

Framework for Live Synchronization of RDF Views of Relational Data

Vânia M. P. Vidal^{✉1}, Narciso Arruda¹, Matheus Cruz¹, Marco A. Casanova², Valéria M. Pequeno³, Ticianne Darin¹

¹Federal University of Ceará, Fortaleza, CE, Brazil
{vvidal, narciso}@lia.ufc.br, matheusmayron@gmail.com,
ticianne@virtual.ufc.br

²Department of Informatics – Pontifical Catholic University of Rio de Janeiro, RJ, Brazil
casanova@inf.puc-rio.br
INESC-ID and Universidade Autónoma de Lisboa, Porto Salvo, Portugal
vmp@inesc-id.pt

Abstract. This Demo presents a framework for the live synchronization of an RDF view defined on top of relational database. In the proposed framework, rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay synchronized with the relational database. The computed changesets are then used for the incremental maintenance of the RDB-RDF views as well as application views. The Demo is based on the LinkedBrainz Live tool, developed to validate the proposed framework.

Keywords: RDF view · View Maintenance · Linked Data · Relational database.

1 Introduction

There is a vast content of structured data available on the Web of Data as Linked Open Data (LOD). In fact, a large number of LOD datasets are RDF views defined on top of relational databases, called *RDB-RDF views*. The content of an RDB-RDF view can be materialized to improve query performance and data availability. However, to be useful, a materialized RDB-RDF view must be continuously maintained to reflect dynamic source updates.

Also, Linked Data applications can fully or partially replicate the contents of a materialized RDB-RDF view, by creating *RDF application views* defined over the RDB-RDF view. The generation of RDF application views improves the efficiency of applications that consume data from the LOD, and increases the flexibility of sharing information. However, the generation of RDF application views raises synchronization problems, since the original datasets can be continuously updated. Thus, updates on an RDB-RDF view must be propagated to maintain the RDF application views.

A popular strategy used by large LOD datasets to maintain RDF application views is to compute and publish changesets, which indicate the difference between two states of the dataset. Applications can then download the changesets and synchronize

their local replicas. For instance, DBpedia (<http://wiki.dbpedia.org>) and LinkedGeoData (<http://linkedgeodata.org/About>) publish their changesets in a public folder.

In this demo, we show a framework, based on rules, that provides live synchronization of RDB_RDF views. In the proposed framework (see Fig. 1), rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay in synchronization with the relational database. The computed changesets are used by the synchronization tools for the incremental maintenance of RDB_RDF views and application views. In [7] we present a formal framework for automatically generating, based on the view mappings, the rules for computing correct changesets for an RDB-RDF view. Based on the mappings, at view definition time, we are able to: (i) identify all relations that are relevant for the view; and (ii) define the rules that compute the changeset required to maintain the view w.r.t an update over a relevant relation. Our formalism allows us to precisely justify that the rules generated by the proposed approach correctly compute the changeset. The demo video is available at <http://tiny.cc/videoivesynrdrdf> (see also <http://www.arida.ufc.br/livesynrdrdf/>).

The remainder of this paper is organized as follows. Section 2 describes our strategy, based on rules, for computing changesets for an RDB-RDF view. Section 3 summarizes related work. Section 4 covers an implementation and experiments. Section 5 presents the conclusions.

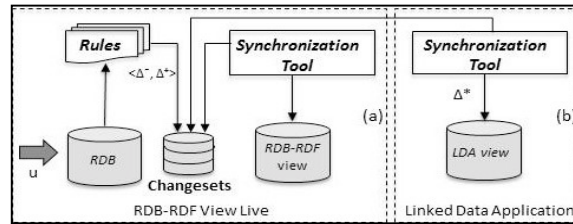


Fig. 1. Framework for live synchronization of RDB_RDF view.

2 Computing Changesets for RDB-RDF Views

In our strategy, we first have to identify the relations in \mathcal{S} that are relevant for \mathcal{V} , that is, the relations whose updates might possibly affect the state of the view \mathcal{V} . For each such relation R , we define triggers that are fired immediately before and after an update on R , called *before* and *after triggers*, respectively, and which are such that:

BEFORE Trigger: computes Δ^- the set of deleted triples

AFTER Trigger: computes Δ^+ the set of inserted triples.

