Optimizing Aggregate SPARQL Queries Using Materialized RDF Views

Dilshod Ibragimov[1,2(✉)], Katja Hose[2], Torben Bach Pedersen[2], and Esteban Zimányi[1]

[1] Université Libre de Bruxelles, Brussels, Belgium
{dibragim,ezimanyi}@ulb.ac.be
[2] Aalborg University, Aalborg, Denmark
{diib,khose,tbp}@cs.aau.dk

Abstract. During recent years, more and more data has been published as native RDF datasets. In this setup, both the size of the datasets and the need to process aggregate queries represent challenges for standard SPARQL query processing techniques. To overcome these limitations, materialized views can be created and used as a source of precomputed partial results during query processing. However, materialized view techniques as proposed for relational databases do not support RDF specifics, such as incompleteness and the need to support implicit (derived) information. To overcome these challenges, this paper proposes MARVEL (MAterialized Rdf Views with Entailment and incompLetness). The approach consists of a view selection algorithm based on an associated RDF-specific cost model, a view definition syntax, and an algorithm for rewriting SPARQL queries using materialized RDF views. The experimental evaluation shows that MARVEL can improve query response time by more than an order of magnitude while effectively handling RDF specifics.

1 Introduction

The growing popularity of the Semantic Web encourages data providers to publish RDF data as Linked Open Data, freely accessible, and queryable via SPARQL endpoints [25]. Some of these datasets consist of billions of triples. In a business use case, the data provided by these sources can be applied in the context of On-Line Analytical Processing (OLAP) on RDF data [5] or provide valuable insight when combined with internal (production) data and help facilitate well-informed decisions by non-expert users [1].

In this context, new requirements and challenges for RDF analytics emerge. Traditionally, OLAP on RDF data was done by extracting multidimensional data from the Semantic Web and inserting it into relational data warehouses [19]. This approach, however, is not applicable to autonomous and highly volatile data on the Web, since changes in the sources may lead to changes in the structure of the data warehouse (new tables or columns might have to be created) and will impact the entire Extract-Transform-Load process that needs to reflect the changes.

P. Groth et al. (Eds.): ISWC 2016, Part I, LNCS 9981, pp. 341–359, 2016.
DOI: 10.1007/978-3-319-46523-4_21

In comparison to relational systems, native RDF systems are better at handling the graph-structured RDF model and other RDF specifics. For example, RDF systems support triples with *blank* nodes (triples with unknown components) whereas relational systems require all attributes to either have some value or *null*. Additionally, RDF systems support entailment, i.e., new information can be derived from the data using RDF semantics while standard relational databases are limited to explicit data.

Processing analytical queries in the context of Linked Data and federations of SPARQL endpoints has been studied in [15,16]. However, performing aggregate queries on large graphs in SPARQL endpoints is costly, especially if RDF specifics need to be taken into account. Thus, triple stores need to employ special techniques to speed up aggregate query execution. One of these techniques is to use materialized views – named queries whose results are physically stored in the system. These aggregated query results can then be used for answering subsequent analytical queries. Materialized views are typically much smaller in size than the original data and can be processed faster.

In this paper, we consider the problem of using materialized views in the form of RDF graphs to speed up analytical SPARQL queries. Our approach (MARVEL) focuses on the issues of selecting RDF views for materialization and rewriting SPARQL aggregate queries using these views. In particular, the contributions of this paper are:

- A cost model and an algorithm for selecting an appropriate set of views to materialize in consideration of RDF specifics
- A SPARQL syntax for defining aggregate views
- An algorithm for rewriting SPARQL queries using materialized RDF views

Our experimental evaluation shows that our techniques lead to gains in performance of up to an order of magnitude.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the used RDF and SPARQL notation and describes the representation of multidimensional data in RDF. Section 4 specifies the cost model for view selection, and Sect. 5 describes query rewriting. We then evaluate MARVEL in Sect. 6, and Sect. 7 concludes the paper with an outlook to future work.

2 Related Work

Answering queries using views is a complex problem that has been extensively studied in the context of relational databases [13]. However, as discussed in [13,22], aggregate queries add additional complexity to the problem.

In relational systems, the literature proposes semantic approaches for rewriting queries [22] as well as syntactic transformations [11]. However, SPARQL query rewriting is more complex. The results for views defined as *SELECT* queries represent solutions in tabular form, so that the solutions need to be converted afterwards into triples for further storage, thus making a view definition in SPARQL more complex and precluding view expansion (replacing the view by its definition).

Another problem in this context is to decide which views to materialize in order to minimize the average response time for a query. [14] addresses this problem in relational systems by proposing a cost model leading to a trade-off between space consumption and query response time for an arbitrary set of queries. [23] provides a method to generate views for a given set of select-project-join queries in a data warehouse by detecting and exploiting common sub-expressions in a set of queries. [2] further optimizes the view selection by automatically selecting an appropriate set of views based on the query workload and view materialization costs. However, these approaches have been developed in the context of relational systems and, therefore, do not take into account RDF specifics such as entailment, the different data organization (triples vs. tuples), the *graph*-like structure of the stored data, etc.

The literature proposes some approaches for answering SPARQL queries using views. [4] proposes a system that analyzes whether query execution can be sped up by using precomputed partial results for conjunctive queries. The system also reduces the number of joins between tables of a back-end relational database system. While [4] examines core system improvements, [18] considers SPARQL query rewriting algorithms over a number of virtual SPARQL views. The algorithm proposed in [18] also removes redundant triple patterns coming from the same view and eliminates rewritings with empty results. Unlike [18], [9] examines materialized views. Based on a cost model and a set of user defined queries, [9] proposes an algorithm to identify a set of candidate views for materialization that also account for implicit triples. However, these approaches [4,9,18] focus on conjunctive queries only. The complexity of loosing the multiplicity on grouping attributes (by grouping on attribute X, we loose the multiplicity of X in data) and aggregating other attributes is not addressed by these solutions.

The performance gain of RDF aggregate views has been empirically evaluated in [17], where views are constructed manually and fully match the predefined set of queries. Hence, the paper empirically evaluates the performance gain of RDF views but does not propose any algorithm for query rewriting and view selection.

Algorithms that use the materialized result of an RDF analytical query to compute the answer to a subsequent query are proposed in [3]. The answer is computed based on the intermediate results of the original analytical query. However, the approach does not propose any algorithm for view selection. It is applicable for the subsequent queries and not to an arbitrary set of queries.

