Large-scale RDF Dataset Slicing

Edgard Marx Saeedeh Shekarpour, Sören Auer, Axel-Cyrille Ngonga Ngomo
AKSW, Computer Science, University of Leipzig, Germany
http://aksw.org, {marx,shekarpour,auer,ngonga}@informatik.uni-leipzig.de

***Abstract*—In the last years an increasing number of structured data was published on the Web as Linked Open Data (LOD). Despite recent advances, consuming and using Linked Open Data within an organization is still a substantial challenge. Many of the LOD datasets are quite large and despite progress in RDF data management their loading and querying within a triple store is extremely time-consuming and resource-demanding. To overcome this consumption obstacle, we propose a process inspired by the classical Extract-Transform-Load (ETL) paradigm. In this article, we focus particularly on the selection and extraction steps of this process. We devise a fragment of SPARQL dubbed SliceSPARQL, which enables the selection of well-defined slices of datasets fulfilling typical information needs. SliceSPARQL supports graph patterns for which each connected subgraph pattern involves a maximum of one variable or IRI in its join conditions. This restriction guarantees the efficient processing of the query against a sequential dataset dump stream. As a result our evaluation shows that dataset slices can be generated an order of magnitude faster than by using the conventional approach of loading the whole dataset into a triple store and retrieving the slice by executing the query against the triple store's SPARQL endpoint.**

I. Introduction

In the last years an increasing number of structured data was published on the Web as Linked Open Data (LOD). Despite recent advances, consuming and using Linked Open Data within an organization is still a substantial challenge. Many of the LOD datasets are quite large and despite progress in RDF data management their loading and querying within a triple store is extremely time consuming and resource demanding. Examples of such datasets are *DBpedia* (version 3.8)[1] and *LinkedGeoData*[2], which encompass more than 1 billion triples each. Loading these datasets into a triple store requires substantial amounts of resources and time (e.g. 8 hours for DBpedia and 100 hours for LinkedGeoData on standard server hardware). Rapid prototyping, experimentation and agile development of semantic applications is currently effectively prevented this way. However, many users are not interested in the whole dataset, but in a very specific part of it. A search engine specialized on entertainment topics, for example, might aim to enrich its search results with facts on actors and movies from DBpedia. A location-based service might want to provide information on points-of-interest in the neighborhood of the users location from LinkedGeoData. In both scenarios, only a tiny fraction of the respective knowledge bases is required: From DBpedia, we need in the first case essentially all instances from the classes `Actor` (2,431 instances) and `Film` (71,715 instances). For the location-based service scenario, we can omit all nodes, ways and relations from LinkedGeoData, which do not belong to the class `point-of-interest` or any of its sub-classes. In that case, 98% of LinkedGeoData can be purged, thus lowering resource requirements and increasing query performance by several orders of magnitude. If we enable users to efficiently extract slices from knowledge bases, which comprise exactly the information they require, building special-purpose Semantic Web applications will become significantly more efficient.

[1]http://dbpedia.org
[2]http://linkedgeodata.org, version of May 3rd, 2013

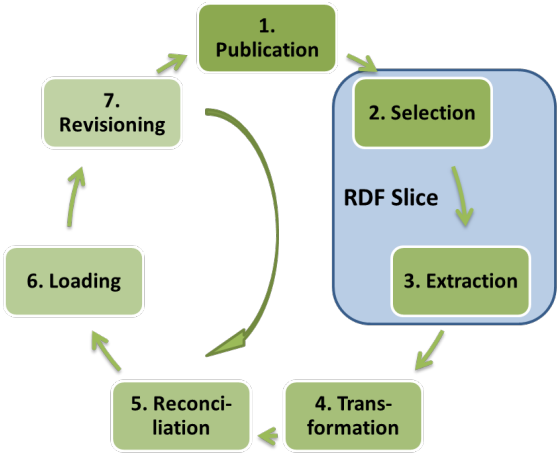


Fig. 1. Linked Open Data consumption process.

Figure 1 depicts a conceptual LOD consumption process. The process is inspired by the classical Extract-Transform-Load (ETL) process known from data warehousing. However, other than ETL the LOD consumption considers both new dataset versions being published *and* revisions being applied to internally used (parts of) these datasets. The steps are:

1) *Publication* is a prerequisite for the remaining consumption steps and comprises the publication of an RDF dataset by a data publisher, mostly as a dataset dump or SPARQL endpoint.
2) *Selection* comprises the definition and specification of a relevant fragment of a dataset, which is envisioned to be used internally by a consuming organization.
3) *Extraction* processes the dataset dump and extracts the relevant fragment.
4) *Transformation* comprises the mapping and mapping execution of the extracted data structure to match organization internal data structures.
5) *Reconciliation* applies revisions made by the organi-

zation to earlier versions of the dataset to the actual version.

6) *Loading* makes the dataset available for internal services and applications, for example, by means of a SPARQL endpoint.
7) *Revisioning* allows the organization to apply (manual) changes to the dataset, such as deleting instances changing properties etc. Revisions applied to a certain version of the dataset should be persistent and be automatically reapplied (after an update of the dataset in the respective reconciliation step.