The key idea of our strategy for computing the changesets is to re-materialize only the tuples whose RDF_State (the tuple triplification) might possibly be affected by the update. Thus, using Δ^- and Δ^+ , one should be able to compute the new RDF state of the tuples that are relevant to the update (formal definitions in [7]). Fig. 2 shows the templates of the triggers associated with an update on a relation R .

For example, consider u , the UPDATE on R , where r_{old} and r_{new} are the old and new state of the updated tuple, respectively. Before the update, Trigger (a) is fired, and Procedure $COMPUTE_ \Delta^- [R]$ computes Δ^- , which contains the OLD RDF_State of

the tuples that are relevant to V w.r.t update u . After the update, Trigger (b) is fired. Using the database state after the update, Procedure $COMPUTE_ \Delta^+[R]$ computes Δ^+ , which contains the new RDF_State of the tuples that are relevant to V w.r.t update u . Note that procedures $COMPUTE_ \Delta^-[R]$ and $COMPUTE_ \Delta^+[R]$ are automatically generated, at view definition time, based on the view mappings [7]. Triggers for insertions and deletions are similarly defined and are omitted here.

Given V_{OLD} , the old state of the RDB_RDF view, in order to stay synchronized with the new state of database, the new state of the view is computed as $V_{NEW} = (V_{OLD} - \Delta^-) \cup \Delta^+$

BEFORE {UPDATE } ON R THEN $\Delta^- := COMPUTE_ \Delta^-[R](r_{old}, r_{new});$ ADD Δ^- to changeset of V . (a)	AFTER {UPDATE} ON R THEN $\Delta^+ := COMPUTE_ \Delta^+[R](r_{old}, r_{new});$ ADD Δ^+ to changeset of V . (b)
---	---

Fig. 2. Triggers to compute changeset of V w.r.t. updates on R

3 Related Work

The incremental view maintenance problem has been extensively studied in the literature for relational views [2], object-oriented views [6], semi-structured views [1], and XLM Views [3]. Despite their important contributions, none of these techniques can be directly applied to compute changesets for RDB-RDF views.

Comparatively less work addresses the problem of incremental maintenance of RDB-RDF views. Vidal et al. [8] proposed an incremental maintenance strategy, based on rules, for RDF views defined on top of relational data. Although the approach in this paper uses a similar formalism for specifying the view mappings, our strategy to compute changeset present in Section 2 differs considerably.

Faisal et al [4] presented an approach to deal with co-evolution, that is, the mutual propagation of the changes between a replica and its origin dataset. Their approach relies on the assumption that either the source dataset provides a tool to compute a changeset at real-time or third party tools can be used for this purpose. Thus, the contribution of this paper is complementary and relevant to satisfy their assumption.

Konstantinou et al [5] investigated the problem of the incremental generation and storage of the RDF graph that is the result of exporting relational database contents. In their approach, when one of the source tuples change, the whole triples map definition will be executed for all tuples in the affected table. By contrast, using our rules, we are able to identify which tuples are relevant to an update, and only the RDF_State of the relevant tuple are re-materialized.

4 Implementation and Experiments

To test our strategy, we implemented the *LinkedBrainz Live tool (LBL tool)*, which propagates the updates over the *MusicBrainz* database (*MBD* database) to the

LinkedMusicBrainz view (*LMB View*). The *LMB View* is intended to help MusicBrainz (<http://musicbrainz.org/doc/about>) to publish its database as Linked Data. Figure 1 depicts the general architecture of our framework, based on rules, for providing live synchronization of RDB_RDF views. The main components of the *LBL* tool are:

- **Local *MBD* database:** We installed a local copy of the *MBD* database available on March 22, 2017.
- ***LMB View* and Mappings:** We created the R2RML mapping for translating *MBD* data into the Music Ontology vocabulary (<http://musicontology.com/>), which is used for publishing the *LMB view*. The *LMB view* was materialized using the D2RQ tool (<http://www.d2rq.org/>). It took 67 minutes to materialize the view with approximately 41.8 GB of NTriples.
- **Triggers:** We created the triggers to implement the rules required to compute and publish the changesets, as discussed in Section 2.
- ***LBL Synchronization tool:*** This component enables the *LMB View* to stay synchronized with the *MBD* database. It is the same synchronization tool used by the DBpediaLive (<http://wiki.dbpedia.org/online-access/DBpediaLive>). It simply downloads the changeset files sequentially, creates the appropriate INSERT/DELETE statement and executes it against the *LMB View* triplestore.
- ***LBL update extractor:*** This component extracts updates from the replication file provided by MusicBrainz, every hour, which contains a sequential list of the update instructions processed by the MusicBrainz database. When there is a new replication file, the updates should be extracted and then executed against the local database.

