LD-LEx: Linked Dataset Link Extractor (short paper)

Ciro Baron Neto[1], Dimitris Kontokostas[1],Gustavo Publio[1], Kay Müller[1], Sebastian Hellmann[1], Eduardo Moletta[2]

[1] AKSW, Department of Computer Science, University of Leipzig, Germany
email{cbaron,kontokostas,gustavo.publio,kay.mueller,hellmann}@informatik.uni-leipzig.de
[2] Federal University of Technology, Department of Computer Science, Paraná, Brazil
moletta@utfpr.edu.br

Abstract. With the steady growth of linked datasets available on the web, it becomes increasingly necessary the creation of efficient approaches for analyzing, search and discover links between RDF datasets. In this paper, we describe LD-LEx, an architecture that creates the possibility of indexing RDF datasets using GridFS documents and probabilistic data structures called Bloom filter. Hence, our lightweight approach provides metadata about quantity and quality of links between datasets. Moreover, we explored these concepts indexing more than 2 billion triples from over a thousand of datasets, providing insights of Bloom filters behavior w.r.t. performance and memory footprint.

Keywords: RDF, Bloom Filter, Linksets, Linked Open Data

1 Introduction

The number of datasets in the Linked Open Data (LOD) cloud have grown significantly in the last couple of years.[3] Finding links between instances of different datasets is rather important since links play important roles in different domains, like question answering, data integration, and large-scale inferences. Therefore, due to the growing number of available datasets, searching and identifying suitable linked datasets becomes more and more difficult since users frequently have to rely on erroneous and outdated datasets metadata description [5,8]. The high level of inaccurate metadata is due to the fact that frequently these data are manually inserted by dataset creators or maintainers.

Given the described challenges, we present *LD-LEx* which consists of a set of lightweight methods based on Bloom filters (BF) [4] and MongoDB[4] that provides fast and accurate link extraction. Here, we close the gap on the generation of reliable metadata describing linksets between datasets. Our methods are based on the premise that it is possible to store datasets as a set of hashes, as long as

[3] `http://lod-cloud.net/#history`
[4] `https://www.mongodb.org/`

it is not necessary to retrieve raw data. Hence, dealing with fixed-sized hashes instead of raw data guarantees good performance and accurate link extraction.

Our novel approach detects and extracts links between datasets using hashes comparison. Thus, we use Bloom filters (BF) to create a map of hashes which represents a dataset, and later can be used as a query endpoint. The link extraction is performed on the fly when a *source dataset* is being streamed. Moreover, we provide an endpoint with RDF data describing indegree/outdegree between datasets, subsets and distributions using the Vocabulary of Interlinked Datasets (VoID)[1], the PROV Ontology (PROV-O)[5] and Data Catalog Vocabulary (DCAT) [8]. More specifically, the links are described using `void:linkset`.

In a nutshell, LD-LEx streams datasets and creates an index based on BF, in which can be further used to compare with other datasets. The methodologies of link discovery described in this paper were implemented as the backbone of the LODVader framework [2] [10]. In [2] we provided a demo focused on the visualization capabilities that are enabled by our architecture, i.g. providing a graph showing datasets and links. In [10] we provided basic link statistics for the current LOD cloud and we briefly discussed GridFS and BF techniques. In this paper, we present in details the methods which are responsible to index the triples, and results w.r.t. Bloom filter precision. Moreover, we describe the architecture of link extraction in specific explaining the role and complexity of bloom filters in the link extraction context.

The remainder of this work is structured as follows: We provide a description of Bloom filters and *linksets* in Section 2, followed by the implementations details in Section 3. Section 4 describes the results and evaluation obtained using our approach, and, in Section 5 we present the related works. Finally, Section 6 we present our conclusions and future work.

2 Background

2.1 Bloom Filters

Bloom filters (BF) are used to create the search indices used in the *buckets* (cf. Section 3). A Bloom filter is a probabilistic data structure created by Burton H. Bloom in 1970 [4]. The main goal is to check whether an element x exists in a set S. This data structure has 100% recall (*false negative* (fn) matches are impossible) while a small percentage of *false positives* (fp) are condoned.

