Torpedo: Improving the State-of-the-Art RDF Dataset Slicing

Conference Paper · March 2017

CITATIONS
0

READS
116

11 authors, including:

Albert Weichselbraun
Hochschule für Technik und Wirtschaft in Chur
87 PUBLICATIONS **531** CITATIONS
SEE PROFILE

Jens Lehmann
University of Bonn
154 PUBLICATIONS **4,833** CITATIONS
SEE PROFILE

Sören Auer
University of Bonn
318 PUBLICATIONS **5,891** CITATIONS
SEE PROFILE

Adrian M.P. Brasoveanu
MODUL University Vienna
15 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Torpedo: Improving the State-of-the-Art RDF Dataset Slicing

Edgard Marx*, Saeedeh Shekarpour*§, Tommaso Soru*, Adrian M.P. Braşoveanu†, Muhammad Saleem*,
Ciro Baron*, Albert Weichselbraun‡, Jens Lehmann*, Axel-Cyrille Ngonga Ngomo*, and Sören Auer*¶
*AKSW, University of Leipzig, Germany
{marx,shekarpour,tsoru,saleem,cbaron,lehmann,ngonga}@informatik.uni-leipzig.de
http://aksw.org
†MODUL University Vienna, Vienna, Austria, adrian.brasoveanu@modul.ac.at
‡SII, University of Applied Sciences, Chur, Switzerland, albert.weichselbraun@htwchur.ch
§Knoesis Research Center, USA
¶EIS, University of Bonn, Germany

***Abstract*—Over the last years, the amount of data published as Linked Data on the Web has grown enormously. In spite of the high availability of Linked Data, organizations still encounter an accessibility challenge while consuming it. This is mostly due to the large size of some of the datasets published as Linked Data. The core observation behind this work is that a subset of these datasets suffices to address the needs of most organizations. In this paper, we introduce Torpedo, an approach for efficiently selecting and extracting relevant subsets from RDF datasets. In particular, Torpedo adds optimization techniques to reduce seek operations costs as well as the support of multi-join graph patterns and SPARQL FILTERs that enable to perform a more granular data selection. We compare the performance of our approach with existing solutions on nine different queries against four datasets. Our results show that our approach is highly scalable and is up to 26% faster than the current state-of-the-art RDF dataset slicing approach.**

I. Introduction

Over the last years, the amount of structured data which has been published on the Web as Linked Open Data (LOD) has grown enormously. Currently, it comprises approximately 150 billion triples from approximately 10,000 datasets.[1] In spite of the high availability of data, organizations still encounter an accessibility challenge while consuming Linked Open Data. RDF datasets are mostly accessible via either SPARQL endpoints or RDF data dumps. The experimental study in [2] (in which 427 public endpoints were examined) revealed that around only one-third of the public endpoints have an availability rate of more than 99%. Therefore, public endpoints are not a reliable option for accessing RDF data. Another option, i.e., using dumps of LOD datasets is also problematic. Since many of the LOD datasets are very large, both loading and querying them via a triple store is extremely time-consuming and resource-demanding. For example, *DBpedia*[2] and *LinkedGeoData*[3] encompass more than 1 billion triples each. The loading time amounts to approximately 8 hours for DBpedia and 100 hours for LinkedGeoData on state-of-the-art server hardware and triple stores.

The main observation of this work is that *commonly, organizations, as well as ordinary users, may not be interested in an entire dataset*; in many cases, they rather require a very specific fragment of these datasets. For instance, for a consumer with an interest in entertainment topics, a fragment of DBpedia containing facts about, e.g., movies (class `Film`, 71,715 instances) and actors (class `Actor`, 2,431 instances) is sufficient. Overall, these classes account for less than 3% of the resources in DBpedia. Another example is providing users with points-of-interest information from the LinkedGeoData dataset starting from the users' location. In this case, we can omit all nodes and relations which are not of the type `Point_of_interest` or any of its sub-classes; thus, 98% of triples in the knowledge base can be purged. In both scenarios, only a small fraction of the underlying knowledge base is sufficient for a particular use case. The intuition behind dataset slicing is that extracting the relevant fragments of large datasets in place is more efficient than downloading, indexing and extracting the same fragment via SPARQL queries issued against triple stores. Although there are many existing approaches for Linked Streaming Data (LSD) (CQELS [13], Streaming SPARQL [5], C-SPARQL [3], SPARQL *stream* [6] and EP-SPARQL [1]), all of them are designed for continuous data streaming with high change rate, e.g., once per second. RDF dataset slice does not address the problem of continuous data streaming. Rather, it focuses on extracting relevant fragments from large files in the distributed static RDF-based LOD. Naturally, this significantly increases query performance since irrelevant but potentially very large parts of a dataset are discarded. As the extracted slices include only the required amount of information, closed-domain Semantic Web applications (i.e., applications with a specific topic) can perform more efficiently.