Although several approaches consider answering queries over RDF views [4,9,18], none of them considers analytical queries and aggregation. In this paper, we address this problem in consideration of RDF specifics such as entailment and data organization in the form of triples, and taking into account the graph structure of the stored data. In particular, this paper proposes techniques for cost-based selection of materialized views for aggregate queries, query rewriting techniques, and a syntax for defining such views.

3 RDF Graphs and Aggregate Queries

The notation we use in this paper follows [21,25] and is based on three disjoint sets: blank nodes B, literals L, and IRIs I (together BLI). An RDF triple $(s, p, o) \in BI \times I \times BLI$ connects a subject s through property p to object o. An RDF dataset (G) consists of a finite set of triples and is often represented as a graph. Queries are based on graph patterns that are matched against G. A Basic Graph Pattern (BGP) consists of a set of triple patterns of the form $(IV) \times (IV) \times (LIV)$, where V ($V \cap BLI = \emptyset$) is a set of query variables. Variable names begin with a question mark symbol, e.g., $?x$. In graph notation, a BGP can be represented as a directed labeled multi-graph whose nodes N correspond to subjects and objects in the triple patterns. The set of edges E contains one edge for each triple pattern and the property as its label. In data analytics, graph patterns have a special, *rooted* pattern [5]. A BGP is rooted in node $n \in N$ iff any node $x \in N$ is reachable from n following directed edges in the graph.

The most common SPARQL [25] aggregate queries conform to the form *SELECT* RD *WHERE* GP *GROUP BY* GRP, where RD is the result description based on a subset of variables in the graph pattern GP. GP defines a BGP and optionally uses functions, such as *assignment* (e.g., *BIND*) and constraints (e.g., *FILTER*). GRP defines a set of grouping variables, whereas RD contains selection description variables as well as aggregation variables with corresponding aggregate functions. In this paper, we consider the standard aggregate functions *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*.

Data that SPARQL analytical queries are typically evaluated on can be represented in an n-dimensional space, called a *data cube*. The data cube is defined by *dimensions* (perspectives used to analyze the data) and *observations* (facts). Dimensions are structured in *hierarchies* to allow analyses at different aggregation levels. Hierarchy level instances are called *members*. Observations (cells of a data cube) have associated values called *measures*, which can be aggregated.

An example of data with hierarchical dimensions is the utilities consumption data from electricity and gas meters for Scottish Government buildings[1] enhanced in the Date dimension. The data is available as energy usage over a daily period. Figure 1 sketches the data with two hierarchical dimensions: Building, Locality, and Region are hierarchy levels in the Geography dimension, Report Date, Month, and Year are hierarchy levels in the Date dimension, and Data represents an observation with the utility consumption measure. A dataset with such observations is stored in a SPARQL endpoint to enable analytical querying with grouping on different hierarchy levels. For example, the following query computes the daily consumption of electricity in each city in September 2015:

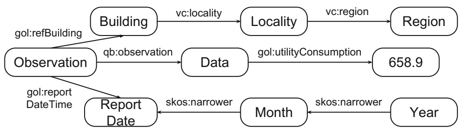


Fig. 1. Representing observations in RDF

[1] http://cofog01.data.scotland.gov.uk/id/dataset/golspie/utilities.

```
SELECT ?dt ?plc (SUM(?v) as ?value) WHERE {
  ?slc gol:refBuilding ?bld ; gol:reportDateTime ?tm ; qb:observation ?ob .
  ?ob  gol:utilityConsumption ?v . ?bld org:siteAddress/vc:adr/vc:locality ?plc .
  ?mn skos:narrower ?tm . ?mn gol:value ?mVal . FILTER (?mVal = 'September 2015')
} GROUP BY ?tm ?plc
```

Listing 1.1. Example query with grouping and aggregation

Based on the multidimensional data model, we can define traditional OLAP operations over the data such as *roll-up*, *drill-down*, *slicing*, and *dicing*. Intuitively, the slice operator fixes a single value for a level of a dimension to define a subcube with one dimension less. Dice uses a Boolean condition and returns a new cube containing only the cells satisfying the condition. Roll-up aggregates measure values at a coarser granularity for a given dimension while drill-down disaggregates previously summarized data to a child level in order to obtain measure values at a finer granularity. In SPARQL, slice and dice can be achieved by adding a constraint function (like *FILTER*) to the graph pattern, while roll-up and drill-down can be achieved by removing/adding connected triple patterns to the existing graph pattern of the query.

4 View Materialization in MARVEL

A high number of triples needs to be processed for evaluating OLAP queries on a dataset. This imposes high execution costs, especially when the amount of data increases. To enable scalable processing, we propose RDF-specific techniques to select a set of materialized views that can be used to evaluate queries more efficiently. We define a materialized view as a named graph described by a query whose results are physically stored in a triple store. Given a query, the system checks whether the query can be answered based on the available materialized views. As materialized views are typically smaller than the original/raw data, this can yield a significant performance boost. Precalculating all possible aggregations over all dimension levels is usually infeasible as it requires much more space than the raw data [24]. Thus, it is important to find an appropriate set of materialized views to minimize the total query response time.

4.1 Creating Materialized RDF Views

Views used for rewriting conjunctive queries can be defined by *CONSTRUCT* queries with *WHERE* clauses [18]. Views defining queries for aggregate views are more complex since these views group and aggregate the original data. Grouping and aggregation are achieved by using *SELECT* queries. However, *SELECT* queries return data in a tabular format, not triples. Thus, the *CONSTRUCT* clause needs to define a new graph structure and triples for the obtained results. As only the combination of values for variables in the *GROUP BY* clause of a *SELECT* query is unique, we can use these values to construct the new triples.