In this article, we focus particularly on the selection and extraction steps. We devise a fragment of SPARQL dubbed SliceSPARQL, which enables the selection of well-defined slices of datasets fulfilling typical information needs. SliceSPARQL supports graph patterns for which each connected subgraph pattern involves a maximum of one variable or IRI in its join conditions. This restriction guarantees the efficient processing of the query against a sequential dataset dump stream. As a result our evaluation shows that dataset slices can be generated an order of magnitude faster than by using the conventional approach of loading the whole dataset into a triple store and retrieving the slice by executing the query against the triple store's SPARQL endpoint.

The remainder of the paper is organized as follows. Section II briefly introduces some important formalisms such as the RDF data model and SPARQL algebra. Section III discusses our approach for Linked Data graph selection and extraction. Section V presents a comprehensive field study comparing conventional and SliceSPARQL Linked Data selection and extraction using a testbed involving a variety of different selections and datasets. Section VI discusses related works. Finally, Section VII concludes with an outlook on future work.

II. Background

Commonly the selection of subsets of RDF is performed using the SPARQL query language[3]. The SPARQL RDF query language can be used to express queries across diverse data sources. It is composed by three main parts: 1. Query Forms, 2. WHERE clause as well as 3. Solution Sequence and Modifiers (SSM).

The *Query Forms* contains variables that will appear in a solution result. It can be used to select all or a subset of the variables bound in a pattern match. Query Forms are designed to form result sets or RDF graphs. There are the four different select query forms SELECT, CONSTRUCT, ASK and DESCRIBE. The SELECT query form is the most common one and is used to return rows of variable bindings. CONTRUCT allows to create a new RDF graph or modify the existing one through substituting variables in a graph templates for each solution. ASK returns a Boolean value indicating whether the graph contains a match or not. Finally, DESCRIBE is used to return all triples about the resources matching the query.

The *WHERE clause* is composed by a *Graph Pattern* and some constraints helpers such as *FILTER*. *OPTIONAL* was designed for situations where there is a necessity to select also some RDF term that is not bound in some BGP. Filters are used to restrict a set of matched RDF terms to a subset where the filter expression evaluates to TRUE. The triple patterns in a BGP could be or not connected by a join condition. BGPs are a composition of one or more triple patterns that contains only variables or both, variables and constants. By rule, a triple pattern cannot be composed only by constants. BGPs are used to select RDF terms from a certain data subgraph and are composed by one or more triple patterns which contains only variables or both, variables and constants. The selected RDF terms are those of the matching subgraphs that was mapped to variables. Please refer to Definition 2 for better understanding.

The use of query Forms and WHERE clauses generates an unordered set of solutions. *Solution Sequence and Modifiers (SSM)* can be applied to this set to generate another sequence or select a portion of the result set. The SSM is composed by six modifiers: ORDER, PROJECTION, DISTINCT, REDUCED, OFFSET and LIMIT. The subsequent formalization of RDF and core SPARQL is closely following [11].

Definition 1 (RDF definition)**.** *Assume there are pairwise disjoint infinite sets I, B, and L (IRIs, blank nodes, and RDF literals, respectively). A triple $(v_s, v_p, v_o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple. In this tuple, v_s is the subject, v_p the predicate and v_p the object. We set $T = I \cup B \cup L$ and call T's elements RDF terms.*

In the following, the same notion is applied to *triple patterns* and *triple maps*. An RDF *graph* is a set of RDF triples (also called RDF dataset, or simply a dataset). Additionally, we assume the existence of an infinite set V of variables with $V \cap T = \emptyset$. The W3C recommendation SPARQL is a query language for RDF. By using *graph patterns*, information can be retrieved from SPARQL-enabled RDF stores. This retrieved information can be further modified by a query's solution modifiers, such as sorting or ordering of the query result. Finally the presentation of the query result is determined by the *query type*, return either a set of triples, a table or a Boolean value. The graph pattern of a query is the base concept of SPARQL and as it defines the part of the RDF graph used for generating the query result, therefore *graph patterns* are the focus of this discussion. We use the same graph pattern syntax definition as [11].

Definition 2 (SPARQL Basic Graph Pattern syntax)**.** *The syntax of a SPARQL Basic Graph Pattern expression is defined recursively as follows:*

1) *A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern).*
2) *The expressions (P_1 AND P_2), (P_1 OPT P_2) and (P_1 UNION P_2) are graph patterns, if P_1 and P_2 are graph patterns.*

[3] http://www.w3.org/TR/sparql11-query

Category	Join Type	Graph Patterns
SS	**subject-subject**	$(s_1, p_1, o_1)(s_1, p_2, o_2)$
PP	**predicate-predicate**	$(s_1, p_1, o_1)(s_2, p_1, o_2)$
OO	**object-object**	$(s_1, p_1, o_1)(s_2, p_2, o_1)$
SO	**subject-object**	$(s_1, p_1, o_1)(s_2, p_2, s_1)$
SP	**subject-predicate**	$(s_1, p_1, o_1)(s_2, s_1, o_2)$
OP	**object-predicate**	$(s_1, p_1, o_1)(s_2, o_1, o_2)$

TABLE I
CATEGORIZATION OF JOIN TYPES IN GRAPH PATTERNS CONSISTING OF TWO TRIPLE PATTERNS.