The previously introduced concept of RDF dataset slice [14], [15] focuses particularly on both the selection and extraction steps of the Linked Open Data consumption process. These steps are essential to reduce space and time

[1]Data gathered on July 17th, 2016; http://stats.lod2.eu/.
[2]Version 2015 can be found at http://dbpedia.org.
[3]Version of January, 2016; http://linkedgeodata.org.

complexity in the whole process since the retrieved fragment is a subset (i.e., a *slice*) of the original dataset. In this paper, we propose a new approach for slicing RDF datasets called Torpedo. It consists into devising the fragment of SPARQL dubbed eSliceSPARQL, which enables a more granular selection of slices of datasets fulfilling typical information needs. eSliceSPARQL is an extension of the previous introduced SliceSPARQL [14], [15] and supports multi-join graph patterns for which each connected sub-graph 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, Torpedo can perform a more granular data selection than the previous introduced RDF dataset slice methods [14], [15]. Our main contributions are as follows:

- We extend previous introduced *SliceSPARQL* operator [14], [15] to the *eSliceSPARQL*, which supports multi-join type queries and SPARQL FILTER operator.
- We introduce an optimization approach for reducing RDF dataset slice disk seek operations.
- We compare *SliceSPARQL* with *eSliceSPARQL* in an extensive evaluation using DBpedia 2014[4] and show that it is around 26% faster than previous introduced RDF dataset slicing methods [14], [15].
- We compare Torpedo with different RDF data streaming approaches and show that it is faster and more scalable.

The rest of this paper is organized as follows. Section III presents our novel RDF slice approach. Section IV presents a comprehensive evaluation of our approach with the state-of-the-art with a focus on runtime performance. Section V given an overview of related works. Finally, we close with the conclusion in Section VI.

II. Background

RDF dataset slicing approaches are designed to extract portions of a given set of RDF streams that abide by a description provided in a restricted SPARQL vocabulary which we call SliceSPARQL [14], [15]. It works in a stream fashion, matching triple patterns of SliceSPARQL queries sequentially against the data processed from the dataset dump file.

```
1 ?s1 a dbo:Drug .
2 ?s1 owl:sameAs ?o1 .
```

Listing 1. An example of an SliceSPARQL query using subject (?s1). The triple in line 1 shares one variable (?s1) with triple in line 2.

The process of dataset slicing comprises three stages [14], [15]. In the first stage, the most restrictive triple pattern is extracted from the SliceSPARQL query and used to extract the matching join-candidates from the dataset. The SliceSPARQL is closed by Marx et al. [14], [15] as follows:

Definition 1 (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 most restrictive triple pattern is the one that contains more constants (i.e `?s1 a dbo:Drug .` in Listing 1). The most restrictive triple pattern is closed by Marx et al. [14], [15] as follows:

Definition 2 (Most Restrictive Triple Pattern): 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 T_r is the set of triple patterns having maximum t_c.

In the *second stage*, SliceSPARQL process the datasets again in order to verify which of the join candidates match the remaining triple patterns of the BGP (i.e. `?s1 owl:sameAs ?o1 .` in Listing 1). The join candidates is formally closed by Marx et al. [14], [15] as follows:

Definition 3 (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.

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

The third and final stage consist in selecting all triples containing an RDF term which matches all patterns in SliceSPARQL.

Please refer to previous works [14], [15] for a more extensive discussion and formalization of RDF dataset slicing approach as well as all possible join types and their respective time complexity.

III. Approach

In this work, we extend the SliceSPARQL to eSliceSPARQL by adding support for `SPARQL FILTER`s and multi-join graph patterns. We also optimize disk seek operations by adding an in-memory layer. As shown in [14], [15], the type of joins in graph patterns consisting of two triple patterns can be classified into six categories. The eSliceSPARQL operator is introduced below.

Definition 4 (eSliceSPARQL): eSliceSPARQL is the fragment of SPARQL for which each pair of connected *triple patterns* shares at most of one variable or IRI with another *triple pattern* (Listing 2).

Notice that, different from SliceSPARQL that deal with a single variable–e.g. `?s1` in Listing 1, the operator eSliceSPARQL can deal with more than one variable (`?s4` and `?04` in Listing 2) in the BGP join condition.

```
1 ?s4 a dbo:Drug .
2 ?s4 owl:sameAs ?o4 .
3 ?o4 ?p5 ?o5 .
```

Listing 2. Listing a multi-join BGP of Query 9.