Listing 1.2 gives an example of such a query, where the *SELECT* query aggregates utility consumptions by City and Date. We use the *IRI* and *STRAFTER* functions to create a resource identifier *id* based on the unique combination of City and Date. The *CONSTRUCT* clause then creates triples by connecting the *id* to the resulting aggregate and grouping values. The algorithm for constructing such view queries is similar to the algorithm described in [17].

```
CONSTRUCT { ?id gol:reportDate ?date ; gol:reportLocality ?vCity ;
    gol:utilityConsumption ?value . } WHERE {
  SELECT ?id ?date ?vCity (SUM(?cons) as ?value)
  WHERE { ?fact gol:refBuilding ?bld ; gol:reportDateTime ?date ;
   qb:observation ?data . ?data  gol:utilityConsumption ?cons .
   ?bld org:siteAddress/vc:adr/vc:locality ?vCity .
   BIND(IRI('http://ex.org/id#', CONCAT(STRAFTER(STR(?dt), 'http://'),
   STRAFTER(STR(?vCity), 'http://'))) AS ?id). } GROUP BY ?id ?date ?vCity }
```

Listing 1.2. Query to construct materialized view

4.2 Data Cube Lattice

To represent dependencies between views, we use the notion of a data cube lattice. The data cube lattice is, essentially, a schema of a data cube with connected nodes, where a node represents an aggregation by a given combination of dimensions. Nodes are connected if a node j can be computed from another node i and the number of grouping attributes of i corresponds to the number of attributes of j plus one. A view is defined by a query with the same grouping as in the corresponding node. For example, in case of 3 dimensions, Part (P), Customer (C) and Date (D), possible nodes (grouping combinations) are *PCD*, *PC*, *PD*, *CD*, *P*, *C*, *D* and *All* (all values are grouped into one group). In our example, the view corresponding to node *PC* can be computed from the view corresponding to node *PCD*. We denote this dependence relation as $PC \preceq PCD$ and refer to view *PCD* as the *ancestor* of view *PC*. In the presence of dimension hierarchies, the total number of different lattice nodes is $\prod_{i=1}^{k}(h_i + 1)$, where h_i represents the number of hierarchy levels in dimension i and $(h_i + 1)$ accounts for the top level *All*.

We use the data cube lattice since it formalizes which views (nodes) can be used to evaluate a particular query. Given a query grouping (*GROUP BY*), the lattice node with the exact same grouping (and its ancestors) can be used. Since these views are smaller in size than the raw data, calculating the answer from the views will be cheaper than calculating it from the raw data. Thus, to answer user queries we need to find an appropriate set of views so that the multidimensional queries posed against the data can be mapped to one of these views.

The data cube lattice has originally been proposed for selecting aggregate views in a relational framework [14]. This framework considers data that is complete and complies with a predefined schema, and therefore cannot be directly applied to RDF graphs that lack these characteristics. Additionally, RDF data may be *incomplete*. For instance, the canonicalized Ontology Infobox dataset from the DBpedia Download 3.8 contains birth place information for 266,205 persons (either as a country, a city or village, or both). However, out of 266,205 records, 16,351 records contain information only about the country of birth. Thus, the information *available* in the source may not contain the information that holds in the world (Open World Assumption) and, therefore, should *ideally* be present in the source. Accordingly, an incomplete data source is defined as a pair $\Omega = (G_a, G_i)$ of two graphs, where $G_a \subseteq G_i$. G_a corresponds to the available graph and G_i is the ideal graph [6]. Thus, a view is *incomplete* if its defining query over the available graph does not produce the same results as the defining query over the ideal graph: $[q_v]G_a \neq [q_v]_{G_i}$.

Such incomplete views may not be used to answer queries involving the grouping over a higher hierarchy level than in the view. In the above example, the aggregation over the city of birth is incomplete and the city level view, due to incompleteness, cannot be used to roll-up to the country level even though the relationship City $\rightarrow$ Country between the levels holds. Instead, the aggregation over the country level needs to be computed from the raw data taking into account the derived information that connects cities of birth to the countries.

In summary, we need to account for the graph-like structure of the stored data, presence of implicit knowledge in data, and incompleteness of views for RDF data cubes. Therefore, we propose MARVEL – a novel aggregate view selection approach that, unlike earlier approaches, supports RDF-specific requirements.

4.3 MARVEL Cost Model

MARVEL assumes that RDF data are stored as triples. Thus, the cost of answering an aggregate SPARQL query in a generic RDF store is defined as the number of triples contained in the materialized view used to answer the query. This cost model is simple and works for the general case. More complex models that account for algorithms and auxiliary structures of a particular triple store are certainly possible.

The number of triples to represent an observation in an RDF view is *(n + m)* where n is the number of dimensions and m is the number of measures. Thus, the size of a view w is equal to $Size(w) = (n + m) * N$, where N is the number of observations. This number is used to calculate the benefit of materializing the view. Note that the size of w serves as the cost of v if v is computed from w: *Cost(v) = Size(w)*. View sizes can be estimated using VoID statistics and cardinality estimation techniques [12], using a small representative subset and, in some cases, with $COUNT$ queries.

Let B_w be the benefit of view w. For every view v such that $v \preceq w$ the benefit of view w relative to v is calculated as $B_{w,v} = (Cost(v) - Size(w))$ if $Cost(v) > Size(w)$ and $B_{w,v} = 0$ otherwise. The difference between the current cost of view v and the possible cost of v (if the materialized view w is used to compute view v) contributes to the benefit of view w. We sum up the benefits for all appropriate views to receive the full benefit of view w: $B_w = \sum B_{w,v_i}$ for all i such that $v_i \preceq w$. Note that this value of benefit is absolute. If the storage space is limited, the benefit of each view per *unit space* can be considered instead. In this case, the value of the benefit is calculated by dividing the absolute benefit of the view by its size: $B_w^{'} = \frac{B_w}{Size(w)}$.

In addition, our cost model needs to account for RDF specifics, such as incomplete views and complex and indirect hierarchies. We use an annotated QB4OLAP schema [7] to describe the dataset and extend it with information about the completeness of levels, the patterns for defining hierarchy steps (which predicates are used), the types of hierarchy levels, etc. This schema reflects the source data structure and does not require adding any triples to the graph. The schema is also used to define aggregate queries for the view selection process (Sect. 4.5). We chose QB4OLAP since unlike alternatives, such as AnS [5] and QB (http://www.w3.org/TR/vocab-data-cube/), QB4OLAP allows to define multidimensional concepts such as dimensions, levels, members, roll-up relations, complex hierarchies (e.g., ragged, recursive), etc.

