RDFViewS: A Storage Tuning Wizard for RDF Applications *

François Goasdoué Konstantinos Karanasos Julien Leblay
Ioana Manolescu
Leo team, INRIA Saclay and LRI, Université Paris-Sud 11
Parc Club Orsay Université, 4 rue J. Monod, 91893 Orsay Cedex, France
firstname.lastname@inria.fr

Abstract

L'émergence du Web sémantique et la multiplication des applications qui y sont liées nous conduisent à rechercher les moyens d'interroger efficacement de grands volumes de données RDF. Dans cette démonstration, nous présentons RDFViewS, un système permettant de trouver automatiquement le meilleur ensemble de vues à matérialiser, pour un ensemble de requêtes SPARQL donné. La solution doit minimiser conjointement le temps d'évaluation des requêtes, le coût de maintenance des vues et l'espace qu'elles occupent. L'algorithme sur lequel repose notre système explore un espace d'états au moyen de stratégies et d'heuristiques, en quête d'une configuration optimale. Ce faisant, il tient compte d'éventuels schémas RDFS accompagnant les données pour garantir la complétude du résultat des requêtes.

Keywords gestion de données RDF, vues matérialisées, optimisation de requêtes, RDFS.

1 Outline

RDF data is increasingly used in data management applications related to traditional Computer Science topics, such as search engines, semantic annotations, social tagging etc. RDF is being used as well in contexts beyond this traditional scope, for instance, it is becoming prevalent in many Life Science and in particular BioInformatics applications. These and other applications have significantly increased the volumes of RDF data to be handled. The size, the complexity and the irregularity of this data pose significant challenges to the task of building an efficient query evaluation engine.

RDF data consists of triples of the form (*subject*, *property*, *object*). This seemingly simple data model leads to complex queries and expensive evaluation, since any meaningful question requires forming chains of several triples, which are translated to many-join queries over a single, huge table containing all the triples. One approach taken to handle such large data volumes consists in mapping the data into one or several relations, and storing them in a relational

*This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004.

database management system (RDBMS), possibly endowed with specific indexes [1, 9]. Then, RDF queries expressed in SPARQL can be translated to SQL queries [2], which are evaluated by the RDBMS. Another approach consists in developing RDF-specific stores and query processors [7], which still share some of the standard notions and features of relational storage engines.

These efforts aim at providing a generic, one-size-fits-all storage model for RDF. However, decades of research and development of RDBMSs have shown that huge performance gains can be achieved by tuning the storage to the data sets and to the requirements of specific applications. This is typically achieved by establishing *materialized views and/or indices* specific to the data and workload [8].

Another important aspect of RDF data management is that rich semantic information can be associated to the data sets, e.g., in the form of an RDF Schema. In this situation, schema-based reasoning may lead to finding answers to a query, which would simply not be found when querying the data alone. Thus, the interpretation of RDF queries may be affected by the existence of associated semantics, and this must be considered when designing a query-inspired set of views. For instance, if an RDF Schema specifies that a *VisitingProfessor* is also a *Professor*, then such visiting professors should be taken into account in the answers of the question "What are the addresses of professors working in French Universities?". In the absence of a schema, they would not be included.

We propose to demonstrate RDFViewS, a system that focuses on automatically choosing the materialized views which are most appropriate for a given data set and query workload. The tool provides many options to guide the search, into which it also incorporates the insights brought by an RDF Schema, if one is available. RDFViewS outputs a set of proposed materialized views (which are automatically created within an RDBMS), as well as a set of rewritings of the original workload, in terms of these materialized views. RDFViewS can be seen as a storage tuning wizard for RDF data, to be used in conjunction with off-the-shelf RDBMSs. The tool's various steps and options can be easily inspected and controlled via a GUI by RDFViewS' target users: administrators of large RDF databases.

RDFViewS was originally presented in [4]. Our work on view selection for RDF databases, including the algorithms behind RDFViewS, is presented in detail in [5]. An initial version of [5] also appeared in [3].

2 Problem Model

RDFViewS takes as input a set of conjunctive SPARQL queries. Each query is endowed with a weight, reflecting its relative importance (e.g. how often it is evaluated).

We model our problem as a search state optimization problem, based on an existing proposal for selecting views to materialize in a relational setting [8], which we adapted to the peculiarities of the RDF model. For a given query workload Q, we define a state as the pair $S_i(Q) = \langle V_i, R_i \rangle$, where V_i is the set of views to materialize and R_i the rewritings needed to answer the queries of Q using exclusively the views in V_i. We use four transitions, which can be applied to a given state and yield a new one, namely *selection cut*, *join cut*, *view break* and *view fusion*. Intuitively, the first three aim at relaxing the queries: *selection cut*, *join cut* by removing some predicates, and *view break* by splitting a query into smaller ones, possibly with overlapping predicates and atoms. The last transition attempts to fuse two candidate views,