[4]http://wiki.dbpedia.org/

A. Supporting SPARQL FILTERs

Providing good filter methods is important in order to support good dataset slice granularity. The state-of-the-art approch for RDF dataset slicing [14], [15] do not support the use of SPARQL `FILTER`s, which imposes several limitations. The proposed approach (Torpedo) is capable to evaluate the SPARQL `FILTER` as a simple or composed Boolean function,[5] when the evaluated function in the SPARQL `FILTER` involves a constant. Simple constant filters can be translated to triple patterns, i.e., the simple BGP '`?s ?p ?o. Filter(?s = <dbpedia:Ayton_Senna>)`' can be simplified to '`<dbpedia:Ayton_Senna> ?p ?o.`'. On the other hand, variables can lead to a significant increase of candidates. For example, the selection of everyone that was born and died in a different place.

```
1 ?s dbo:bornPlace ?bornPlace .
2 ?s dbo:deathPlace ?deathPlace .
3 FILTER (?bornPlace != ?deathPlace)
```

Listing 3. Listing a Filter using variables as restriction.

Another possibility is to apply the full SPARQL `FILTER` in Subject-Segmented data. However, this approach is only possible if the restrict conditions in the filter apply to the same subject. For instance, the SPARQL `FILTER` of the previous examples.

B. Slicing with Multi-join type Queries

As noticed by previous works [14], [15], the use of multi-join types query is not common. However, there are some queries that make use of multi-join BGP for data extraction. That is the case for the extraction of `owl:sameAs` data from a particular Graph Pattern (GP). For instance, the following BGP Listing 2 is taken from the Query 9 in Table III. Notice that the mentioned BGP in Listing 2 has a multi-join pattern, that is, the `variable` used in the join between lines one and two is different from the `variable` in the join between lines two and three. Our proposed approach (Torpedo) supports multi-join type queries as the simple join queries, which means at first evaluating the most restrictive triple pattern. In order to perform that, BGP containing multi-join type queries are divided in sub-BGPs. The sub-BGPs are slices from the SPARQL query where the GP contains a single variable in the join condition. For instance, the introduced BGP in Query 9 Table III is divided into two BGPs, one containing the subject with variable $?s4$ (Listing 4) and the other containing the subject-object join with variable $?o4$ (Listing 5).

```
1 ?s4 a           dbo:Drug .
2 ?s4 owl:sameAs ?o4 .
```

Listing 4. Listing the sub-BGP SS of Query 9

```
1 ?s4 owl:sameAs ?o4 .
2 ?o4 ?p5        ?o5 .
```

Listing 5. Listing a sub-BGP SO of Query 9

[5]http://www.w3.org/TR/sparql11-query/#evaluation

In this example, the most restrictive triple pattern is found in the first slice (Listing 4) (`?s4 a dbo:Drug.`). However, when a triple matches the most restrictive triple pattern, it is used to generate a candidate match for the most restrictive triple pattern of the sub-BGP. Thus, a match in the most restrictive triple pattern in the given example generates a candidate for the most restrictive triple patterns in its subgraph patterns. That is, in the previous example, if the triple `dbr:Aspirin a dbo:Drug` is a candidate for the most restrictive triple pattern, it will generate the candidate `dbr:Aspirin a dbo:Drug` for Listing 4 and `dbr:Aspirin owl:sameAs ?o4` for Listing 5. Finally, the evaluation of the multi-join BGP is given by applying a logical `AND` between all its sub-BGPs. Unfortunately, there are limitations in the described approach. The most restrictive triple pattern is used to evaluate the candidates in all sub-BGPs. Thus, all sub-BGPs are forced to share at least one variable or constant with it. For instance, in the given example above, the variable $?s4$ of the most restrictive triple pattern (`?s4 a dbo:Drug`) is shared between all sub-BGPs (see Listing 4 and Listing 5).

C. Complexity analysis

In this sections we extend the previous discussed complexity analysis of RDF dataset slice extraction process [14], [15] to `SPARQL FILTER`s and multi-join type queries.

Theorem 1: The complexity of extraction using filter or multi-join type queries is $O(n \log n)$.

Case 1: The complexity of extraction using filter is $O(n \log n)$.

The extraction can be performed by evaluating the triples that match the SPARQL `FILTER`s. Since filters use constants (see Section III-A), in the worst case, the selection can be performed in three passes by using the generic method. First pass evaluates the most restrictive triple pattern. Second pass evaluates if the triple patterns fulfill the assigned BGP filters, i.e. it evaluates the logical `AND` of each individual SPARQL `FILTER` F in the BGP ($F_1 \wedge F_2 \wedge F_n$). The third pass performs the selection. As previously discussed, the generic method can be performed in $O(n \log n)$.

Case 2: The complexity of extraction using multi-join type queries is $O(n \log n)$.

The extraction of a join can be performed by evaluating the triples that match each of the sub graph patterns (see Section III-B). Since the evaluation of the sub graph patterns can be performed in the worst case in $O(n \log n)$, the complexity of extracting a slice from a multi-join type query can be executed in $\Theta(sub_{GP} * nlogn)$ where sub_{GP} is the number of sub graph patterns in a query. As the number of sub graph patterns tends to be inferior to five [14], [15], sub_{GP} can be evaluated as a constant, therefore $sub_{GP} << n$. Furthermore, as discussed before, one pass ($nlogn$) is enough to evaluate all triple patterns in a BGP. Thus, the complexity of evaluating a multi-join type query in practice is $O(n \log n)$.