For example, for a birth place dimension we can specify that the roll-up to the Country level should be calculated from both the City and the Person levels since for some people we might only know the birth country, whereas for others we know the city. When a hierarchy level is computed from several ancestor levels, we say that the view corresponding to this level should be calculated from a *set* of views (to avoid double-counting in such cases, MARVEL uses the *MINUS* statement). We denote this dependence relation as $w \preceq \{v_i, \ldots, v_n\}$, where w is the current view and $\{v_i, \ldots, v_n\}$ are the ancestor views. In general, we can distinguish the following roll-up cases:

- **Single path roll-up**: a view w can be derived from any of the views $v_1, \ldots, v_n$, i.e., $\exists w, v_1, \ldots, v_n$ such that $w \preceq v_i$ and $v_i \not\preceq v_j$ for $i, j = \{1, \ldots, n\}$
- **Multiple path roll-up**: a view w can be derived from the union of views $v_1 \cup \cdots \cup v_n$ while deriving w from any single v_i will be incomplete: $\exists w, v_1, \ldots, v_n$ such that $w \preceq \{v_i, \ldots, v_n\}$, $w \not\preceq v_i$, and $v_i \not\preceq v_j$ for $i, j = \{1, \ldots, n\}$

However, before selecting the views to materialize we should take into account implicit triples since they are considered to be part of the graph.

4.4 RDF Entailment

Accounting for implicit triples in views is necessary for returning a complete answer. The W3C RDF Recommendation (http://www.w3.org/RDF/) defines a number of entailment patterns which lead to deriving implicit triples from RDF datasets. RDF Schema (RDFS) entailment patterns are particularly interesting since RDFS encodes the domain semantics.

Aggregate queries are designed to run only on available correct data; computing the sum over a set of unknown values, for instance, would not yield any useful results. Hence, in this paper we focus on deriving implicit information based on existing data and specified semantics only. Deriving information unknown due to the Open World Assumption or adding missing information using logical rules are orthogonal problems that are difficult to solve in general [8] and therefore beyond the scope of this paper.

There are two main methods for processing queries when considering RDF entailment. In the *dataset saturation* approach, all implicit triples are materialized and added to the dataset. While requiring more space and complex maintenance, this method benefits from applying plain query evaluation techniques to compute the answer. *Query reformulation*, on the other hand, leaves the dataset unchanged but reformulates a query to a union of queries and increases the overhead during query evaluation.

MARVEL uses the RDFS entailment regime during view materialization. The system reformulates queries to materialize the complete answer of a view, which allows us to leave a dataset unchanged but still account for implicit triples in query answers. Taking into account that the evaluation of the view query takes place once and the results are reused for other queries, we believe that this overhead is justified.

4.5 MARVEL View Selection Algorithm

Given the open nature of SPARQL endpoints, we assume that all groupings in user queries are equally likely. Algorithm 1 outlines the method for selecting materialized views in MARVEL; the goal is to materialize N views with the maximum benefit, regardless of their size.

```
   Input: Set of views $W$, cube schema $S$, number of needed views $N$
   Output: Selected views $W'$
1  $W' = \emptyset$ -- set of selected views ;
2  while $|W'| \leq N$ do
3      RecalculateViewCosts(W) ;
4      $\{V \times B\} = \emptyset$ -- set of views together with the benefit ;
5      foreach view $w \in W$ do
6          $\{V \times B\} = \{V \times B\} \cup$(w, CalculateBenefit(w)) ;
7      foreach $\{w_1 ... w_n\}$ for which $\exists v$ such that $v \preceq \{w_1 ... w_n\}$ (according to $S$) do
8          $\{V \times B\} = \{V \times B\} \cup$ ($\{w_1 ... w_n\}$, CalculateBenefit($\{w_1 ... w_n\}$)) ;
9      Select (set of) views $w$ from $\{V \times B\}$ for which $B_w$ is MAX and
       $|w| \leq (N - |W'|)$;
10     $W' = W' \cup w$ ; $W = W \setminus w$ ;
11 return $W'$ ;
```

Algorithm 1. Algorithm for selecting views to materialize in MARVEL

Given all views as candidates, we start by assigning each view initial costs corresponding to the size of the original dataset (line 3). View costs are recalculated in each iteration to take previously selected view(s) into account. Then, we compute the benefit of a candidate view for the cases when it is used to derive a full answer (single path roll-up) to another view in the cube lattice (line 6). The benefit of the candidate view is computed according to the cost model defined in Sect. 4.3. The same algorithm is applied when a view should be computed from a set of views (multiple path roll-up – line 8). In these cases, all the views in the set are considered together.

Having calculated the benefit of the views, the algorithm selects the view with the maximum benefit and adds it to the set of views proposed for materialization. This process is repeated until we have identified N views.

5 Query Rewriting in MARVEL

There are several aspects that complicate the problem of rewriting queries over SQL aggregate views. First, in SPARQL a user query and a view definition may use different variables to refer to the same entity. Thus, the query rewriting algorithms require variable mapping to rewrite a query. A variable mapping maps elements of a triple pattern in the BGP of the view to the same elements of a triple pattern in the BGP of the query. Second, the algorithms need to match the new graph structure that is formed by the *CONSTRUCT* query of the view to the graph patterns of the user query and possibly aggregate and group these data anew. Third, complex and indirect hierarchies present in RDF data complicate query rewriting and need to be taken into consideration.

The rewriting algorithms proposed in [9,18] target conjunctive queries and do not consider grouping and aggregation of data. Therefore, we built upon these algorithms and developed an algorithm to rewrite aggregate queries that identifies the views which may be used for query rewriting and selects the one with the least computational cost.

For ease of explanation, we split the algorithm used in MARVEL for aggregate query rewriting using views into two parts: an algorithm for identifying the best view for rewriting (Algorithm 2) and a query rewriting algorithm (Algorithm 3). In the algorithms, we need to look for dimension *roll-up paths* (RUPs), i.e., path-shaped joins of triple patterns of the form $\{(root, p_1, o_1), (s_2, p_2, o_2), \ldots, (s_n, p_n, d)\}$ where *root* is the root of the BGP, p_x is a predicate from the set of hierarchy steps defined for hierarchies in a cube schema, and triple patterns in the path are joined by subject and object values, e.g., $o_{x-1} = s_x$. We denote such a RUP as $\delta_{p_{dim}}(d_i)$ where p_{dim} is a predicate connecting the root variable to the first variable in the roll-up path and d_i represents the last variable in the path. These algorithms use $\gamma(agg_N)$ and $\gamma(g_N)$ to denote sets of triple patterns in the CONSTRUCT clause $CnPtrn$ $\{(s, p^V_{C1}, g_1), \ldots, (s, p^V_{Cn}, g_n), (s, p^V_{Cm}, agg_m), \ldots, (s, p^V_{Ck}, agg_k)\}$ describing the results of aggregation, e.g., (s, p^V_{Cx}, agg_x), and grouping, e.g., (s, p^V_{Cx}, g_x).