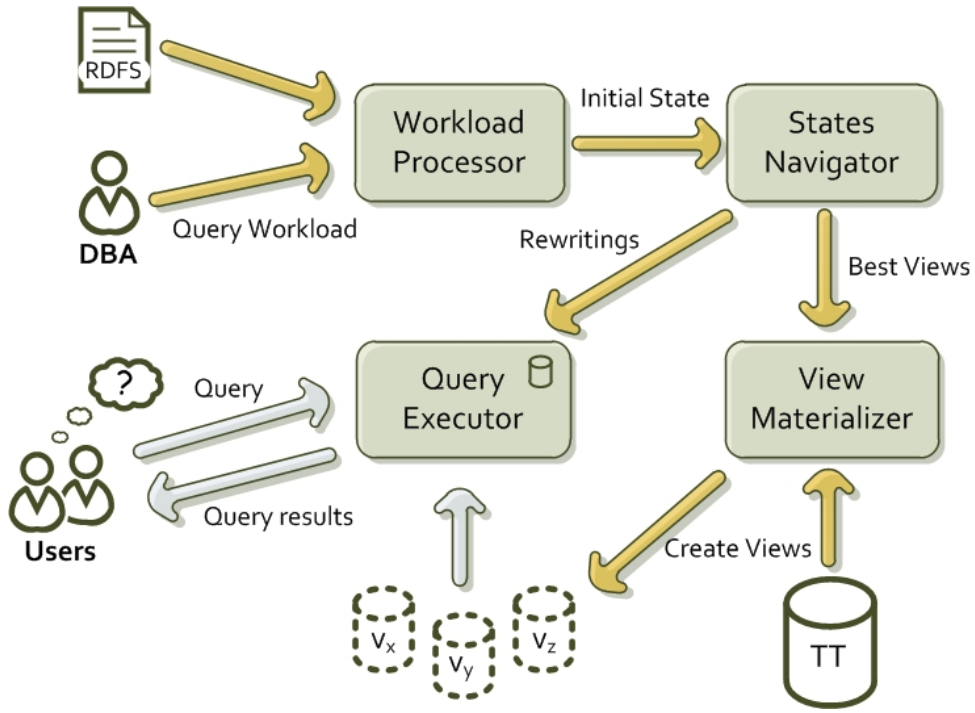

Figure 1: RDFViewS architecture.

replacing them with a single one. The relaxation steps help finding common sub-expressions among initial queries, such that *view fusion* may then be applied. If the workload queries have common sub-queries, these will be identified as useful views to materialize.

The cost of each state is assessed using a *cost function*, which reflects the query execution time, the view maintenance cost and the space needed for materializing the views of the state. Starting from an initial state, we apply the transitions and navigate in the search space according to a search strategy. As the *initial state* of our search, we choose the one that proposes to materialize exactly the query workload (best execution time). At the end of the search, we return the state with the minimum combined cost.

3 System Architecture

The architecture of RDFViewS is depicted in Figure 1. The RDF data is initially stored into an RDBMS as a single triple table (*TT*); for efficiency, and following many similar works [6], the table is dictionary-encoded, i.e., URIs and string constants are assigned distinct integers, and the *TT* table stores triples of integers. The database administrator (*DBA*) uses RDFViewS to further tune the store. To this end, she provides the SPARQL query workload to the **Workload Processor** through a graphical interface. In the presence of an RDF Schema, the queries are reformulated, compiling the knowledge of the Schema inside them and transforming each query to a union of queries [5]. The (possibly reformulated) queries are used to create the *initial state* of the search.

The initial state is then loaded to the **States Navigator**, which constitutes the gist of our system. We have devised exhaustive strategies that navigate through the whole search space. However, as the problem we address is known to be well above exponential, we employ heuristics which significantly prune the search space. Moreover, we provide the option to apply some