D. Optimizing Disk Seek Operations

Read and write operations can be very costly, especially when carried out on *disk* (e.g., memory access can be more

Fig. 1. In this example, the self-configured in-memory layer detects an availability of 4 GB on a machine. In the diagram, one step corresponds to the amount of time needed to store 1 GB of data. After the fourth step, data is stored in-disk.

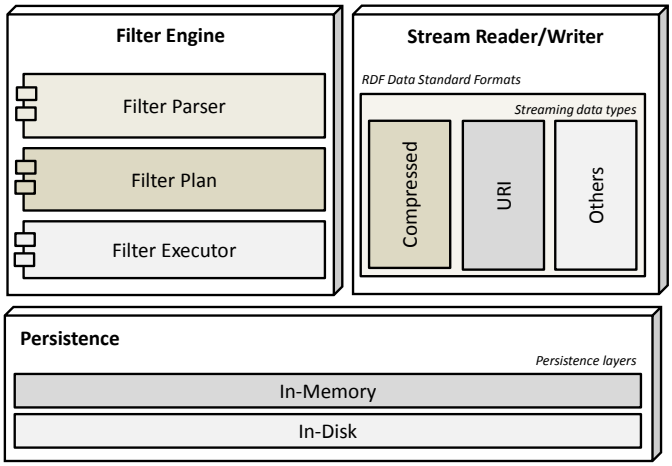

Fig. 2. Torpedo Engine Architecture.

than 10^5 times faster than disk access [12]). However, memory is a scarce resource that may simply not be available for storing a large volume of data. In order to further optimize the RDF stream execution, we provide a transparent two-layer persistent system. With the use of the transparent layer, the data flow is directed to the fast in-memory layer first. When the volatile layer reaches its limit, the data flow is automatically directed to the slow in-disk layer. The data access is done in the same fashion of the persistence; that is, if the desired information is not found in the in-memory, then it is sought in the in-disk persistent layer. Moreover, the in-memory seek operation is optimized using a hash index. All information persisted on the in-memory layer is converted to a *perfect hash* key that helps to uniquely identify it, avoiding collisions. Thus, in-memory entries can be accessed efficiently. To allow the RDF engine to be used in a wide range of different scenarios, we ensure that the in-memory layer is self-configuring. When the application starts, it automatically detects how much memory is available for its execution. Smartly, a concise portion (70%) of the available memory is allocated to the in-memory persistent layer.

E. Architecture

The architecture of RDFSlice is depicted in Figure 2; it consists of three main components. (1) The *Stream Reader* component is responsible for reading stream data which can flow from different sources. Approaches for publishing Linked Open Data may be as diverse as data dumps, data streams, and SPARQL endpoints. The data formats can also be disparate. Torpedo Stream Reader is built upon the OpenRDF API,[6] which is capable of supporting all mainstream RDF formats such as RDF/XML, Turtle, N-Triples, N-Quads, JSON-LD, TriG and TriX. However, the optimized selection of BGPs containing SO and SS in sorted files is only available for N-Triple format [14], [15]. It is also possible to deal with different streaming data types, which allow to query large data dumps, as well as continuous streaming data flows. Nevertheless, differently from Linked Streaming Data approaches, RDF slice approaches does not implement time windows. Thus, it is designed to query only for Subject-Segmented or sorted data streaming [14], [15]. Another important feature of the Stream Reader module is the ability to reestablish the data streaming from the point where it was interrupted, available only for remote data dump slicing. (2) The *Query Engine* executes the parser of the *Filter clause* and selects the relevant triples from the incoming data stream. The Query Engine has tree modules: (i) Query Parser, (ii) Plan and (iii) Executor. The Query Parser parses the query. The Query Plan decides what is the best data selection method based on the data type (data dumps, continuous streaming) and format (i.e. N-Triples, Turtle, etc). For instance, data dumps may be unsorted and thus need a more complex operation (see subsection III-C). The Query Engine supports two types of filters, Regex and SPARQL Query. The Regex Filter is designed to enable the selection of data by using simple string patterns. SPARQL Query Filter is more flexible and provides a richer data selection by using Basic Graph Patterns. However, not all features available in SPARQL queries are supported by the Query Engine, i.e.:

1) The use of triple patterns in which a triple shares more than one join condition with another triple (see Definition 4);
2) Use of Solution Modifiers (i.e., `ORDER BY` or `PROJECTION`);
3) Other Query Forms if not `SELECT` (i.e., `DESCRIBE`, `CONSTRUCT` and `ASK`).