Therefore, the fp margin of error can be adjusted in advance. The *false positive probability* (fpp) is calculated according to the dataset size. Equation 1 defines the fpp value used in our experiments. An fpp of 0.9/distributionSize guarantees an expected value (EV) of finding 0.9 links per distribution that are not links (false positives).

$$fpp = \begin{cases} 0.9/distributionSize, & \text{if } size > 1000000, \\ 0.0000001, & \text{otherwise.} \end{cases} \quad (1)$$

[5] https://www.w3.org/TR/prov-o/

In order to have a fixed *fp* rate, the length of the structure must grow linearly with the number of elements. The total number of bits m for the desired number of elements n and *fp* rate p, is defined as:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{2}$$

An optimal number of hash functions is given by:

$$k = (m/n) \ln 2. \tag{3}$$

LD-LEx methods are based on the optimal number of hash functions, since reducing the number of hashes would significantly decrease the BF precision. The space and time advantages of this probabilistic data structure are more coherent in this scenario compared to more commonly used data structures, such as binary search trees, hash tables, arrays or linked lists. More detailed BF benchmarks can be found at Putze's work [12].

2.2 Linkset Definition

Linksets are RDF descriptions of relations between datasets, subsets or distributions, represented by links. We adopted the DCAT and VoID vocabulary to describe the number of links, as well as source and target datasets. Therefore, in order to clarify the definition of the existing variables for a *linkset*, a brief explanation is given.

- ID: a dataset, described by `void:Dataset` or `dcat:Dataset`;
- $< s, p, o >$: the RDF triple which represents the subject s, predicate p and object o for a given relation.
- d_n: the n-th distribution consisting of a set of RDF triples.
- D_{ID}: the set of distributions, described by `dcat:distributions`, of the dataset ID .
- $L_{d_s \to d_t}$: the set of existing links between two distributions, having d_s as source distribution and d_t as target distribution. We define that a link occurs from a distribution d_s to a distribution d_t whenever d_s contains $< s_s, p_s, o_s >$ and d_t contains $< s_t, p_t, o_t >$ where $o_s = s_t$. We then call the triple $< s_s, p_s, o_s >$ in the source distribution a link (regardless of the used property) and say that the distributions are linked with each other. From this definition it easily follows that *linksets* between distributions (subsets or datasets) can be aggregated in a straightforward manner. Consequently, a dataset ID_s is linked to another dataset ID_t, if a non-empty *linkset* from any distribution $D_{S_{ID_s}}$ to $D_{S_{ID_t}}$ exists.

3 Design overview - The Bucket Structure

In order to extract links between datasets, we need to create indexes which represent *subjects* and *objects* of an RDF dataset. On logical grounds, the first

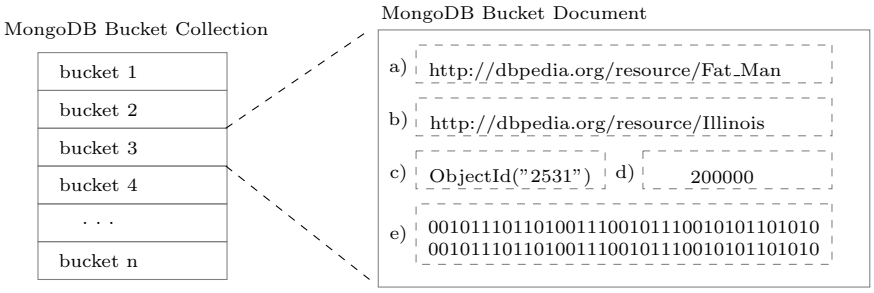

Fig. 1: Details of the *bucket* collection and document.

approach would be to create two BFs, the first containing all *subjects* and the other containing all *objects*. Hence, for search operations within a dataset would be necessary to query the specifics BF to find out whether the resource is present. However, considering that we are creating BFs with high precision, the size of a BF might be large. Clearly, this is a problem, since to query a dataset we have to load the entire BF into the memory. Considering the precision described on eq. (1) and the performance of a regular SSD drive, if a dataset contains more than 200,000 triples we create multiples fixed-size BFs. The value of 200,000 was taken since is suitable for the SSD drive used in our experiments. The average size of a BF which holds 200,000 elements and has the false positive rate of 1 each 10 million elements is 840kB. Considering that an SSD easily reads up to 500MB/s, takes only 1.6 milliseconds to load one filter from the storage. Other values can be used, and the relation between BF size and precision can be seen in fig. 2. Furthermore, we use GridFS[6] to store and manage the filters and its bytecode serialization.