3) *The expression (P FILTER R) is a graph pattern, if P is a graph pattern and R is a SPARQL constraint.*

SPARQL constraints are composed of functions and logical expressions, and are supposed to evaluate to Boolean values. Additionally, we assume that the query pattern is well-defined according to [11]. Table I categorizes the types of joins in graph patterns consisting of two triple patterns. For example, the join type SS means that the two triple patterns have the same subject, while SO means that the subject of the first triple pattern is the same as the object of the second triple pattern.

Although SPARQL allows a variety of types of selections, an empirical study over real queries [**?**] shows that the most frequent *Query Forms* executed against DBpedia and SWDF[4] is SELECT, comprising 96.9% and 99.7% respectively while ASK, CONSTRUCT and DESCRIBE are scarcely used. This study also states that the most common triple patterns found only have a variable at the object position (DBpedia 66.35%; SWDF 47.79%). The authors also conclude that most queries are simple, i.e., 66.41% of DBpedia queries and 97.25% of SWDF just contain a single triple pattern. Another important finding is that joins are typically of the types SS (∼60%), SO (∼35%) and OO (∼4.5%). The study also shows that most of queries (99.97%) have a star-shaped graph pattern, and the chains in 98% of the queries have length one, with the longest path having a length of five.

III. RDF DATA SLICING

The goal of our slicing approach is to compute portions of a given set of RDF streams that abide by a description provided in a restricted SPARQL vocabulary which we call SliceSPARQL. An overview of the approach is given in Figure 2. We base our slicing approach on matching triple patterns of SliceSPARQL queries sequentially against the data read from the dataset dump file.

Definition 3 (SliceSPARQL)**.** *SliceSPARQL is the fragment of SPARQL for which each connected subgraph pattern of the SPARQL graph pattern involves a maximum of one variable or IRI in its join conditions.*

The process of dataset slicing as depicted in Figure 2 comprises three passes. In the first pass, the SliceSPARQL query is analyzed in order to recognize the maximally connected subgraph patterns and an associated most restrictive triple pattern. Then, the most restrictive pattern is used to extract the matching join candidate triples from the dataset dump file. In the *second pass*, the datasets are processed again in order to verify which of the join candidates match the remaining triple patterns of the respective SliceSPARQL's maximally connected subgraph pattern.

Definition 4 (Most Restrictive Triple Patterns)**.** *For a given triple pattern t, the number of constants contained in t is denoted by t_c. The set of the most restrictive triple patterns of a SliceSPARQL query is the set of triple patterns having maximum t_c.*

The type of joins in graph patterns consisting of two triple patterns can be categorized in six categories. Table I shows the list of all possible join types.

Definition 5 (Set of join candidates)**.** *A graph pattern p matching the triple t is denoted by $p(t)$. Consider all maximally connected subgraph patterns P of a SliceSPARQL query with respect to the join position. For a given graph pattern $p \in P$, the set of the join candidates is the set of all RDF terms in the join position of triples matching p. This set is denoted by C_p and formally is defined as:*

$$C_p = \{RDFTerm(t) | t \in D \wedge p(t) \wedge p \in P\}$$

Considering the set of all patterns P, the set of join candidate C is the intersection of all C_p.

$$C = \bigcap_{\forall p \in P} C_p$$

Finally, in the *third pass*, we process the dataset once more and select all triples containing an RDF term which matches all patterns in SliceSPARQL.

The two final passes are omitted in any of the following situations: 1. All triple patterns are disjoint; 2. The dataset is segmented by subject and the graph pattern contains only SS-joins; 3. The dataset is sorted and the graph pattern contains only SS, SO or SP join conditions. The first pass can be also omitted if the triple patterns do not contain any variables. In this (rather rare) case the constants used for joining are the join candidates.

We decided to perform the extraction in the three stages for efficiency reasons. Note that the data could be extracted in just a single pass, but doing that could take as long as loading the data into a triple store. The last pass could also be omitted if we select all possible triples already in the second pass. This would save a little bit space once possible join candidates were selected in the first pass, but most of the candidates would not fulfill the triple patterns. For example, in case that we aim to create a slice with data about cities from New York state all world cities would be candidates, but just a small portion are actually relevant. A tree-pass process is better suited, since it allows to determine the desired slice more precisely in the first two passes and to extract the required information in a targeted way in the third pass.

Example 1. *Let us look at the selection of drugs from the following example dataset composed out of triples extracted*