The first step in Algorithm 2 is to replace all literals and IRIs in the user query with variables and corresponding *FILTER* statements (line 2): $(?s, p, \#o) \rightarrow (?s, p, ?o)\ .\ FILTER(?o = \#o)$. We do this to make graph patterns of views and queries more compatible with each other, since the graph patterns in the aggregated views should not contain literals. This may also potentially increase the number of candidate views since we may now use the views grouping by the hierarchy level of the replaced literal and then apply restrictions imposed by the *FILTER* statement.

To make the user query and the view query more compatible, we rename all variable names in the user query to the corresponding variable names in a

```
    Input: Set of materialized views $MV$, query $Q$, data cube schema $S$
    Output: Selected view $w$
 1  $W = \emptyset$ -- Set of candidate views ;
 2  $Q = ReplaceLiteralsAndURI(Q)$ ;
 3  foreach $v \in MV$ do
 4      $Q = RenameVariables(Q, v)$ ;
 5      $\{d_1^Q, \dots d_n^Q\} = FindMinimalRUP(Q)$ ;
 6      $\{d_1^v, \dots d_n^v\} = FindMinimalRUP(v)$ ;
 7      let $\{hlvl(d_1)^Q \dots hlvl(d_n)^Q\}$ be a set of hierarchy levels of $Q$ defined in $S$ ;
 8      let $\{hlvl(d_1)^v \dots hlvl(d_m)^v\}$ be a set of hierarchy levels of $v$ defined in $S$ ;
 9      $agg^Q = \{\varphi(o_1), ..., \varphi(o_n)\}$ -- All aggregate expressions in $Q$ ;
10      $agg^v = \{\varphi(o_1), ..., \varphi(o_m)\}$ -- All aggregate expressions in $v$ ;
11      if $agg^Q \subseteq agg^v$ and $(\{hlvl(d_1)^Q \dots hlvl(d_n)^Q\} \preceq \{hlvl(d_1)^v \dots hlvl(d_m)^v\})$
        such that $hlvl(d_i)^Q \preceq hlvl(d_i)^v$ for all $i$ then
12          $W = W \cup v$;
13  return $w \in W$ with minimal costs ;
```

Algorithm 2. Algorithm for selecting a candidate view

view (line 4). We start from the root variable and replace all occurrences of this variable name in the user query with the name that is used in the view query. We then continue renaming variables that are directly connected to the previously renamed variables. We continue until we have renamed all corresponding variables in the user query.

Afterwards, for each dimension of the query graph pattern we define the appropriate roll-up path that the candidate view should have (lines 5–6). This path depends on the conditions (*FILTER* statements) and/or grouping related to the corresponding hierarchy and is the minimum of both; we take the roll-up paths to variables in *FILTER* and *GROUP BY* for the same dimension and keep only the triple patterns that are the same in both – common RUP. For example, if the query groups by regions of a country but the *FILTER* statement restricts the returned values to only some cities ($Region \preceq City$), the required level of the hierarchy in the view should not be higher than the City level.

Then, we identify the hierarchy levels for all dimensions in the query and all dimensions in a view and compare them. We check that the hierarchy levels of all dimensions defined in the view do not exceed the needed hierarchy levels of the query and that the set of aggregate expressions defined in a view may be used to compute the aggregations defined in the query. The views complying with these conditions are added to the set of candidate views (line 12). Out of these views we select one with the least cost for answering the query (line 13).

Let us consider an example. Given the materialized view described in Listing 1.2 and the query of Listing 1.1, the system renames all variables in the query to the corresponding variable names in the view (i.e. *?place* $\rightarrow$ *?vCity*; *?fact* $\rightarrow$ *?obs*; *?val* $\rightarrow$ *?cons*) and defines the roll-up paths for the dimensions in the query (i.e. (*?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity*) and (*?fact gol:reportDateTime ?date*)). Note that the roll-up path in the Date dimension contains the Date level and not the Month level since the query groups by dates. Then the system identifies the roll-up paths for the dimensions in the view (i.e. (*?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity*) and (*?fact gol:reportDateTime ?date*)) and compares them. The system also identifies aggregation expressions in the query and the view (*?fact qb:observation/gol:utilityConsumption ?cons, (SUM(?cons) as ?value)*). Since the view contains the same aggregate expression and all necessary dimensions and the hierarchy levels of the dimensions in the view do not exceed those in the query, this view is added to the set of candidate views.

Given one of the collected views, MARVEL uses Algorithm 3 to rewrite a query. For every dimension in the query we identify the common roll-up path in the query and the view. In the rewritten query Q', these triple patterns will be replaced by the triple patterns from the *CONSTRUCT* clause of the view ($\gamma^V(c^V)$). The remaining triple patterns belonging to the dimensions ($\Delta(d^Q)$) remain unchanged (lines 4–11).

```
   Input: View $v$, query $Q$
   Output: Rewritten query $Q'$
1  $GP' = \emptyset$; $RD' = \emptyset$; $GBD = vars^Q_{GRP}$;
2  let $\Phi^Q$ be assignment and constraint functions of $Q$ ;
3  $GBGP' = \emptyset$; -- A graph pattern of GRAPH statement ;
4  $qDims = \{\delta_p(d^Q)\dots\}$ -- Set of RUP in query $Q$ ;
5  $vDims = \{\delta_p(d^v)\dots\}$ -- Set of RUP in view $v$ ;
6  foreach $\delta_p(d^Q) \in qDims$ do
7      $\delta_p(c^Q) = \delta_p(d^Q) \cap \delta_p(d^v)$ -- Common RUP in $Q$ and $v$ ;
8      $\Delta(d^Q) = \delta_p(d^Q) \setminus \delta_p(c^Q)$ -- Remaining part of a RUP (remaining triple
       patterns) in $Q$ after subtracting the part in common with v;
9      let $\gamma^v(c^v)$ be a triple pattern $\in CnPtrn$ such that $\gamma^v(c^v)$ represents $\delta_p(c^v)$ ;
10     $GP' = GP' \cup \Delta(d^Q)$; $GBGP' = GBGP' \cup \gamma^v(c^v)$;
11     $RD' = RD' \cup \{d^Q\}$;
12 $agg^Q = \{\varphi(o_1), ..., \varphi(o_n)\}$ -- Aggregate expressions in $Q$ over variables
   $\{o_1 \dots o_n\}$ ;
13 $agg^v = \{\varphi(o_1), ..., \varphi(o_m)\}$ -- Aggregate expressions in $v$ over variables
   $\{o_1 \dots o_m\}$ ;
14 foreach $\varphi^Q(x) \in agg^Q$ do
15     let $\gamma^v(x)$ be a triple pattern $\in CnPtrn$ such that $\gamma^v(x)$ represents
       $\varphi^v(x) \in agg^v$ and $\varphi^v = \varphi^Q$ ;
16     $GBGP' = GBGP' \cup \gamma^v(x)$ ;
17     $RD' = RD' \cup \{f'(\gamma^v(x))\}$ where $f'$ is a rewrite of the aggregate function $\varphi$ ;
18 $GP' = GBGP' \cup GP' \cup \Phi^Q)$;
19 $Q' = SELECT\ RD'\ WHERE\ GP'\ GROUP\ BY\ GBD$ ;
20 return $Q'$ ;
```

