Update Strategies for DBpedia Live

Claus Stadler+, Michael Martin*, Jens Lehmann*, and Sebastian Hellmann*

Universität Leipzig, Institut für Informatik, Johannisgasse 26,
D-04103 Leipzig, Germany,
+cstadler@informatik.uni-leipzig.de *{lastname}@informatik.uni-leipzig.de
http://aksw.org

Abstract. Wikipedia is one of the largest public information spaces with a huge user community, which collaboratively works on the largest online encyclopedia. Their users add or edit up to 150 thousand wiki pages per day. The DBpedia project extracts RDF from Wikipedia and interlinks it with other knowledge bases. In the DBpedia live extraction mode, Wikipedia edits are instantly processed to update information in DBpedia. Due to the high number of edits and the growth of Wikipedia, the update process has to be very efficient and scalable. In this paper, we present different strategies to tackle this challenging problem and describe how we modified the DBpedia live extraction algorithm to work more efficiently.

1 Introduction

The Linking Open Data (LOD) cloud continues to grow and contains various very large knowledge bases. Many of those knowledge bases are extracted or derived from other sources. Hence, they need to be synchronised to the original source in order to reflect changes in it. This raises performance challenges, in particular if the knowledge bases aim at a timely synchronisation. In this article, we describe how we handle this challenge in the specific case of the DBpedia knowledge base. We hope that this example can spur a broader discussion of this topic.

The DBpedia live extraction framework [5] extracts RDF data from articles in the English Wikipedia[1] with a delay of only a few seconds after they are edited. In brief, the extraction proceeds as follows: When an article is modified, the live extraction framework polls its latest revision via Wikipedia's non public OAIRepository. After that, a set of extractors are run which generate RDF output for this Wikipedia page. Finally, this data is written into a publicly accessible triple store where it can be accessed via Linked Data [3] and SPARQL [9].

As the DBpedia store should only reflect the data extracted from the latest revision of an article, a strategy for identifying and removing outdated triples needs to be employed. In our case the triple management strategy needs to respect two things: Firstly, the DBpedia live store is seeded with the DBpedia

[1] http://en.wikipedia.org

dataset [6, 1] in version 3.4. The seeding is done in order to provide initial data about articles that have not been edited since the start of the live extraction process. As an effect, the extraction takes place on an existing dataset, which not only contains data extracted from the english Wikipedia but also third party datasets, amongst them YAGO[10], SKOS[2], UMBEL[3], and Open-Cyc[4]. On the one hand the live extraction needs to keep the data of the third party datasets intact. On the other hand when an article gets edited, its corresponding data in the seeding dataset must be updated. Since all data resides in the same graph[5], this becomes a complex task. Secondly, the state of the extractors needs to be taken into account. An extractor can be in one of the states *Active*, *Purge*, and *Keep* which affects the generation and removal of triples as follows:

- *Active* The extractor is invoked on that page so that triples are generated.
- *Purge* The extractor is disabled and all triples previously generated by the extractor for a that page should be removed from the store.
- *Keep* The extractor is disabled but previously generated triples for that page should be retained.

Our initial strategy is as follows: Upon the first edit of an article seen by the extraction framework a clean up is performed using the queries described in Section 2.1. The clean up removes all but the static facts from the seeding data set for the article's corresponding resource. The new triples are then inserted together with annotations. Each triple is annotated with its extractor, DBpedia URI and its date of extraction, using OWL 2 axiom annotations. Once these annotations exist, they allow for simple subsequent deletions of all triples corresponding to a certain page and extractor in the event of repeated article edits.

As DBpedia consists of approximately 300 million facts, annotations would boost this value by a factor of six[6]. As the amount of data in the store grew, we soon realized that the update performance of the store became so slow that edits on Wikipedia occurred more frequently than they could be processed. Before resorting to acquire better hardware, we considered alternative triple management approaches.

The paper is structured as follows: We describe the concepts for optimizing the update process in the Section 2. We also provide a short evaluation of the performance improvement that was facilitated by the new deployed update strategy in Section 3. We conclude and present related as well as future work in the Section 4.

---

[2] `http://www.w3.org/TR/skos-reference/`
[3] `http://www.umbel.org`
[4] `http://www.opencyc.org`
[5] http://dbpedia.org
[6] Three triples of the annotation vocabulary (omitting `?s rdf:type owl:Axiom`), and the three annotations (extractor, page, and extraction-date)

2 Concepts for Optimizing the Update Process

The extraction of changed facts in DBpedia, which have changed through edits in Wikipedia, is described in [5]. After changed facts have been computed, the DBpedia knowledge base has to be updated. The formerly used process for updating the model with new information caused some performance problems as described in Section 1. To optimize the update process, we sketch three strategies as follows in this section.