[4]http://data.semanticweb.org

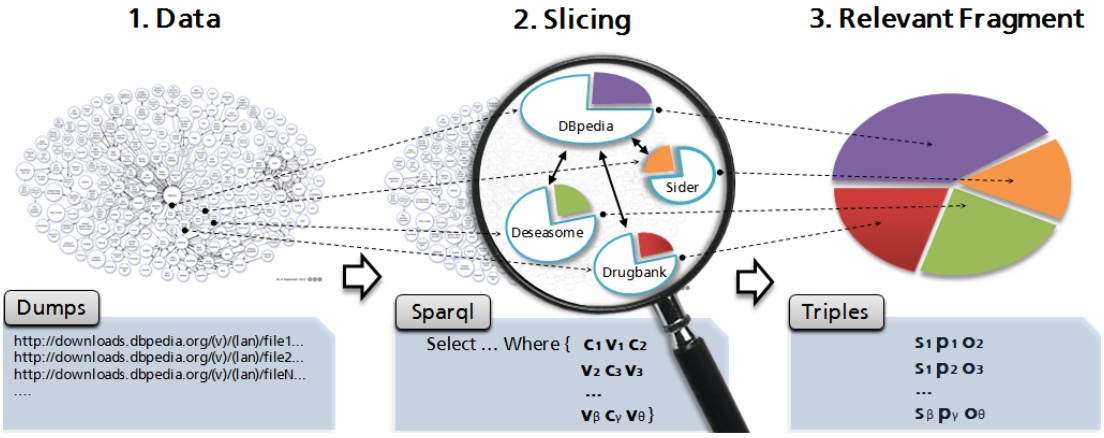


Fig. 2. Overview of the RDF data slicing approach: (1) dataset dumps are accessed on the Web, (2) the SliceSPARQL patterns are evaluated against the sequential dataset dump stream, (3) relevant fragments from different datasets can be combined into an application specific knowledge base.

from DBpedia:

```
1 dbr:Aspirin     a                dbo:Drug .
2 dbr:Aspirin     rdfs:label       "Aspirin"@en .
3 dbr:California  rdf:type         dbo:Place .
4 dbr:California  dbo:largestCity  dbpedia:Los_Angeles .
```

The following query can be used to select all instances of the DBpedia class `Drug` *and all resources referring to them:*

```
1 SELECT * WHERE { {
2     ?s    a     dbo:Drug .
3     ?s    ?p    ?o.
4   } UNION {
5     ?s1   a     dbo:Drug .
6     ?o1   ?p1   ?s1. }
7 } }
```

In the first pass the SPARQL query is split into two BGPs (lines 2-3 and 5-6) each containing two triple patterns. The first one has an SS join condition using the variable ?s (SSv). The second one has an SO join condition using the variable ?s1 (SOv). The more restrictive triple pattern is then chosen for each of them. In both cases, the first triple pattern (i.e. line 2 and 5) is the more restrictive one. In the first pass, all triples in the dataset matching one of these restrictive patterns are then selected as join candidates. Note that the triple in line 1 of the given dataset matches both more restrictive patterns:

In the second pass the RDF terms of the join candidates used in a join condition (variables or constants) are then used to evaluate which of the join candidates matches all other triple patterns in each of the BGPs. In our example, the RDF term of our candidate fully matches only one BGP. The RDF term `dbpedia:Aspirin` *matches both triples* `?s a dbo:Drug` *and* `?s ?p ?o`.

In the last and final pass all triples in which the RDF term in the join condition fulfills all triple patterns of some BGP are extracted, i.e.:

```
1 dbr:Aspirin  a           dbo:Drug
2 dbr:Aspirin  rdfs:label  "Aspirin"@en.
```

IV. Complexity Analysis

In order to better understand the time complexity of the approach we look at all types of join in Table I. An extraction can be performed in one, two or three passes depending on the SliceSPARQL pattern and the representation of the source dataset. The complexity of each of this methods is shown in Table II and described in the sequel. Please note that the join candidates are stored in a B-tree structure.

Case 1. *Selection in unsorted datasets with any type of joins.*

Case 1.1. *The time complexity of the process where the join RDF term is a variable (e.g. Q1, Q3, Q4, Q5 in Table IV) is* $\mathrm{O}(n \log n)$. *We call this the* generic method *because it can be used for unsorted datasets with any type of join. The complexity of the generic process is as follows:*

The first pass *comprises the (1) selection of a most restrictive pattern and the (2) selection of join candidates. The selection of a most restrictive pattern is carried out by retrieving the triple patterns in the graph pattern with the least variables. With* t *we refer to the number of triple patterns in the graph pattern and as we discussed in Section II* t *is small (i.e.,* $t << n$*). After selecting the most restrictive triple pattern, the selection of the join candidates is performed by reading the dataset sequentially. Let* n *be the number of triples in the dataset. Each triple that matches the most restrictive triple pattern requires an insertion in a B-tree structure, which can be performed in* $\mathrm{O}(\log m)$*where* m *is the number of elements in the B-tree, i.e. the size of the join candidate dataset. Consequently, the first stage can be described with the follow formula and complexity* $t + \sum_{m=1}^{n} log m \approx \mathrm{O}(n \log n)$.

The second pass *consists of reading the target dataset and checking each triple for whether it matches some of the triple patterns. This can be done in* $O(t)$*. If the triple matches some triple pattern, then an update is necessary. Such an update can be carried out in* $O(\log m)$*. Taking the size* n *of the dataset into account, the second stage can be carried out in* $\sum_{m=1}^{n} t \times \log m = t \times \mathrm{O}(n \log n)$.

The third pass *performs the actual extraction. For each*

triple in the dataset it is checked if the triple contains an RDF term from the join candidates matches all triple patterns from some of the graph patterns. Searching in the stored join candidates can be performed in $\log m$*. Hence, the time complexity of this pass is* $\sum_{m=1}^{n} \log m = \mathrm{O}(n \log n)$.

The final complexity of the generic method is the sum of the complexity of each stage, i.e. $t \times O(n \log n) \approx \mathrm{O}(n \log n)$.