Algorithm 3. Algorithm for query rewriting using a view

Afterwards, the algorithm compares the aggregate functions of the query and the *SELECT* clause of the view and identifies those that are needed for rewriting. We add the corresponding triple pattern from the *CONSTRUCT* clause and rewrite the aggregate functions to account for the type of the function (algebraic or distributive) (lines 12–17). *GROUP BY* and *ORDER BY* clauses do not change. Additionally, the triple patterns of the *CONSTRUCT* clause will be placed inside the *GRAPH* statement of the SPARQL query to account for the different storage of the view triple patterns (lines 10, 16, 18).

```
SELECT ?date ?vCity (SUM(?value) as ?aggValue)
FROM <http://data.gov.uk>  FROM NAMED <http://data.gov.uk/matview1>
WHERE { GRAPH <http://data.gov.uk/matview1> { ?id gol:reportDate ?date;
     gol:reportLocality ?vCity;  gol:utilityConsumption ?value. }
  ?month skos:narrower ?date . ?month gol:value ?mVal .
  FILTER (?mVal = 'September 2015')  } GROUP BY ?date ?vCity
```

Listing 1.3. Rewritten query

Listing 1.3 shows the result of rewriting the query from Listing 1.1 using the view from Listing 1.2. The algorithm identifies common roll-up paths for the two dimensions in the view and in the query: *?fact gol:refBuilding/org:siteAddress/vc:adr/vc:locality ?vCity* and *?fact gol:reportDateTime ?date*. The system replaces these triple patterns with the triple pattern from the *CONSTRUCT* clause and puts these replaced triple patterns inside the *GRAPH* statement. The remaining triple patterns in the Date dimension (*?month skos:narrower ?date .* and *?month gol:value ?mVal .*) are added to the query graph pattern outside the *GRAPH* statement. The aggregate function is rewritten; since *SUM* is a distributive function, it is rewritten using the same aggregation (*SUM*). All assignment and constraint functions (e.g., *FILTER*) are copied to the rewritten query.

6 Evaluation

To evaluate the performance gain for queries executed over materialized views against the queries over the raw data, we implemented MARVEL using the .NET Framework 4.0 and the dotNetRDF (http://dotnetrdf.org/) API with Virtuoso v07.10.3207 as triple store. The times reported in this section represent total response time, i.e., they include query rewriting and query execution. All queries were executed 5 times following a single warm-up run. The average runtime is reported for all queries. The triple store was installed on a machine running 64-bit Ubuntu 14.04 LTS with CPU Intel(R) Core(TM) i7-950, 24GB RAM, 600GB HDD.

6.1 Datasets and Queries

Unfortunately, none of the benchmarks for SPARQL queries are applicable to our setup. Data generators for benchmarks produce a complete set of data; they do not have an option to withhold some data and generate instead the implicit data that can be used to derive the missing information. Furthermore, existing benchmarks either do not define analytical SPARQL queries or do not require

RDFS entailment to answer these queries. Therefore, we decided to test our approach on 2 different datasets and adapt the data generators and queries to our needs. All queries, schemas, and datasets are available at http://extbi.cs.aau.dk/aggview.

LUBM [10] uses an ontology in the university domain. We decided to build our data cube and corresponding queries on the information related to courses. Inspired by [3], we defined 6 analytical SPARQL queries (using *COUNT*) involving grouping over several classification dimensions. These queries compute the number of courses offered by departments, number of courses taught by professors in each department, number of graduate courses in each department, etc. The data cube schema is defined in QB4OLAP, specifying 3 dimensions (Student, Staff, and Course), hierarchy levels, and steps between the levels. In total, the schema contains 183 triples.

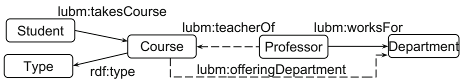


Fig. 2. Excerpt of an altered LUBM schema

We changed the data generator to omit some information that relates staff to courses. Instead, we introduced information about the department that offers these courses (*lubm:offeringDepartment*). In this case, the roll-up path Course → Staff → Department needs to be complemented by the roll-up path Course → Department and the aggregation of courses by Department cannot be answered by the results of the aggregation by Staff. A simplified schema of the data structure is presented in Fig. 2. We generated 3 datasets containing 30, 100, and 300 universities (4, 13.5 and 40M triples). We applied Algorithm 1 to select a set of views providing a good performance gain for answering user queries. The execution of the algorithm on a data cube lattice with 60 nodes and known view sizes took 213 ms.