(3) The Persistence Module performs the data access management, manages the persistence, as well as updates the retrieval of the join candidates. The Persistence Module contains two Persistence layers, i.e. *in-Memory* and *in-Disk*. The in-Memory yields fast update operations, but the available size can be insufficient, depending on the slice operation. Thus, the in-Disk acts as a contingency solution, providing more space when necessary. The data management policy implemented by the Persistence Module is better explained in the next section.

IV. Evaluation

The goal of our evaluation was to answer the following questions:

1) How efficient is our approach perform in standard benchmarks?
2) How it performs with regard to the state-of-the-art RDF slice approach?

[6]http://rdf4j.org/

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 I
RDF SLICE BENCHMARK QUERIES [14], [15].

3) Does the proposed approach scales in large size datasets?
4) How does our approach performs in comparison of other approaches as triple stores and Linked Data Streaming (LDS)?

Due the experiments, all tests with triple store were performed using an open-source *Virtuoso Server*[7] version 7.0.0. To this end, we used the previous implemented a Virtuoso utility application [14], [15]. The Virtuoso application was built using the Virtuoso JDBC Driver.[8] The rationale of using the Virtuoso JDBC Driver was to communicate directly with the Virtuoso instance without the overhead generated by other means such 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`[9] was used in order to speed up the loading. All files generated during the evaluation (e.g. logs and tables) are available online.[10]

A. Experimental Setup

We evaluated our approach on four interlinked datasets, i.e. DrugBank,[11] Sider,[12] Diseasome[13] and the full English dataset of DBpedia Version 2014. Table II shows the sizes of these datasets. DBpedia-slice refers to a version of DBpedia comprising all DBpedia datasets except *page links* and *info box properties*. There are several SPARQL benchmarks designed for measuring the performance of triple stores [18], [4], [8], [17] or RDF streaming approaches [21]. They focus on evaluating SPARQL features (e.g. Order By, Group By) in endpoints or streaming RDF data rather than the extraction of relevant fragments. Thus, we use the previously introduced RDF Slice benchmark [14], [15] with addition to Query 9 Table I. The RDF Slice benchmark [14], [15] in Table I was generated by two experts in SPARQL and in the ontology of the underlying datasets created nine SPARQL queries shown in Table I. The RDF Slice benchmark queries were designed to evaluate scenarios where a slicing tool can be used. The provided queries take into account the two most frequent types of join (i.e., `SS` and `SO`) and their resp. time complexities. The join types of the associated BGPs and the related datasets are shown. For instance, query `Q1` running on DBpedia contains eight triple patterns, which can be divided into four disjoint BGPs having as join type either subject-subject (SS) or subject-object (SO). The benchmark does not take into account queries containing patterns with join types SP, OP, and OO, since they are very rare (∼5%) in real SPARQL queries and have the same complexity as SO queries [14], [15], [16]. We measured the performance of our slicing approach with Query 9 Table I in terms of `runtime` and `memory consumption`. For the evaluation, we considered a scenario where a user is searching for the relevant fragment in the LOD Cloud. Therefore, we compared our proposed approach with the traditional method of downloading, loading and slicing datasets with triple stores. Although this paper extends the first approach on RDF Dataset Slicing, we conduct comparative experiments with other frameworks. The experiments were performed on a Windows 7 machine with an Intel Core M 620 processor, 6GB of RAM and a 1TB SSD.

Dataset	Size	Triples	Entities
DBpedia$_{2014_{en}}$	122 GB	858,089,974	29,747,387
DBpedia-slice	61 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 II
EVALUATION DATASETS STATISTICS.

B. Results

Figure 3 presents the *runtime* and *memory* consumption on execution Torpedo. We also measured the *Explored graph* and the *Disk Space* consumption during the slicing process. Similar to the previous works [14], [15], these parameters were recorded after each time one additional MB or GB of data was processed resp. on small and large files. The diagram shows the dataset being analyzed at most three times during the slicing process. Therefore, the Figure 3 of the explored graph contains three hops. In regards to the optimizations discussed in Section III, results show that Torpedo gains 26% on runtime performance while compared with the state-of-the-art.

Table IV show the total runtime required for slicing DBpedia on either triple store and Torpedo. In all cases we consider

[7]http://virtuoso.openlinksw.com/
[8]http://docs.openlinksw.com/virtuoso/VirtuosoDriverJDBC.html
[9]http://docs.openlinksw.com/virtuoso/fn_ld_dir.html
[10]http://aksw.org/projects/RDFSlice
[11]http://www.drugbank.ca/downloads
[12]http://wifo5-03.informatik.uni-mannheim.de/sider/
[13]http://diseasome.eu/

Query	Triple Patterns	Join	Dataset
Q9	{?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.} {?s4 a dbo:Drug. ?s4 owl:sameAs ?o4. ?o4 ?p5 ?o5. } {?s5 a dbo:Disease. ?s5 owl:sameAs ?o6. ?o6 ?p6 ?o7.}	SS+SO	DBpedia DrugBank Sider Diseasome

TABLE III
QUERY 9, AN EXTENSION OF THE QUERY 1 WITH THE ADDITION OF MULTI-JOIN BGPS.