Case 1.2. *The time complexity in case of a constant join RDF term (e.g. Q2 in Table IV) is* $\mathrm{O}(n)$.

Since the join constant in the graph pattern is already defined, there is no necessity to perform pass one. The slicing takes two passes. (1) First checking if all triple patterns appear in the dataset and a second one selecting the matching triples. As the both passes require $\mathrm{O}(n)$ *the final complexity is* $\mathrm{O}(n)$.

Case 2. *The extraction from sorted datasets using SO and SP triple patterns where the most restrictive triple pattern has is joined via the object or predicate respectively can be performed in one pass.*

Case 2.1. *The RDF join term is a constant and the most restrictive triple pattern contains a constant as subject (e.g. Q7 in Table IV).*

In the case of the RDF join term being a constant, the extraction can be performed using two binary searches, one to find the target subject and another one to find the target object ($\sum_{1}^{2} \log n = \mathrm{O}(\log n)$*).*

Case 2.2. *The join RDF term is a variable (SOv) (Q8 in Table IV).*

One pass through the dataset leads to all triples matching the triple pattern having the join RDF term as object or predicate. However, after finding these triples a binary search then finds the corresponding triple pair. Consequently, the time complexity is $\sum_{n=1}^{n} logn = \mathrm{O}(n \log n)$*. Nevertheless, despite* Case 1 *and* Case 2 *have the same complexity they require a different number of passes (three for* Case 1 *and one for* Case 2*).*

Some other triple patterns can also be profiled in one pass e.g. SO where the most restrictive pattern has the join RDF term as subject. In that case, choosing the less restrictive triple pattern can also lead to extracting the desired data in one pass. As the process with tree passes selects the more restrictive triple pattern, we can say that the first approach requires $3 * mlogn$ *where* $m \leq n$*. For typical RDF term distributions* $3 * m \leq n$*. For instance, 99% of properties instances in DBpedia belongs to only 10% of the properties. A similar distribution can be found in the generic-infobox, mapping-base infobox and pagelinks datasets in terms of node indegree [3].*

Case 3. *Extraction from sorted or subject segmented dataset dumps.*

Case 3.1. *The complexity with a variable as join RDF term (*SSv*) and the dataset segmented by subject (Q2 in Table IV) is* $\mathrm{O}(n)$.

Since the number of triples for a particular subject segment is small enough to fit into memory, there is no need for more than one pass. Loading each segment of the instance in memory and checking the patterns contained in SS can be performed in one pass. The time complexity in this case is $\sum_{m=1}^{n} 5 = \mathrm{O}(n)$.

Category	Unsorted	Instance Segmented	Sorted
SSv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
SSc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(\log n)$
PPv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$
PPc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
OOv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$
OOc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
SOv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$
SOc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(\log n)$
SPv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$
SPc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(\log n)$
OPv	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$	$\mathrm{O}(n \log n)$
OPc	$\mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$

TABLE II
TIME COMPLEXITY FROM DIFFERENT JOIN TYPES.

Case 3.2. *The complexity with a constant join RDF term (*SSc*) and the dataset being sorted by subject (Q6 in Table IV) is* $\mathrm{O}(\log n)$.

The extraction can be performed with one binary search to find the target subject, i.e. $\sum_{1}^{2} logn = \mathrm{O}(\log n)$.

V. EVALUATION

The goal of our evaluation was to determine: (1) How efficient is the slicing approach for various queries? (2) How does the slicing scale for datasets of different size? (3) How does our approach compare to the traditional approach (i.e. loading complete dumps into the triplestore and extracting by querying). All files generated during the evaluation as well as logs and tables are available online[5].

A. Experimental Setup

We used four interlinked datasets, i.e. Drugbank, Sider, Diseasome and DBpedia Version 3.8 for the evaluation. Table III shows the sizes of these datasets. DBpedia-slice refers to a version of DBpedia comprising all DBpedia datasets excluding the *page links undirected* dataset. We selected Drugbank, Sider, Diseasome and DBpedia because they are a fragment of the well interlinked part of Linked Open Data. Especially, DBpedia is an ideal case with respect to size as it is very large. Two students expert to SPARQL and schema of the underlying datasets created eight SPARQL queries shown in Table III. The provided queries take into account the two most frequent type of join (i.e. `SS` and `SO`) as well as different time complexities. The type of the joins of the associated BGPs and the related dataset are shown. For instance, the query Q1 running on DBpedia contains eight triple patterns which can be divided to four disjoint BGPs having as join type either subject-subject (SS) or subject-object (SO). We do not take queries into account containing patterns with join types SP, OP and OO, since they seem to be very rare in real SPARQL

[5]http://aksw.org/projects/RDFSlice

Dataset	Size	Triples	Entities
DBpedia	52 GB	283,928,812	22,857,222
DBpedia-slice	29 GB	125,554,842	13,410,215
DrugBank	98 MB	517,150	19,696
Sider	16 MB	91,569	2,674
Diseasome	12 MB	72,463	8,152

TABLE III
EVALUATION DATASETS STATISTICS.

queries (5%) [**?**] and have the same complexity as in the SO case. We measured the performance of our slicing approach in terms of `runtime` and `memory consumption`. All experiments were carried out on a Windows 7 machine with an Intel Core M 620 processor and 6GB of RAM and a 230GB SSD.