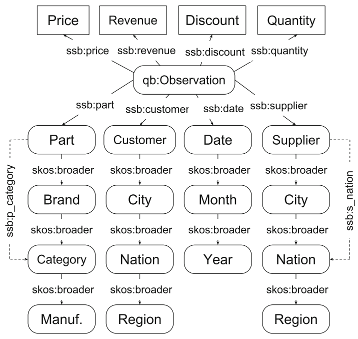
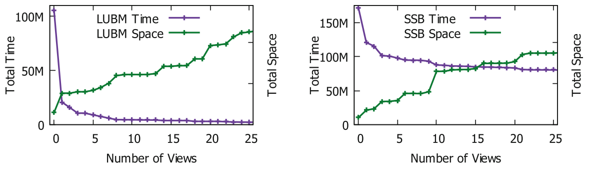


Fig. 3. SSB Dataset in QB4OLAP format

To choose which views to materialize, we ran MARVEL's view selection algorithm and measured (i) the total query response time for all queries in the cube using materialized views whenever possible and (ii) the total space these views require. The unit in which we measured both space and time consumption is the number of triples. The results for the first 25 views sorted by their benefit are presented in Fig. 4a. Based on these results we decided to materialize the first 5 views where the benefit in total response time for the views is good compared to the growth in space consumption for storing these views. Selecting more views substantially increases the used space while the total query time does not decrease significantly. Generating the views took 2:49, 8:38, and 18:47 min with 5, 15, and 33M triples in all views.

In our experiments we also used the Star Schema Benchmark (SSB) [20], originally designed for aggregate queries in relational database systems. This benchmark is well-known in the database community and was chosen for its well-defined testbed and its simple design.

The data in the SSB benchmark represent sales in a retail company; each transaction is defined as an observation described by 4 dimensions (Parts, Customers, Suppliers, and Dates). We translated the data into the RDF multidimensional representation (QB4OLAP) introducing incompleteness to this dataset as well, as illustrated in Fig. 3. An observation is connected to dimensions (objects) via certain predicates. Every connected dimension object is in turn defined as a path-shaped subgraph. Hierarchies in dimensions are connected via the *skos:broader* predicate. Measures (represented as rectangles in Fig. 3) are directly connected to observations. We changed the data generator to omit some information that relates suppliers to their corresponding cities in the Supplier dimension (and parts to their brands in the Part dimension). Instead, we connected suppliers with missing city information directly to their respective nations (*ssb:s_nation*) and parts with missing brand information directly to the categories (*ssb:p_category*). Thus, in the roll-up path Supplier → City → Nation → Region the City level is incomplete. The Part dimension is affected in the level Brand (Part → Brand → Category → Manufacturer). We used scaling factors 1 to 3 to obtain datasets of different sizes (122 to 365M triples).



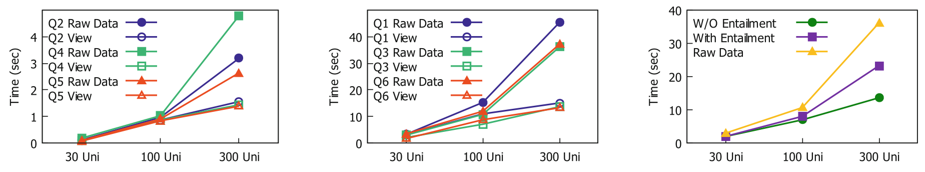
(a) LUBM Dataset (b) SSB Dataset

Fig. 4. Time and space vs number of views

SSB defines 13 classic data warehouse queries that are typical in business intelligence scenarios. We converted all 13 queries into SPARQL. Then we applied Algorithm 1 to select a set of materialized views. The execution of the algorithm on a cube lattice with 500 nodes and known view sizes took 11.8 seconds. We then conducted the same time and space analysis as described above (Fig. 4b). We identified and materialized 6 views with the maximum benefit and stored these views in named graphs. Generating the views took 20:42, 43:21, and 59:48 min with 104, 191 and 277M triples in all views.

6.2 Query Evaluation

LUBM. Figure 5 shows the results of executing the LUBM queries for 3 scale factors – queries with similar runtimes are grouped into separate graphs for better visualization. For queries over raw data we materialized the implicit triples and saved them to the dataset to avoid the entailment during query execution. Note that the performance gain for queries over materialized views becomes more evident with the growth in the volume of data, due to the growing difference in



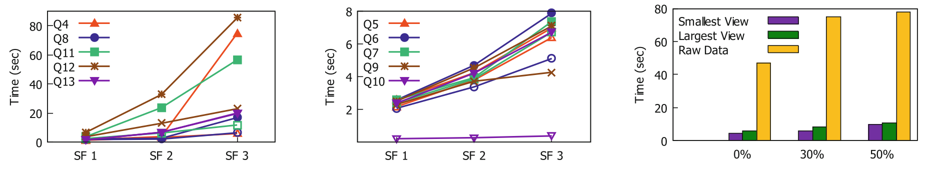
(a) Queries 2, 5 and 4 (b) Queries 1, 3 and 6 (c) Entailment in views

Fig. 5. Execution times of LUBM queries over raw data and views

their sizes. For scale factor 3 the execution of the queries over materialized views is on average 3 times faster.

We also compared the performance of the queries over views that take implicit triples into account and those that do not. Query 3 requests information on the number of courses taken by research assistants whose advisors are professors. We materialized 2 views: one takes into account that all professor ranks are subclasses of the more general class Professor and the other view does not. The execution of Query 3 over the view with implicit information for scale factor 3 was 1.7 times faster than the execution over the other view (Fig. 5c).

SSB. Given the set of materialized views, MARVEL was able to rewrite 10 out of the 13 queries. The other 3 benchmark queries (Q1, Q2, and Q3) apply restrictions on measures. Since the views group by dimensions and only store aggregates over the measures, these queries cannot be evaluated on any aggregate view.



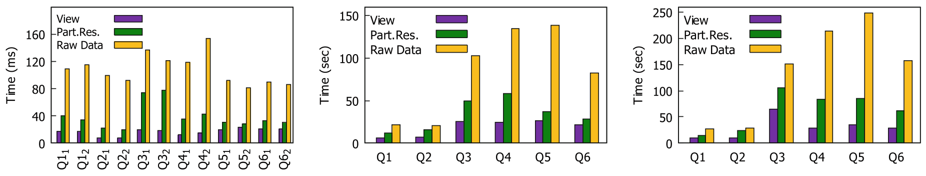
(a) SSB queries (b) SSB queries (c) Various incompleteness

Fig. 6. Execution times of SSB queries over raw data and views

Figure 6 shows the runtime of the queries evaluated on the original datasets and on the views (dashed lines of the same colors indicate the execution times over views). Our results for scale factor 3 show that evaluating queries using views is on average 5 times faster (up to 18 times faster for Query 10). This can be explained by the decreased size of the data and the availability of partial results.