In our experiments, we used the replication file with sequential number 103114, which has 4,557 updates. Table 1 and Table 2 summarize our experimental results. Due to space limitation we consider only the relevant relations (RR) *Artist* and *Track*.

- Table 1 shows: The total number of tuples in the RR and the total time (in milliseconds) spent to triplify the tuples in RR.
- Table 2 shows: the total number of updates on the RR, the average number of tuples relevant for updates on the RR, and the average time (in milliseconds) to compute the changeset $\langle \Delta^-, \Delta^+ \rangle$ for insertions (*i*) and updates (*u*) on RR. In the replication file, there is no deletion from relevant relations *Artist* and *Track*.

Table 1. Time spent for triplification on March 22, 2017

Relevant Relation (RR)	Number of Tuple (k)	Triplification Time (ms)
<i>Artist</i>	1,189	340,721
<i>Track</i>	22,087	435,693

Table 2. Time spent for computing changesets w.r.t replication file 103114

Relevant Relation (RR)	Number of Updates	Avg number of Relevant tuples (by update)	Δ^- (avg time) (ms)		Δ^+ (avg time) (ms)	
<i>Artist</i>	31	1.55	25	102	18	5
<i>Track</i>	397	4.05	28	39	4	3

The experiments demonstrated that the runtime for computing the changeset is negligible, when the number of relevant tuples is relatively small. This is what is expected, since the *RDB_RDF* View should be frequently updated to ensure that it remains consistent and up-to-date. Thus, we can conclude that the incremental strategy far outperforms full re-materialization, and also the re-materialization of the affected tables [5].

5 Conclusions

This Demo presented a framework for providing live synchronization of an RDF view defined on top of relational database. In the proposed framework, rules are responsible for computing and publishing the changeset required for the RDB-RDF view to stay synchronized with the relational database. The computed changesets are used for the incremental maintenance of the *RDB_RDF* views as well as application views.

We also implemented the *LinkedBrainz Live tool* to validate the proposed framework. We are currently working on the development of a tool to automate the generation of the rules for computing the changesets.

6 References

1. Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., Wiener, J. L.: Incremental Maintenance for Materialized Views over Semistructured Data. In VLDB 1998, pp. 38–49 (1998)
2. Ceri, S. and Widom, J.: Deriving productions rules for incremental view maintenance. In VLDB 1991, pp. 577–589 (1991)
3. Dimitrova, K., El-Sayed, M., Rundensteiner, E.A.: Order-sensitive View Maintenance of Materialized XQuery Views. In ER 2003, pp. 144–157 (2003)
4. Faisal, S., Endris, K.M., Shekarpour, S., Auer, S.: Co-evolution of RDF Datasets. In: 16th International Conference – ICWE 2016 (2016)
5. Konstantinou, N., Spanos, D.E., Kouis, D., Mitrou, N.: An Approach for the incremental export of relational databases into rdf graphs. International Journal on Artificial Intelligence Tools 24(2), (2015)
6. Kuno, H. A. and Rundensteiner, E. A.: Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. In IEEE TDKE, vol. 10, no. 5, pp. 768–792 (1998)
7. Vidal, V.M.P., Arruda, N., Casanova, M.A., Brito, C., Pequeno, V.M.: Computing Changesets for RDF Views of Relational Data. Technical Report, Federal University of Ceara (2017). Available at <http://tiny.cc/TechnicalReportUFC2017>
8. Vidal, V.M.P., Casanova, M.A., Cardoso, D.S.: Incremental Maintenance of RDF Views of Relational Data. In ODBASE 2013, pp 572-587 (2013)