B. Implementation

To profile the implementation of the approach we use a set of applications developed in Java. To ease the file management we create three applications for the (1) download, (2) sorting and (3) decompression of files respectively. All the tests were profiled using *Virtuoso Server*[6], for this propose we implemented a (4) Virtuoso utility application. The Virtuoso application was built using the Virtuoso JDBC Driver[7]. The rationale of using the Virtuoso JDBC Driver was to communicate directly with the Virtuoso instance without the overhead generated by other methods as HTTP clients. The Virtuoso utility allows dropping graphs, loading dump files and profiling queries natively as ISQL client. To load the dump files into Virtuoso the function `ld_dir()`[8] was used in order to speed up the loading.

To profile the slicing approach we created another application (5). The slicing approach was developed and main tested with N-Triple files and is due to the SliceSPARQL fragment not compatible with SPARQL 1.1[9].

The storage of join candidates was achieved by using a custom file storage which relies B-trees. The structure of the each stored join candidate is composed by: *Join Term*, *GP position*, *Join Condition* and *Match Vector*. The *GP position* is the position where the GP appears in query e.g. the first GP is zero, second GP is one and the n-th position is n minus one. The *join term* is the term used to join triple patterns in a GP. The *join condition* is one of the join conditions listed in Table I. The *match vector* is a string with length n where n is the number of triple patterns in BGP. Each position of the *match vector* represents a triple pattern, containing one if the triple pattern matches or zero if not.

C. Results

Table V-C shows the `runtime` versus `memory` (being either explored or used) for four parameters of the entire slicing process. These four parameters are:

[6]http://virtuoso.openlinksw.com/
[7]http://docs.openlinksw.com/virtuoso/VirtuosoDriverJDBC.html
[8]http://docs.openlinksw.com/virtuoso/fn_ld_dir.html
[9]http://www.w3.org/TR/sparql11-query/

Query	Triple Patterns	Join	Dataset

Q1	`?s a dbo:Drug.`
	`?s ?p ?o.`
	`?s1 a dbo:Drug.`
	`?o1 ?p1 ?s1.`
	`?s2 a dbo:Disease.`
	`?s2 ?p2 ?o2.`
	`?s3 a dbo:Disease.`
	`?o3 ?p3 ?s3.`
		SS+SO	**DBpedia**

Q2	`?s a dbo:Drug.`
	`?s ?p ?o.`
	`?s1 a dbo:Disease.`
	`?s1 ?p1 ?o1.`
		SS+SS	**DBpedia**

Q3	`?s a diseasome:diseases.`
	`?s ?p ?o.`
	`?s1 a diseasome:diseases.`
	`?s1 ?p1 ?o1.`
		SS+SO	**Diseasome**

Q4	`?s a DrugBank:Drugs.`
	`?s ?p ?o.`
	`?s1 a DrugBank:Drugs.`
	`?s1 ?p1 ?o1.`
		SS+SO	**DrugBank**

Q5	`?s a Sider:Drugs.`
	`?s ?p ?o.`
	`?s1 a Sider:Drugs.`
	`?s1 ?p1 ?o1.`
		SS+SO	**Sider**

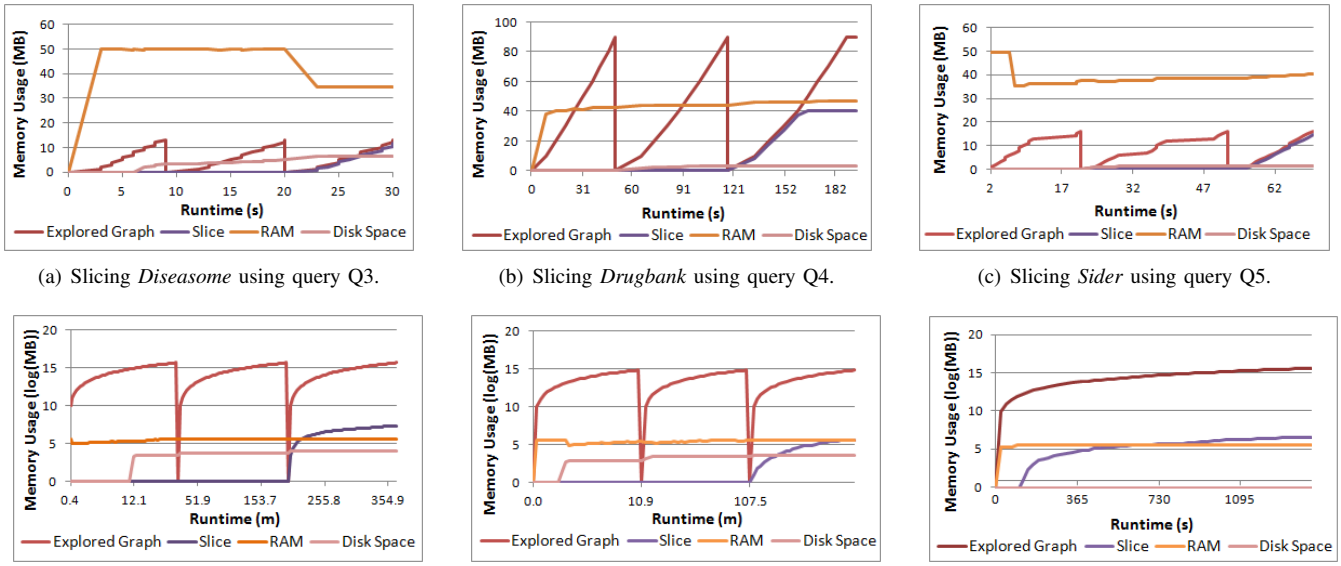
Q6	`dbr:Cladribine dbo:iupacName ?o.` `dbr:Cladribine ?p1 ?o1.`	**SS**	**DBpedia**
Q7	`dbr:Delirium dbo:wikiPageWikiLink ?o.` `?o ?p ?q.`	**SO**	**DBpedia**
Q8	`?s1 dbo:lastWin ?o.` `?o ?p ?o1.`	**SO**	**DBpedia**