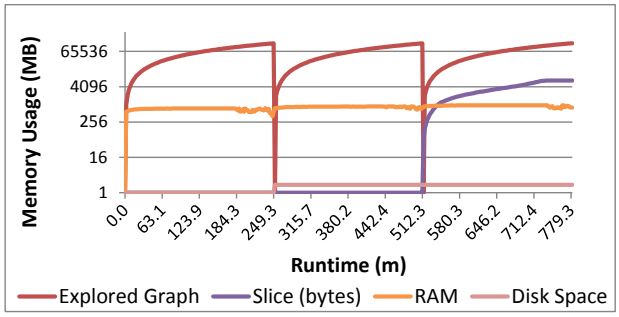

Fig. 3. Slicing unsorted DBpedia datasets using Q9.

Approach	Query	Sort	Load	Extraction	Total	% Gain
Triple store	Q1	-	875.6	1.8	877.4	64.66
Torpedo		-	-	310	310	
Triple store	Q2	-	875.6	2.1	877.7	64.11
Torpedo		95.7	-	219.3	315	
Triple store	Q6	-	875.6	0.0	875.6	89.07
Torpedo		95.7	-	0.0	95.7	
Triple store	Q7	-	875.6	0.0	875.6	89.07
Torpedo		95.7	-	0.0	95.7	
Triple store	Q8	-	875.6	2.95	878.55	87.13
Torpedo		95.7	-	17.38	113.08	

TABLE IV
A COMPARISON OF THE TOTAL RUNTIME IN MINUTES REQUIRED FOR SLICING $DBpedia_{2014_{en}}$ USING TORPEDO AND TRIPLE STORE WITH FIVE DIFFERENT JOIN TYPE QUERIES.

Dataset	Query	Extraction
DBpedia$_{2014_{EN}}$	Q1	310.1
DBpedia-slice$_{2014_{EN}}$	Q1	168.4
Drugbank	Q4	3.2
Sider	Q5	1.2
Diseasome	Q3	0.5

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

the scenario discussed in the introduction where the user does not have the dataset already indexed, but rather is looking for relevant fragments to be indexed. Thus, the load time (i.e., loading data into a triple store) is taken into account. Therefore, we consider the indexing time for each query. Three of the five queries—i.e, `Q1, Q2, Q6, Q7, Q8`—used in this experiment perform an order of magnitude faster in comparison to the total time computed over the triple store–i.e. `Q6, Q7, Q8` perform ∼80% faster than triple stores. The Table V presents the extraction time for slicing DBpedia, DBpedia-slice, Drugbank, Sider and Diseasome using respectively the queries `Q1, Q3, Q4, and Q5`. Table IV-B compares total runtime requiring for slicing DBpedia using Torpedo and the state-of-the-art RDF dataset slice approach.

C. Comparing different RDF streaming approaches

We evaluated our approach against CQELS and LDFragments. Table VII shows the result for streaming DBpedia over *CQELS* engine using *Q1*. The CQELS engine can query static files as well as continuous RDF data streaming; we evaluated it using both targets. For static files, the engine returns an `OUT OF MEMORY` error. This issue is due to CQELS processes static graphs using the Apache Jena[14] in-memory model, which consists of loading the whole graph into memory. Thus, approaches that load the whole dataset in memory are usually unable to handle big dataset files. Thereafter, we evaluate CQELS using continuous RDF data streaming from DBpedia dump files. The streaming was performed with two parameters: (1) using a time range of 10 seconds performing the query every 5 seconds (parameters `RANGE 5 STEP 5`) and (2) using time range of one day in tumbling mode (parameters `RANGE 1d TUMBLING`). In both cases, CQELS ran the processes for more than 36 hours and no results were given back. We terminated the process since it was not possible to estimate for how long it would have kept running. Nevertheless, we measured the time to process a chunk of 20,000 triples, which is described in Table VIII. Based on that, we estimated, supposing CQELS kept a constant processing time, it would take ∼284 days to process the whole DBpedia dataset dump files. We ran LDFragments using HDT index with the same setup to slice DBpedia and measure the performance. The results in Table VII demonstrate that the slicing process fails in the loading stage. The analysis of the LDFragment was made in two environments: first dedicating 4GB of RAM to the loading process, and secondly dedicating 170GB of RAM to the loading process. In both environments, the loading process failed to return an unexpected `OUT OF MEMORY` error. The later, after running for `150` minutes. When it comes to big datasets, the loading stage of HDT index did not perform

Version	Query	Extraction	% Gain
Torpedo	Q1	310	26.36
State-of-the-art	Q1	421	

TABLE VI
EVALUATING THE DIFFERENT SLICE RUNTIME BETWEEN THE TORPEDO AND THE STATE-OF-THE-ART RDF DATASET SLICING APPROACH.