In GridFS the BF bytecode is stored in the format of collection documents, and thus, there is a possibility of adding metadata to the stored BF's. In this paper, we call the collection which stores BFs *bucket*. The in-memory *bucket* size complies with eq. (2). A full *bucket* size is around 840kb. Figure 1 depicts the structure of a *bucket* document. The most relevant metadata fields are *a)* and *b)* which represents the first and the last resource inserted in the BF, *c)* represents the *id* of the distribution. Having those data is vital since simplifies the process of search. For instance, it is possible to query the depicted BF for the resource http://dbpedia.org/resource/Hawaii since "Hawaii" is alphabetically comprised between "Fat_Man" and "Illinois". *d)* represents how many resources were inserted in the BF (normally 200,000 or less) and *e)* is the BF bytecode itself.

[6] https://docs.mongodb.org/manual/core/gridfs/

Round	Dataset	Dist.	Triples	tp	fp	Precision	F-Measure	Time
1	English DBpedia	48	812M	9,013,302,727	1.354	0.9999998	0.99999992	01:43:20
2	LOV Vocabularies	395	891K	31,027,759	77	0.9999975	0.99999875	00:01:12
3	Routers128	1	7k	2501	0	1	1	00:00:07
4	RSS-500	1	10k	1265	0	1	1	00:00:09
5	Brown Corpus	123	3,4M	890,760	3	0.9999966	0.99999831	00:00:42

Table 1: Performance and bucket precision

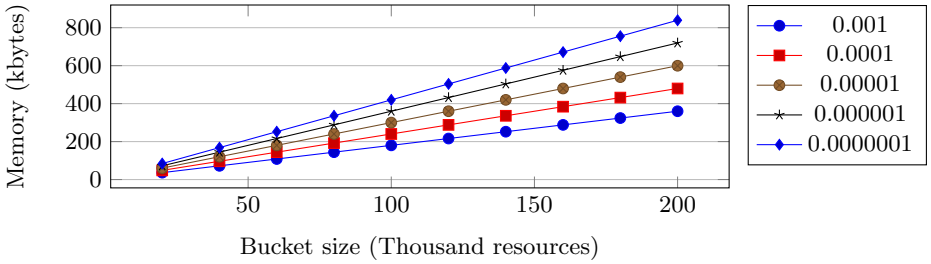

Fig. 2: Memory footprint varying the fpp and bucket size.

4 Results and Evaluation

All experiments were performed using Intel(R) Xeon(R) CPU X5650, 32GB of memory and 25TB of SSD in RAID 5. We used the BF implementation of Google Guava version 18.0. The source code is available at our Github[7] repository.

We created three setups for the evaluation. Firstly, we tackle the memory usage of the buckets, Figure 2 shows the memory footprint for the bucket structure. We depict five different levels of *false positive rate* varying from 0.001 to 0.0000001. Secondly, in order to assess our approach precision, we fetched all datasets from 48 English DBpedia distributions[8], 395 vocabularies available in LOV[9], and three NIF[6] corpora: Reuters128, RSS-500[10] and Brown Corpus[11] (which contains 123 distributions). Using fpp described in Equation (1), we measured the false positive rate. The results are shown in Table 1.

Overall, the general evaluation is given as follows:

Memory Efficiency: Figure 2 shows the size of the filter in different configurations. Overall, the bucket size grows linearly. Even with a high precision (fpp of 0.0000001), it is possible to index 200,000 triples using only 840kb of memory.