TABLE IV
EVALUATION QUERIES.

1) *Explored graph:* The size of the associated graph of the underlying file which is being explored.
2) *RAM:* The size of RAM memory which is occupied.
3) *Slice:* The size of the generated slice from data.
4) *Disk space:* The size of the disk used to run the application (excluding the slice size).

These parameters are recorded after reading and processing 1 MB and 1 GB for small and large files respectively. Accordingly, the diagrams a-f in Table V-C represent the above parameters for the Diseasome, Drugbank, Sider, DBpedia and DBpedia-slice datasets, respectively. A general behavior which can be observed in all diagrams is: During the slicing process, the associated dataset dump is analyzed maximum three times. Therefore, the diagram of the explored graph shows three hops except of the diagram f. That is due to DBpedia files being segmented by subject and the the query Q2 containing only subject-subject joins. Thus, one exploration suffices for slicing. With respect to the slice graph, since only in the last pass matches are being found and stored, the size is increasing. Monitoring the last two graphs reveal that a maximum 50 MB of RAM and a very low amount of disk space are occupied. More interestingly, a short while after starting the process, RAM and disk usage are remaining fixed, since in the entire process only join candidates are required to be stored.

Table V compares total runtime requiring for slicing DBpedia on data being available in triple store and files. Since an advantage of our approach is the extraction directly from files, the deploying time (i.e. loading data into the triple store) was taken into account. From the five queries (i.e Q1, Q2, Q6, Q7, Q8) used in this experiment, four queries (i.e. Q2, Q6, Q7, Q8) perform an order of magnitude faster in comparison to the total time computed over triple store (800% faster).



(a) Slicing *Diseasome* using query Q3. (b) Slicing *Drugbank* using query Q4. (c) Slicing *Sider* using query Q5.

(d) Slicing *DBpedia* using query Q1. (e) Slicing *DBpedia-slice* using the Q1. (f) Slicing *DBpedia* using query Q2.

Fig. 3. Slicing different datasets with different types of queries.

Approach	Query	Sort	Deploy	Extraction	Total	Gain
Triplestore	Q1	-	446.4	1.8	448.2	21.4%
File		-	-	369	369	
Triplestore	Q2	-	446.4	2.1	448.5	1,916.6%
File		-	-	23.4	23.4	
Triplestore	Q6	-	446.4	0.0	446.4	1,162.5%
File		38.4	-	0.0	38.4	
Triplestore	Q7	-	446.4	0.0	446.4	1,162.5%
File		38.4	-	0.0	38.4	
Triplestore	Q8	-	446.4	2.95	449.35	800%
File		38.4	-	17.38	55.78	

TABLE V
TOTAL RUNTIME IN MINUTES REQUIRING FOR SLICING DBPEDIA AGAINST FIVE DIFFERENT QUERIES ON DATA BEING AVAILABLE IN TRIPLE STORE AND FILES.

Unlike the other queries, the total runtime of Q1 is very high (still 18% faster than the triple store approach). That is due to the fact that the query is applied on unsorted files and contains triple patterns of the join type SO. Furthermore, Table VI presents the extraction time in the unsorted version of underlying datasets using the queries i.e. `Q1, Q3, Q4 and Q5`. Although the queries `Q1, Q3 and Q4` have both the `SS` and `SO` join type, the selection time is considerable low because due to the small size of the underlying datasets. In case of DBpedia and DBpedia-slice, the effect of file size is more tangible (45% decrease in the file size leads to a 45% decrease in the extraction time).

