry Bishop, Atanas Kiryakov, Damyan Ognyanov, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. FactForge: A fast track to the Web of data. In *SWJ*, 2011.
3. Tyrone Cadenhead, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham. A Language for Provenance Access Control. In *CODASPY*, 2011.
4. Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named Graphs, Provenance and Trust. In *WWW*, 2005.
5. Roger Castillo. RDFMatView: Indexing RDF Data for SPARQL Queries. In *SSWS*, 2010.
6. Artem Chebotko, John Abraham, Pearl Brazier, Anthony Piazza, Andrey Kashlev, and Shiyong Lu. Storing, Indexing and Querying Large Provenance Data Sets as RDF Graphs in Apache HBase. In *SERVICES*, 2013.
7. James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. In *Foundations and Trends in Databases*, 2009.

8. Dario Colazzo, Tushar I. Ghosh, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. WaRG: Warehousing RDF Graphs. In *Bases de Données Avancées*, 2013.
9. Richard Cyganiak and Dave Reynolds. The RDF Data Cube Vocabulary. W3C recommendation, 2014. http://www.w3.org/TR/2014/REC-vocab-data-cube-20140116/.
10. Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching Multidimensional Queries Using Chunks. *SIGMOD Rec.*, 27(2):259–270, 1998.
11. Orri Erling and Ivan Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge - Networked Media*, 2009.
12. Luis Galárraga, Kim Ahlstrøm Meyn Mathiassen, and Katja Hose. QBOAirbase: The European Air Quality Database as an RDF Cube. In *ISWC, Posters & Demonstrations*, 2017.
13. Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance Semirings. In *PODS*, 2007.
14. Olaf Hartig. Foundations of RDF* and SPARQL* - An Alternative Approach to Statement-Level Metadata in RDF. In *AMW*, 2017.
15. Dilshod Ibragimov, Katja Hose, Torben Bach Pedersen, and Esteban Zimányi. Optimizing Aggregate SPARQL Queries Using Materialized RDF Views. In *ISWC*, 2016.
16. Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An Adaptive Peer-to-peer Network for Distributed Caching of OLAP Results. In *SIGMOD*, 2002.
17. Benedikt Kämpgen, Sean O'Riain, and Andreas Harth. Interacting with Statistical Linked Data Via OLAP Operations. In *ILD*, 2015.
18. Johannes Lorey and Felix Naumann. Caching and Prefetching Strategies for SPARQL Queries. In *ESWC (Satellite Events)*, 2013.
19. Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *ESWC*, 2010.
20. Deborah McGuinness, Timothy Lebo, and Satya Sahoo. PROV-O: The PROV Ontology. W3C recommendation, 2013. http://www.w3.org/TR/2013/REC-prov-o-20130430/.
21. Pablo N. Mendes, Hannes Mühleisen, and Christian Bizer. Sieve: Linked Data Quality Assessment and Fusion. In *EDBT-ICDT*, pages 116–123, 2012.
22. Vinh Nguyen, Olivier Bodenreider, and Amit Sheth. Don't Like RDF Reification?: Making Statements About Statements Using Singleton Property. In *WWW*, 2014.
23. Pat O'Neil, Betty O'Neil, and Xuedong Chen. Star Schema Benchmark. Technical report, UMass/Boston, 2009. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.
24. Yves Raimond and Guus Schreiber. RDF 1.1 primer. W3C recommendation, 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.
25. Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.
26. Yannis Theoharis, Irini Fundulaki, Grigoris Karvounarakis, and Vassilis Christophides. On Provenance of Queries on Semantic Web Data. In *IEEE Internet Computing*, 2011.
27. Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira. Hybrid SPARQL Queries: Fresh vs. Fast Results. In *ISWC*, 2012.
28. Jovan Varga, Alejandro A. Vaisman, Oscar Romero, Lorena Etcheverry, Torben Bach Pedersen, and Christian Thomsen. Dimensional Enrichment of Statistical Linked Open Data. In *Journal of Web Semantics*, 2016.
29. Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple Pattern Fragments: A Low-cost Knowledge Graph Interface for the Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 37-38:184–206, 2016.
30. Gregory Todd Williams and Jesse Weaver. Enabling Fine-Grained HTTP Caching of SPARQL Query Results. In *ISWC*, 2011.
31. Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. TripleProv: Efficient Processing of Lineage Queries in a Native RDF Store. In *WWW*, 2014.
32. Marcin Wylot, Philippe Cudre-Mauroux, and Paul Groth. Executing Provenance-Enabled Queries over Web Data. In *WWW*, 2015.