[7] https://github.com/AKSW/LODVader
[8] http://downloads.dbpedia.org/3.9/en/
[9] http://lov.okfn.org/lov.nq.gz
[10] https://github.com/AKSW/n3-collection/
[11] http://brown.nlp2rdf.org/

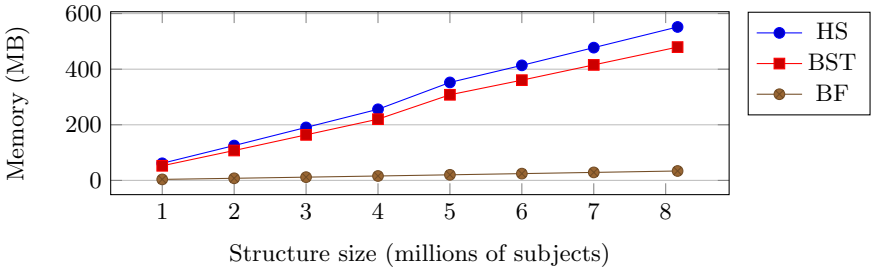

Fig. 3: Memory usage per indexed resource

Parameter	Jena TDB 2.12.0	Sesame 2.8.9	Virtuoso 7.1.0	*LD-LEx*
Load time	11:10:02	Out of MEM	24:02:36h	06:32:01h
Disk usage	75.83Gb	-	84.28Gb	4.03Gb

Table 2: Triple load time and disk usage

The small size influences in the query time since we have to bring the filter from the MongoDB. Any modern storage device could load a set of filters in a fraction of a second.

Bucket Precision and Disk Usage: Table 1 verifies the impact of BF false positives conducted in five experiments. It is clear that the precision of the BF remains high even when dealing with many millions of triples. Good performance is also essential. For example, for DBpedia distributions we could extract more than 9 billion links in 01:43:20. There are two reasons for the high number of links: (1) DBpedia is a dense graph with a high indegree within the dataset. (2) most of the distributions use the same set of subjects (the DBpedia identifiers). So if a DBpedia URI occurs as an object, it is highly likely to have an intersection with most of the distributions. Out analysis clearly show that the size of the dataset doesn't influence the high F-measure. The true positive (tp) score was measured with Binary Search Tree.

Figure 3 shows the main advantages of using BF w.r.t. the memory usage while varying the number of resources for each structure. The difference from HS and BST to BF is notable. Storing 8 million resources HS, and BST use over 0.5 GB of RAM memory. Considering that a regular dataset can easily have more than this number of triples, the usage of HS and BST is unfeasible. It is important to stress that Figure 3 shows the memory usage for loading only one structure. However, usually a dataset is compared with not only one, but multiple datasets, and memory efficiency is fundamental when multiple BF are loaded at the same time. BF fulfills its function using less than 34MB of memory, performing on average 12 times better than HS and 10 times better than BST.

We are aware that *LD-LEx* is not a triplestore, and do not store sufficient data do make an SPARQL query. However, it is important to notice that our methods are specialized to solve specific problems, and making complexes SPARQL

queries out of the scope of our approach. Table 2 shows the results w.r.t for loading and indexing triples and the total space used on the storage. OpenLink Virtuoso loaded triples in 24:02:23, while *LD-LEx* 06:32:01, making the execution 3.67 times faster. The amount of data stored was 84,28Gb for OpenLink Virtuoso and 4,03Gb for the current *LD-LEx* indexing strategy. For Jena TDB *LD-LEx* was 1.7 times faster indexing triples and used only 5% of the total space used by Jena. We could not load the datasets into Sesame since we got an *Out of Memory* error.

5 Related Work

In the RDF context, Bloom filters have been used in several works in the LOD domain. As an example, in [13], Bloom filters are used as an SPARQL extension for testing blank nodes membership. In addition, the paper shows that in certain circumstances Bloom filters have the potential to reduce the overall bandwidth requirements of queries. In some cases, queries were reduced by 97% of their original size. Furthermore, even when accepting a false positive rate, Bloom filters can reduce the I/O activity during analysis operations. In [7] the authors use same size Bloom filters to retrieve data from filter intersections. The evaluation shows a clear improvement over overlap-oblivious queries and preserved the perfect recall of almost all queries. The authors in [11] present an approach for answering queries through an evolutionary search algorithm which uses Bloom filter for rapid approximate evaluation of generated solutions.