- A **specialized update process** which uses a set of DBpedia-specific SPARUL queries.
- An **update process on the basis of multiple resource specific graphs** which uses separate graphs for each set of triples generated by an extractor from an article.
- An **RDB assisted update process** which uses an additional relational database table for storing temporarily affected RDF resources.

2.1 Specialized Update Process

As our generic solution using annotations turned out to be too slow, we chose to use a domain specific one, in order to reduce the amount of explicit generated metadata. As opposed to our initial approach which allows extractors to generate arbitrary data, we now require the data to satisfy two constraints:

- All subjects of the triples extracted from an Wikipedia article must start with the DBpedia URI corresponding to the article. If the subject is not equal to the DBpedia URI, we call this a *subresource*. For instance, for the article "London" both subjects `dbpedia:London`[7] and `dbpedia:London/prop1` would meet this naming constraint.
- Extractors must only generate triples whose predicates and/or objects are specific to that extractor. For instance, the extractor for infoboxes would be the only one to generate triples whose properties are in the `http://dbpedia.org/property/` namespace.

As a consequence, a triple's subject and predicate implicitely uniquely determine the corresponding article and extractor. Whenever an article is modified, the deletion procedure is as follows: As subresources are so far only generated by the infobox extractor (when recursively extracting data from nested templates) they can be deleted unless this extractor is in state *keep*. Th query for that task is shown in Listing 1.1. Deletion of the main DBpedia article URIs is more complex: Triples need to be filtered by the state of their generating extractor as well as by their membership to the static part of DBpedia. This results in a complex dynamically built query as sketched in Listing 1.2.

[7] The prefix `dbpedia` stands for `http://dbpedia.org/resource/`.

Listing 1.1. Deleting all statements matching part of the URI of London

```
DELETE
FROM <http://dbpedia.org> { ?sub ?p ?o . }
FROM <http://dbpedia.org> {
    <http://dbpedia.org/resource/London> ?p ?sub .
    ?sub ?p  ?o .
    FILTER(REGEX(?sub, '^http://dbpedia.org/resource/London/'>) }
```

Listing 1.2. Deleting resources according to specific extractors while preventing the deletion of the static part

```
DELETE
FROM <http://dbpedia.org>
{ <http://dbpedia.org/resource/London> ?p ?o . }
{ <http://dbpedia.org/resource/London> ?p ?o .
  # Dynamically generated filters based on extractors in
  # active and purge state
  FILTER(REGEX(?p, '^http://dbpedia.org/property/') ||
         ?p = foaf:homepage ||
         # more conditions for other extractors
  ) .
  # Static filters preventing deletion of the static DBpedia part
  FILTER((?p != <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ||
         !(REGEX(?o, '^http://dbpedia.org/class/yago/')) &&
         # more conditions for the static part
  ) .
}
```

2.2 Update Process with Resource Specific Graphs

The previously mentioned attempts have the disadvantage of either introducing a high overhead with respect to the amount of triples needed to store meta data or being very complex. A different approach is to put each set of triples generated by an extractor from an article into its own graph. For instance a URI containing a hash of the extractor and article name could serve as the graph name. The update process then becomes greatly simplified as upon an edit, it is only necessary to clear the corresponding graph and insert the new triples. This approach requires splitting the seeding DBpedia dataset into separate graphs from the beginning. As the DBpedia dataset v3.4 comes in separate files for each extractor, the subjects of the triples in these files determine the target graph. The downside of this approach is, that the data no longer resides in a single graph. Therefore it is not possible to specify the dataset in the SPARQL FROM clause. Instead, a FILTER over the graphs is required as show in Listing 1.3.

Listing 1.3. Selecting triples across multiple graphs.

```
SELECT ?s ?p ?o
{ GRAPH ?g {?s ?p ?o} .
  FILTER(REGEX(?g, '\^http://dbpedia.org/')) .
}
```

2.3 RDB Assisted Update Process

The third approach we evaluated and implemented is to use to store RDF statements in a relational database (RDB) in addition. This approach is motivated

by the observation that most changes made to Wikipedia articles only cause small changes in the corresponding RDF data. Therefore, the idea is to have a method for quickly retrieving the set of triples previously generated for an article, comparing it to the new set of triples and only performing the necessary updates.