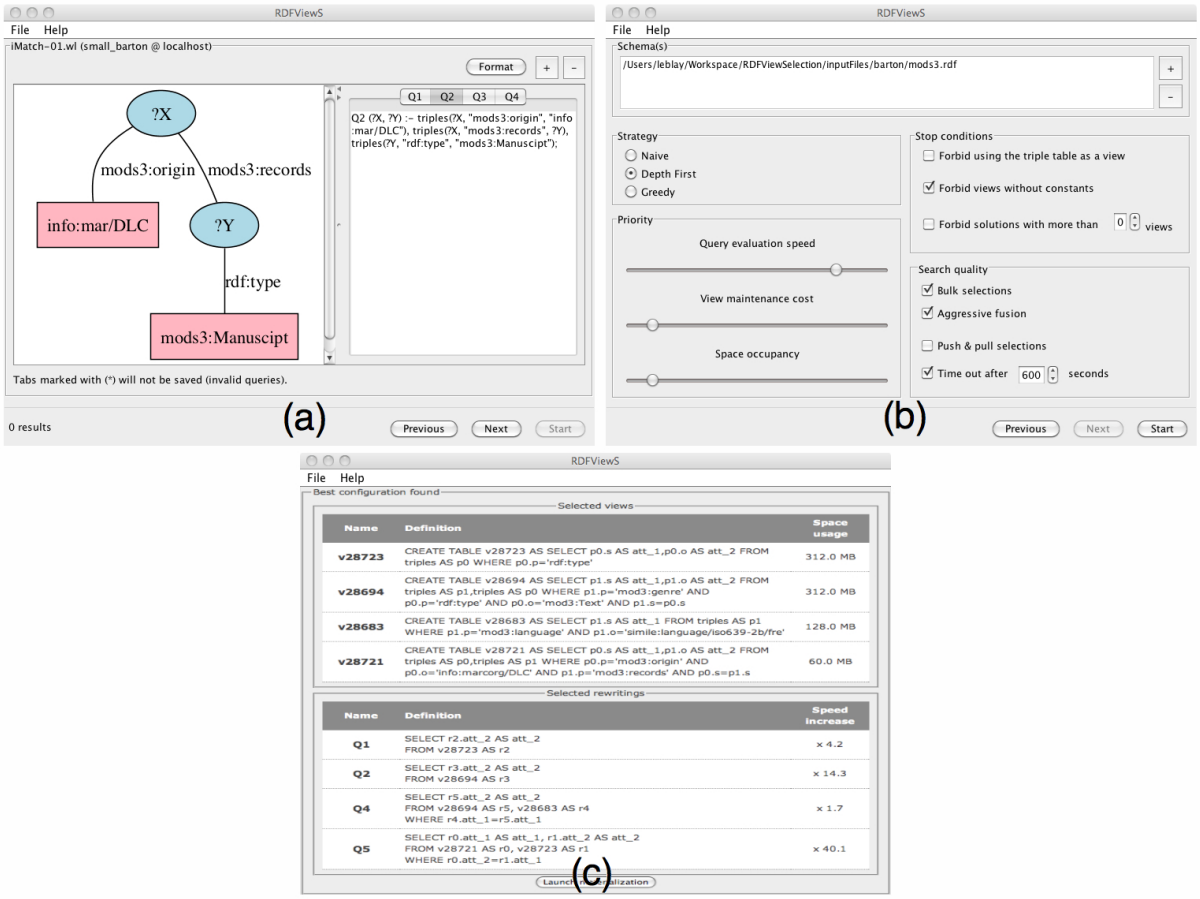

Figure 2: Screen captures of the wizard interface: (a) Query Processor, (b) State Navigator's settings, (c) selected views with statistics

additional stop conditions: we identify states that have some specific characteristics and we do not allow more transitions to be applied on these states. More details about the search strategies can be found in [5].

Once the search is finished, we obtain the best state according to our quality function and we materialize the views of this state, after translating them to SQL. Then, we push the rewritings contained in the best state to the **Query Executor**, which stores them for future use. Whenever a user issues to the system a query from the workload, the Query Executor uses the stored rewritings to efficiently answer the query by using the already materialized views.

4 Demonstration Scenario

Our system has been fully implemented in Java 6. The triple table and the materialized views are stored in PostgreSQL v8.4.4. We have built a stand-alone interface which enables users to interact with the system, extensively parameterize it and follow in detail the view selection process. Screen captures of Figure 2 depict key stages of the application workflow. More screen captures of the system can be found at the RDFViewS website[1].

[1] http://rdfvs.saclay.inria.fr

Demo attendees will play the role of a database administrator. Using the interface, they will first choose one of the pre-loaded RDF datasets (among others, some of the most widely-used RDF datasets will be available: Barton, Yago, Uniprot and LUBM), and the query workload for which they want to tune the database. They may also load their own datasets, modify the existing query workloads or add new ones (Figure 2.a). The queries can be modified either by using a SPARQL editor or through a visual editor we have created.

Before initializing the search for the best view configuration, attendees will define some additional details of the searching process, according to their specific preferences, and pick the RDF Schema(s) they wish to use (Figure 2.b). In particular, they will choose whether they prefer a quick search, or a search that lasts longer but guarantees the optimal solution. Furthermore, they will tune the cost function by adjusting the weights of its components (giving more importance to the query execution time, to the view maintenance or to the space needed).

At the end of the search, the selected views are displayed, together with the cost gains (Figure 2.c). Moreover, a graphical overview of the search space can also be shown. This information acts as feedback to the user, which may choose to tune differently the cost functions in a subsequent search.

To verify the performance benefits brought by RDFViewS, attendees will then act as simple users issuing queries, which will be first answered against the triple table and then by exploiting the materialized views.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] A. Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.*, 68(10), 2009.

[3] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. Materialized View-Based Processing of RDF Queries. In *Bases de Données Avancées*, Toulouse France, 2010.

[4] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In *CIKM*, 2010.

[5] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(1), 2012.

[6] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. of VLDB*, 1(1), 2008.

[7] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, New York, NY, USA, 2009. ACM.

[8] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3), 2001.

[9] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. of VLDB*, 1(1), 2008.