There are several tools designed to reach Link Discovery over the LOD. In order to survey the state-of-the-art in existing solutions which could be applied to solve specific linking tasks, the authors in [9] present a survey over eleven different tools and frameworks that are available nowadays.

6 Conclusions and future work

In this paper, we presented *LD-LEx* which comprises in a set of methods for efficiently index RDF data. Since our approach is based on Bloom filters stored in a MongoDB database, our methods are fast enough to be used as a link discovery framework, analyzing datasets on the streaming time. Even for billions of triples, we keep a high accuracy and performance, showing that our approach can be used to create a central index for RDF data allowing the end user to create simple queries, retrieve data in `void:linkset` format, and retrieve statistical data. This work is implemented and is available as an open source project[12].

Although the methods described in this paper are fully implemented, as future work we still aim to improve in two directions. Firstly, we aim to index more datasets in the near future, extending our coverage to tens of billions of datasets. This is possible, for example, fetching data from LOD-Laundromat [3] and from other datasets hubs. Secondly, we plan to create a version which is fully scalable between multiple hosts.

[12] https://github.com/AKSW/LODVader

Acknowledgement This paper's research activities were funded by grants from the FP7 & H2020 EU projects ALIGNED (GA-644055), LIDER (GA-610782), FREME (GA-644771), from the Federal Ministry of Education and Research (BMBF) project Smart Data Web (GA-01MD15010B) and CAPES foundation (Ministry of Education of Brazil) for the given scholarship (13204/13-0).

References

1. K. Alexander and M. Hausenblas. Describing linked datasets - on the design and usage of void, the 'vocabulary of interlinked datasets. In *In Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference (WWW 09*, 2009.
2. C. Baron Neto, K. Müller, M. Brümmer, D. Kontokostas, and S. Hellmann. Lodvader: An interface to lod visualization, analyticsand discovery in real-time. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion. International World Wide Web Conferences, 2016.
3. W. Beek, L. Rietveld, H. Bazoobandi, J. Wielemaker, and S. Schlobach. Lod laundromat: A uniform way of publishing other people's dirty data. In *The Semantic Web – ISWC 2014*, volume 8796 of *Lecture Notes in Computer Science*, pages 213–228. Springer International Publishing, 2014.
4. B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.
5. M. Brümmer, C. Baron, I. Ermilov, M. Freudenberg, D. Kontokostas, and S. Hellmann. DataID: Towards Semantically Rich Metadata for Complex Datasets. In *Proceedings of the 10th International Conference on Semantic Systems*, SEM '14, pages 84–91. ACM, 2014.
6. S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer. Integrating NLP Using Linked Data. In *The Semantic Web – ISWC 2013*, pages 98–113. 2013.
7. K. Hose and R. Schenkel. Towards Benefit-based RDF Source Selection for SPARQL Queries. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM '12, pages 2:1–2:8, New York, NY, USA, 2012. ACM.
8. F. Maali and J. Erickson. Data Catalog Vocabulary (DCAT). W3C recommendation, W3C, Jan. 2014.
9. M. Nentwig, M. Hartung, A.-C. N. Ngomo, and E. Rahm. A survey of current link discovery frameworks. *Semantic Web*, (Preprint):1–18, 2015.
10. C. B. Neto, D. Kontokostas, S. Hellmann, K. Müller, and M. Brümmer. Assessing quantity and quality of links between linked data datasets. In *Proceedings of the Workshop on Linked Data on the Web co-located with the 25th International World Wide Web Conference (WWW 2016)*, Apr. 2016.
11. E. Oren, C. Guéret, and S. Schlobach. Anytime Query Answering in RDF Through Evolutionary Algorithms. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 98–113, Berlin, Heidelberg, 2008. Springer-Verlag.
12. F. Putze, P. Sanders, and J. Singler. Cache-, Hash-, and Space-efficient Bloom Filters. *J. Exp. Algorithmics*, 14:4:4.4–4:4.18, Jan. 2010.
13. G. T. Williams. Supporting Identity Reasoning in SPARQL Using Bloom Filters. In *Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*, 2008.