We also compared the performance gain for queries over views with different levels of incompleteness. For scale factor 3, we generated datasets with 0 %, 30 %, and 50 % levels of incompleteness and identified a set of views for every dataset. In each case, the set of materialized views is different due to the difference in the size and the benefit of the views. We then evaluated the execution of Query 4 over the raw data and over the largest and the smallest view. The slight increase in the query execution time over the raw data for incomplete datasets is caused by a rewriting of the query into a more complex query. The results show that in all cases the evaluation of queries over views is far more beneficial (on average 11 times more beneficial – Fig. 6c).

Additionally, we compared the performance gain of MARVEL to the approach in [3] which materializes partial results of user queries to answer subsequent queries. We used the original (non-modified) LUBM dataset containing approx. 100M triples, analytical queries, and views introduced in the technical report of [3]. The execution times for the queries over the original data and views are reported in Fig. 7. As shown in the figure, MARVEL is on average more than twice as fast as partial result materialization [3]. This can be explained by the difference in the size of the data – partial results contain identifiers for facts while our materialized views contain aggregated data only.



(a) Slice (b) Dice (c) Roll-up

Fig. 7. Comparison with results from [3]

In summary, the experimental results show that MARVEL accounts for RDF-specific requirements and finds an appropriate set of views that provide a good balance between the benefit of the views and their storage space. The rewriting algorithm of MARVEL is able to rewrite analytical SPARQL queries based on a set of materialized views. The experiments also show that evaluating queries over materialized views is on average 3–11 times faster than evaluating the queries over raw data.

7 Conclusion and Future Work

In this paper, we have addressed the problem of selecting a set of aggregate RDF views to materialize and proposed a cost model and techniques for choosing these views. The selected materialized views account for implicit triples present in the dataset. The paper also proposes a SPARQL syntax for defining RDF views and an algorithm for rewriting user queries given a set of materialized RDF views.

A comprehensive experimental evaluation showed the efficiency and scalability of MARVEL resulting in 3–10 times speedup in query execution. In future work, we plan to investigate algorithms for incrementally maintaining the materialized views in the presence of updates.

Acknowledgments. This research is partially funded by the Erasmus Mundus Joint Doctorate in "Information Technologies for Business Intelligence – Doctoral College".

References

1. Abelló, A., Romero, O., Pedersen, T.B., Berlanga, R., Nebot, V., Aramburu, M., Simitsis, A.: Using semantic web technologies for exploratory OLAP: a survey. TKDE **27**(2), 571–588 (2015)
2. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: VLDB, pp. 496–505 (2000)
3. Azirani, E.A., Goasdoué, F., Manolescu, I., Roatis, A.: Efficient OLAP operations for RDF analytics. In: ICDE Workshops, pp. 71–76 (2015)
4. Castillo, R., Rothe, C., Leser, U., RDFMatView: indexing RDF data using materialized SPARQL queries. In: SSWS (2010)
5. Colazzo, D., Goasdoué, F., Manolescu, I., Roatis, A.: RDF analytics: lenses over semantic graphs. In: WWW, pp. 467–478 (2014)
6. Darari, F., Nutt, W., Pirrò, G., Razniewski, S.: Completeness statements about RDF data sources and their use for query answering. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 66–83. Springer, Heidelberg (2013)
7. Etcheverry, L., Vaisman, A., Zimányi, E.: Modeling and querying data warehouses on the semantic web using QB4OLAP. In: Bellatreche, L., Mohania, M.K. (eds.) DaWaK 2014. LNCS, vol. 8646, pp. 45–56. Springer, Heidelberg (2014)
8. Galárraga, L., Teflioudi, C., Hose, K., Suchanek, F.: Fast rule mining in ontological knowledge bases with AMIE+. VLDB J. **24**(6), 707–730 (2015)
9. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View selection in semantic web databases. PVLDB **5**(2), 97–108 (2011)
10. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**(2–3), 158–182 (2005)
11. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: VLDB, pp. 358–369 (1995)
12. Hagedorn, S., Hose, K., Sattler, K-U., Umbrich, J.: Resource planning for SPARQL query execution on data sharing platforms. In: COLD (2014)
13. Halevy, A.: Answering queries using views: a survey. VLDB J. **10**(4), 270–294 (2001)
14. Harinarayan, V., Rajaraman, A., Ullman, J.: Implementing data cubes efficiently. ACM SIGMOD **25**, 205–216 (1996)
15. Ibragimov, D., Hose, K., Pedersen, T.B., Zimányi, E.: Towards exploratory OLAP over linked open data – a case study. In: Castellanos, M., Dayal, U., Pedersen, T.B., Tatbul, N. (eds.) BIRTE 2013 and 2014. LNBIP, vol. 206, pp. 114–132. Springer, Heidelberg (2015)
16. Ibragimov, D., Hose, K., Pedersen, T.B., Zimányi, E.: Processing aggregate queries in a federation of SPARQL endpoints. In: Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A. (eds.) ESWC 2015. LNCS, vol. 9088, pp. 269–285. Springer, Heidelberg (2015)

17. Kämpgen, B., Harth, A.: No size fits all – running the star schema benchmark with SPARQL and RDF aggregate views. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 290–304. Springer, Heidelberg (2013)
18. Le, W., Duan, S., Kementsietsidis, A., Li, F., Wang, M.: Rewriting queries on SPARQL views. In: WWW, pp. 655–664 (2011)
19. Nebot, V., Berlanga, R.: Building data warehouses with semantic web data. DSS **52**(4), 853–868 (2012)
20. O'Neil, P., O'Neil, E.J., Chen, X.: The star schema benchmark (SSB). Technical report, UMass/Boston, June 2009
21. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
22. Srivastava, D., Dar, S., Jagadish, H., Levy, A.: Answering queries with aggregation using views. In: VLDB, pp. 318–329 (1996)
23. Theodoratos, D., Ligoudistianos, S., Sellis, T.K.: View selection for designing the global data warehouse. DKE **39**(3), 219–240 (2001)
24. Vaisman, A., Zimányi, E.: Data Warehouse Systems - Design and Implementation. Springer, Berlin (2014)
25. WWW Consortium. SPARQL 1.1 Query Language (W3C Recommendation 21 March 2013). http://www.w3.org/TR/sparql11-query/