VI. Related Work

To the best of our knowledge this is the first work specifically targeting RDF data slicing. However, it is related to approaches in the three areas of RDF data a) crawling, b) streaming and c) replication, which we briefly discuss in the following subsections.

a) Crawling: RDF data crawlers harvest and index RDF content from the Web of Data and Documents. *MultiCrawler* [6] allows to extract information not only from HTML documents but also from structured data on the Web. It focuses on locating relevant links to content in order to extract data. MultiCrawler explores the use of not only well-known text indexing but also structured data indexing, by converting all non-structured information found into corresponding structured data. This approach is useful for indexing and finding documents with relevant content, but not to extract fragments of larger datasets. *LDSpider* [8] is a lightweight LOD crawler for integration into Semantic Web applications. LDSpider can be used to traverse LOD content and deliver the extracted data via an API to the application through listeners. The application itself has to select the relevant content. One of LDSpider's main features is the capability to process a variety of Web data formats including Turtle, RDF/XML, Notion 3 and RDFa employing different crawler strategies as breadth-first and load-balancing. *Semantic Web Client library* [7] was developed to crawl the Web of Data in order to facilitate query

Dataset	Query	Extraction
DBpedia	Q1	369
DBpedia-slice	Q1	202.2
Drugbank	Q4	3.2
Sider	Q5	1.18
Diseasome	Q3	0.5

TABLE VI
EXTRACTION TIME IN MINUTES FOR UNSORTED DATASETS OF DIFFERENT SIZES.

answering. The rationale is to discover relevant information in the crawled structured data from different sources during query execution time. Similar as crawlers RDFSlice can be used to retrieve and extract relevant RDF data from the Web. However, RDFSlice focuses on structured data in large files, it does not traverse links and uses specific selection criteria (i.e. SliceSPARQL graph patterns) instead of heuristics.

b) Streaming: Linked Streaming Data (LSD) is an extension of the RDF data model to support the representation of stream data generated from sensors and social network. They are designed for continuous query data with high rate of change, e. g. once per second. There are already many proposed approaches for RDF Streaming as CQELS [9], Streaming SPARQL [4], C-SPARQL [2], Sparql $stream$ [5] and EP-SPARQL [1]. These approaches work in a similar fashion: In LSD approaches, the user defines a time window in which the data will be selected. When the window expires, the data collected is then used to query. The approaches could also use some Linked Data to enrich the information or easily the selection. Differently from the SPARQL Streaming approaches, the Slicing is not designed for querying streaming data. Rather, we aim to extract relevant fragments from large files in the distributed static RDF LOD. Nevertheless, Slicing could also be used as a prefilter in RDF streaming data, helping to select a relevant subset of the data during the time window.

c) Replication: Although data replication is a well known study in database field to improve performance and availability, there is still a lack in methods and approaches into concerns Linked Data. We also observed that, most of the existing approaches focus in triple stores instead of dump files, their work also are simplified by focusing in managing existing instances rather than create new ones. Some related problems are data selection and synchronization. RDFSync [12] profile data replication between triple stores by decomposing the graphs into smaller Minimum Self-Contained Graphs (MSGs) and comparing their hashes. However, RDFSync does not take into account relevant data replication. Another related work is sparqlPuSH [10]. SparqlPuSH works as a notification message system, notifying the servers registered as listeners about changes into the triple store. SparqlPush could also provide relevant data replication with some restriction by the use of subsection notification of PubSubHubbub protocol[10].

VII. Conclusions

In this article we presented an approach facilitating Linked Data consumption by effectively selecting relevant parts and efficiently extracting these from very large RDF datasets. We deem this to be a major step towards simplifying the consumption of large RDF datasets.

We see this work as the first step in a larger research agenda to dramatically improve Linked Data consumption. The presented approach focuses only on two out of six stages of the consumption process. In future, we aim to develop and integrate support for subsequent stages such as transformation, reconciliation, and revisioning. We envision that organizations will thus be empowered to seamlessly integrate LOD data into their internal processes and applications. A particular challenge is the mapping of the LOD data to existing internal information structures and the establishment of a co-evolution between private and public data involving continuous update propagation from LOD sources while preserving revisions applied to prior versions of these datasets.

[10]http://code.google.com/p/pubsubhubbub/

Acknowledgment

This work was partly supported by CNPq, under the program Ciências Sem Fronteiras, by Intituto de Desenvolvimento e Pesquisa Albert Schirmer (CNPJ 14.120.192/0001-84) and by a grant from the European Union's 7th Framework Programme provided for the project LOD2 (GA no. 257943).

References

[1] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[2] Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *CoRR*, abs/1103.5043, 2011.

[3] Davide Francesco Barbieri, Michael Grossniklaus, and Milano Dipartimento. An Execution Environment for C-SPARQL Queries.

[4] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semant.*, 7(3):154–165, September 2009.

[5] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *ESWC2008*, pages 448–462. Springer-Verlag, 2008.

[6] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, ISWC'10, pages 96–111, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Andreas Harth and Stefan Decker. MultiCrawler : A Pipelined Architecture for Crawling and Indexing Semantic Web Data. pages 258–271, 2006.

[8] Olaf Hartig, Christian Bizer, and Johann-christoph Freytag. Executing SPARQL Queries over the Web of Linked Data.

[9] Robert Isele, Christian Bizer, and Andreas Harth. LDSpider : An open-source crawling framework for the Web of Linked Data. pages 6–9.

[10] Danh Le-phuoc, Minh Dao-tran, and Josiane Xavier Parreira. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. 1380(24761), 2007.

[11] Alexandre Passant and Pablo Mendes. sparqlPuSH: Proactive notification of data updates in RDF stores using PubSubHubbub. volume 699 of *CEUR Workshop Proceedings ISSN 1613-0073*, February 2010.

[12] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[13] Giovanni Tummarello, Christian Morbidoni, Reto Bachmann-Gmür, and Orri Erling. Rdfsync: Efficient remote synchronization of rdf models. pages 537–551. 2008.