[14]https://jena.apache.org/

well due to the linear growth of the memory consumption. Therefore, the use of using HDT index—including approaches that uses it e.g. LDFragments$_{HDT}$—is not indicated for slicing in scenarios where the datasets are large. For instance, when extracting relevant fragments from the LOD Cloud the user will require as much as RAM as the size of the LOD, which is not realistic in real scenarios. We also perform a slice using Jena and show that Torpedo is ∼75% faster in terms of runtime. In all cases Torpedo perform better because the runtime required for build the index in triple stores scenarios have the same time complexity as processing the whole triple file—i.e., O($n \log n$)–, but Torpedo stores a very small fragment of the required slice (i.e. the most restrictives triple patterns T_r), reducing the operations to $O(n \log T_r)$, wheres $T_r << n$. Thus, in practice, the slice operations performed by Torpedo are close to O(n), which explains why it achieves better performance in comparision with traditional approaches. Furthermore, as mentioned before, triple stores require the downloading and indexing steps, which, apart from the runtime, may require large storage consumption (e.g., to extract Drug and Disease information from the LOD Data Cloud (Q9), it will be required to download and index the entire LOD Data Cloud). Table VII shows *CQELS* compared in three modes: (1) using static graph; (2) in streaming mode with RANGE 10s and STEP 5s; (3) in streaming mode using 1-day RANGE in TUMBLING. The static graph mode generates an OUT OF MEMORY error since CQELS loads the entire graph into memory. The other two modes were interrupted after running for 36 hours and no result was found. Table VIII shows that the processing of big data streaming is inadequate for CQELS. LDFragments fails during the loading stage generating an OUT OF MEMORY error either using 4GB and 170GB of RAM dedicated to the loading process. Overall, our results show that our implementation of *Torpedo* clearly outperforms all alternative state-of-the-art approaches for dealing with slicing RDF data.

V. Related Work

To the best of our knowledge, Marx et al. [14], [15] provides the first work specifically targeting RDF dataset slicing. However, as pointed before [14], [15], RDF dataset slicing is related to the works in the three different areas a) crawling, b) RDF Indexes and c) Streaming. We briefly discuss them in the following subsections.

A. Crawling

RDF data crawlers are designed to harvest RDF content from the Web of Data or embedded in Documents. For example, *MultiCrawler* [9] allows the extraction of RDF published in HTML documents as well as files in a RDF-like format from the Web. It focuses on locating relevant links to content in order to extract data. Its approach is designed for indexing and finding documents with relevant content rather than extract fragments of larger datasets. *LDSpider* [11] is a lightweight RDF LOD crawler that delivers relevant RDF data crawled to the user. Through LDSpider, users can specify the relevant content that must be dereferenced. With *Semantic Web Client library* [10] users can perform SPARQL queries over crawled RDF data on query execution time. Although RDF dataset slicing approaches [14], [15] are also designed to extract relevant RDF data from the Web, they are not designed to traverse links and uses specific selection criteria (i.e., SPARQL graph patterns) to determine the data that should be extracted.

B. RDF Indexes

HDT [7] is a binary representation addressing the problem of efficient formats for publication and exchange of RDF data. The proposed approach is based on three RDF dataset components: (i) header information, (ii) a dictionary, and (iii) the actual triple structure, called HDT. This approach works in a streaming processing fashion. Using the HDT index, data published on the Web can be queried with simple triple patterns (i.e., (s, p, o), $(s, ?p, ?o)$ and $(s, p, ?o)$). The experiments show that datasets in HDT format can be compressed by more than fifteen times as compared to other representations. Yuan et al. [20] design TripleBit, a compact RDF store bit matrix storage structure and the encoding-based compression method for storing huge RDF graphs more efficiently. The storage structure based on bit matrix enables TripleBit to reduce the number and the size of indexes while speeding up the scan and merge-join performance.

C. Streaming

Linked Streaming Data (LSD) such as *CQELS* [13], *Streaming SPARQL* [5], *C-SPARQL* [3], *Sparql stream* [6] and *EP-SPARQL* [1] are designed for continuous query data streaming. The LSD approaches use a time window to define the data against which the query will be performed. LSD approaches allow users to select the query valuation criteria on either when window closes (e.g. C-SPARQL [3]) or changes (e.g. CQELS [13]). Some approaches as CQELS also supports the querying static graphs as dump files. Different from LSD, RDF data slicing approaches are designed for extracting relevant fragments from atomic data streaming, i.e., large files in the distributed static LOD.