For selection of resources which have to be updated after a periodically finished Wikipedia extraction process, we firstly created an RDB table as illustrated in Figure 1. Whenever a Wikipedia page is edited, the extraction method gener-

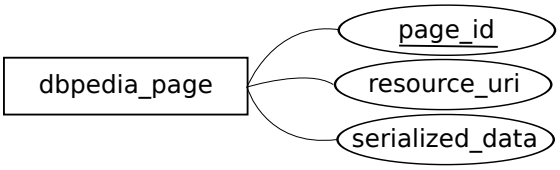

Fig. 1. Definition of the RDB table

ates a JSON object holding information about each extractor and its generated triples. After serialization of such an object, it will be stored in combination with the corresponding page identifier. In case a record with the same page identifier already exists in this table, this old JSON object and the new one are being compared. The results of this comparison are two disjoint sets of triples which are used on the one hand for adding statements to the DBpedia RDF graph and on the other hand for removing statements from this graph. Therefore the update procedure becomes straight forward:

With this strategy, once the initial clean up for a page has been performed, all further modifications to that page only trigger a simple update process. This update process no longer involves complex SPARQL filters, instead it can modify the affected triples directly.

Listing 1.4. SQL Statements for fetching data for a resource

```
SELECT data FROM dbpedia_page
WHERE page_id =  http://dbpedia.org/resource/London ;

INSERT INTO dbpedia_page(page_id, data)
VALUES( http://dbpedia.org/resource/London , <JSON-Object>);

UPDATE dbpedia_page SET data = <JSON-Object> WHERE page_id = <pageId>
```

Listing 1.5. Simple SPARQL Delete and insert queries

```
Delete From <http://dbpedia.org> { ... concrete triples ... }
Insert Into <http://dbpedia.org> { ... concrete triples ... }
```

3 Evaluation of the RDB Assisted Update Process

We did a small evaluation by comparing the RDB assisted update process to a simplified version of the DBpedia specific one. This simplified version deletes

Algorithm 1 Algorithm of the RDB assisted update process

```
//The data to be put into the store is included in the extractionResult
//object
pageId ← extractionResult[pageId]
resourceUri ← extractionResult[resourceUri]
newTriples ← extractionResult[triples]

//Attempt to retrieve previously inserted data for the pageId
jsonObject ← fetchFromSQLDB(pageId)
if jsonObject ≠ NULL then
   oldTriples ← extractTriples(jsonObject)
   insertSet ← newTriples − oldTriples
   removeSet ← oldTriples − newTriples
   removeTriplesFromRDFStore(removeSet)
   addTriplesToRDFStore(insertSet)
else
   cleanUpRDFStore(pageId)
   insertIntoRDFStore(newTriples)
end if

jsonObject ← generateJSONObject(pageId, resourceUri, newTriples)
putIntoSQLDB(jsonObject)
```

triples with a certain subject using 1.6 instead of 1.2. The difference is only that the complex filter patterns were omitted.

Listing 1.6. Example of the simplified delete query

```
DELETE
FROM <http://dbpedia.org>
{ <http://dbpedia.org/resource/London> ?p ?o . }
{ <http://dbpedia.org/resource/London> ?p ?o . }
```

The benchmark simulates edits of articles and was set up as follows. 5000 distinct resources were picked at random from the DBpedia dataset. For each resource two sets O and N were created by randomly picking p% of the triples whose subject starts with the resource. The sets O and N represent the sets of triples corresponding to an article prior and posterior to an edit, respectively. A run of the benchmark first clears the target graph and `dbpedia_page` table. Then each resource' O-set is inserted into the store. Finally the time to update the old sets of triples to the new ones using either the simplified specialized update strategy or the RDB assisted one[8] is measured. Additionally the total number of triples that were removed ($O - N$), added ($N - O$) and retained ($N \cap O$) were counted. Three runs were performed with $p = 0.9$, $p = 0.8$, and $p = 0.5$ meaning that the simulated edits changed 10%, 20% and 50% of the triples, respectively. We assume that the actual ratio of triples updated by the

[8] As this approach involves a JSON object holding information about each extractor, the generation of the sets O and N was related to a dummy extractor

live extraction process in the event of repeated edits of articles is between 10 and 20 percent. However the exact value has not been determined yet. The benchmark was run on machine with a two core 1,2GHz Celeron CPU and 2GB Ram. The triple store used was "Virtuoso Open-Source Edition 6.1.1" in its default configuration with four indices GS, SP, POGS, and OP.

p	**Added**	**Removed**	**Retained**	**Strategy**	**Time taken (sec)**
0.5	124924	124937	123319	SQL	240
				RDF	200
0.8	79605	79710	318149	SQL	200
				RDF	250
0.9	44629	44554	402748	SQL	170
				RDF	300

Table 1. Benchmark results.

In Table 1 the value *SQL* indiciates the RDB assisted approach, and *RDF* the specialized one. As can be seen from the table, the former approach - which reduces the updates to the triple store to a minimum - performs better than specialized version when there is sufficient overlap between O and N ($p = 0.8$ and $p = 0.9$) On the other hand, the smaller the overlaps the more the RDB becomes a bottleneck ($p = 0.5$). This is expected as in the worst case there is no overlap between O and N. In this situation the specialized approach would delete and reinsert triples directly. The RDB assisted approach would ultimatively do the same; however with the overhead of additionally reading from and writing to the `dbpedia_page` table.

4 Related Work, Future Work and Conclusion

In this paper we sketched four different approaches for managing triples in the context of the DBpedia live extraction: The first based on OWL 2 annotations, the second using domain specific queries, the third using individual graphs and the fourth being assisted by an RDB. Initially because of RDFs flexibility we were tempted to find a solution which operates on the triple store alone. In regard to the RDB assisted approach we were sceptical as it meant having to duplicate every single triple. The lesson learnt is that for application scenarios involving frequent minor updates of resources in a triple store, an RDB assisted approach may be advantageous - despite the implied data duplication.

Related Work Apart from the strategies described in this article, there are a number other ways to improve the performance of synchronising a knowledge

base to its original source: The first and most obvious challenge in research and practice is to further improve performance of triple stores, in particular for SPARUL queries. Although the Berlin SPARQL Benchmark[4] (BSBM) became a reference for measuring the query performance of SPARQL endpoints, up to now there is no such benchmark playing a comparable role for SPARUL. Another method is to avoid decoupling the original source and the generated knowledge base. For instance, the Triplify tool[2] is a thin layer above a relational database. An RDF representation is generated by SQL queries augmented with syntactic sugar. This lightweight integration does not require a synchonisation process (of course, it could be that a mirror of the original source needs to be kept in sync). However, it is usually preferable only for simple extraction processes, otherwise the burden to generate an RDF presentation in real time becomes computationally very expensive. Furthermore, complex transformations of the original source, as present in the DBpedia extraction framework, are difficult to handle.

Future Work For improving the performance of the DBpedia Navigator [7] we will integrate an adaptive SPARQL query cache [8] as a proxy layer on top of the DBpedia SPARQL Endpoint. This caching solution analyses the triple patterns of the SPARQL queries and stores them in combination with their result sets. To trigger the invalidation process of the cache proxy, all added and updated statements, which are committed to the RDF store, have to be committed to the cache proxy as well. The invalidation process of this cache proxy works very selectively and invalidates only those cache objects whose aggregated triple pattern matches the added or updated statements. However, an invalidation process is an expensive process and should only be triggered if necessary. The deployment of strategies presented in this paper are contributed to reduce the change sets of statements for the DBpedia update process. However they contribute as well to the cache proxy as the amount of statements that must be considered in the invalidation process is minimized.

References

1. Sören Auer, Chris Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference (ISWC)*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2008.
2. Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumueller. Triplify: light-weight linked data publication from relational databases. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 621–630. ACM, 2009.
3. Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.
4. Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

5. Sebastian Hellmann, Claus Stadler, Jens Lehmann, and Sören Auer. DBpedia live extraction. In *Proc. of 8th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, volume 5871 of *Lecture Notes in Computer Science*, pages 1209–1223, 2009.
6. Jens Lehmann, Chris Bizer, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Journal of Web Semantics*, 7(3):154–165, 2009.
7. Jens Lehmann and Sebastian Knappe. DBpedia navigator. Semantic Web Challenge, International Semantic Web Conference 2008, 2008.
8. Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the performance of semantic web applications with SPARQL query caching. In *Proceedings of 7th Extended Semantic Web Conference (ESWC 2010), 30 May – 3 June 2010, Heraklion, Greece*, 2010.
9. Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C Recommendation, 2008. http://www.w3.org/TR/rdf-sparql-query.
10. Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *16th international World Wide Web conference (WWW 2007)*, New York, NY, USA, 2007. ACM Press.