RDF Streaming Indexes and their Applications. *LDFragments* [19] introduces a low-cost query technique based on triple pattern fragments. Here, the concept of fragments can be aligned to the notion of query evaluation previously introduced by Marx et al. [14]; however, fragments are extended to triple patterns, pages, and collections. LDFragments employs the HDT index and introduces a client-side SPARQL query processing algorithm based on a dynamic iterator pipeline. As HDT allows only the selection of simple triple patterns, there is a need for a dynamic interaction between all triple patterns in a given SPARQL query. Thus, the proposed method generates many sub-queries per SPARQL query and is significantly slower than full-index approaches as triple stores. However, the authors argue pro its scalability.

VI. Conclusions

In this work, we present Torpedo, an RDF dataset slicing approach that enables users to slice portions of a dataset

Approach	Query	Machine	Load	Query Execution	Parameters	Extraction	% Gain
CQELS	Q1	PC	-	OUT OF MEMORY MAX 4GB RAM	STATIC GRAPH [RANGE 10s STEP 5s]	> 2160	>85.64
CQELS	Q1	PC	-	INTERRUPTED AFTER 36h	STREAMING [RANGE 10s STEP 5s]	> 2160	>85.64
CQELS	Q1	PC	-	INTERRUPTED AFTER 36h	STREAMING [RANGE 1d TUMBLING]	> 2160	>85.64
Jena	Q1	PC	21h	12m	-	1272	75.66
LDFragments$_{HDT}$	Q1	PC	OUT OF MEMORY	4m	MAX 4GB RAM	FAIL	-
LDFragments$_{HDT}$	Q1	SERVER	OUT OF MEMORY (>150m)	4m	MAX 170GB RAM	FAIL	-

TABLE VII
COMPARING OTHER RDF STREAM ENGINES IN SLICING DBPEDIA USING QUERY 1 (Q1).

Approach	Total Time(s)
CQELS	572.325
Torpedo	0.135

TABLE VIII
COMPARING THE TIME FOR PROCESSING 20,000 STREAMED TRIPLES BETWEEN TORPEDO AND CQELS.

that they want to use in their applications. The proposed approach works through a portion of SPARQL which we dubbed eSliceSPARQL (a relevant and frequently used subset of SPARQL). We showed that our approach for RDF dataset slicing named as Torpedo allows to select and extract relevant knowledge from linked open datasets efficiently. We provided a performance optimization through a transparent in-memory layer. We evaluated different performance aspects of RDF dataset slicing on the DBpedia, DrugBank, Sider and Diseasome datasets. The observed results show an increased performance w.r.t. the state-of-the-art RDF dataset slicing, triple store, and RDF streaming approach. The main practical benefit of our work is that it eliminates the need to index and query irrelevant and potentially very large parts of datasets, which can be resource-demanding and time-consuming. Torpedo achieves this by only considering relevant portions of a dataset for a particular use case. In future work, we will study maximal fragments of SPARQL that can be executed efficiently using our approach.

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] Carlos Buil Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II*, pages 277–293, 2013.

[3] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, and Michael Grossniklaus. An execution environment for c-sparql queries. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 441–452, New York, NY, USA, 2010. ACM.

[4] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 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] Javier D. Fernndez, Miguel A. Martnez-Prieto, Claudio Gutirrez, Axel Polleres, and Mario Arias. Binary RDF Representation for Publication and Exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013.

[8] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[9] Andreas Harth, Jrgen Umbrich, and Stefan Decker. Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In *In 5th International Semantic Web Conference*, pages 258–271, 2006.

[10] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing sparql queries over the web of linked data. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 293–309, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Robert Isele, Jürgen Umbrich, Christian Bizer, and Andreas Harth. Ldspider: An open-source crawling framework for the web of linked data. In *Proceedings of the 2010 International Conference on Posters & Demonstrations Track-Volume 658*, pages 29–32. CEUR-WS. org, 2010.

[12] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, August 2009.

[13] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 370–388, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Edgard Marx, Saeedeh Shekarpour, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Large-scale RDF Dataset Slicing. In *7th IEEE International Conference on Semantic Computing, September 16-18, 2013, Irvine, California, USA*, 2013.

[15] Edgard Marx, Tommaso Soru, Saedeeh Shekarpour, Sören Auer, Axel-Cyrille Ngonga Ngomo, and Karin Breitman. Towards an Efficient RDF Dataset Slicing. *International Journal of Semantic Computing*, 07(04):455–477, 2013.

[16] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Lsq: The linked sparql queries dataset. In *ISWC*, pages 261–269, 2015.

[17] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. FEASIBLE: A Feature-Based SPARQL Benchmark Generation Framework. In *International Semantic Web Conference (ISWC)*, 2015.

[18] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Spˆ 2bench: a sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[19] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. Querying Datasets on the Web with High Availability. In *Semantic Web Conference (1)*, volume 8796, pages 180–196. Springer, 2014.

[20] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. TripleBit: A Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

[21] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. Srbench: a streaming rdf/sparql benchmark. In *The Semantic Web–ISWC 2012*, pages 641–657. Springer, 2012.