Towards an Approximated Accurate Measure of Links on the Data Web and a Scalable Generation Method for Dynamic LOD Cloud Diagrams

Ciro Baron Neto[1], Sebastian Hellmann[1], Diego Esteves[1]

AKSW`http://aksw.org`, Department of Computer Science, University of Leipzig, Germany
email{cbaron,hellmann,esteves}@informatik.uni-leipzig.de

Abstract. The Web of Linked Data is growing and it becomes increasingly necessary to discover the relationship between different datasets. This paper proposes an approach for accurate link counting which uses Bloom filters (BF) to compare and approximately count links between datasets, solving the problem of lack of up-to-date meta-data about linksets. We compare the performance to classical approaches such as binary search tree (BST) and hash tables (HT). The results show that Bloom filter is 12x more efficient regarding of memory usage with adequate query speed performance. BF maintains a low level of false positives and a precision level quite close to 100%. In addition, we created a small cloud comparing all English DBpedia datasets and vocabularies available in Linked Open Vocabularies (LOV). The whole process took only 1h45min. Moreover, we developed a GUI that uses the generated meta-data and creates a visualization of datasets and distributions, making it possible to apply real time filters. DataID files can also be enriched with VoID *linksets* based on statistical data generated by our approach.

Keywords: Linked Open Data, Cloud Diagram, DataID, Linksets, Bloom filter.

1 Introduction

The amount of datasets in the world of Linked Data is growing each day. Yet, counting links and visualizing the relations between these datasets is one of the challenges which the community still struggles to solve. The creation of diagrams which visualize the Data Web with a rich level of meta-data is a hard task. A crucial challenge of any analysis is to provide accurate methods to analyze links between data. While many approaches use approaches relying on a mixture of linked data crawling, exploitation of `http://datahub.io` and link counts based on *fully qualified domain name* (FQDN) in the linked data space, we argue that this does not accurately reflect the status of data integration in the Data Web.

A second challenge is the freshness of such an analysis. Especially in the case of large amount of big datasets, which are hosted in a decentralized way, detecting and counting links requires a centralized index and methods that are scalable enough to keep up-to-date. Having this index updated would easily be possible if dataset creators or maintainers provide basic meta-data regarding dataset description. However, the lack of updated meta-data description, especially about *linksets*[1] is still a critical problem and poses extra burden on the publisher.

Given the described gaps, Dynamic-LOD intends to explore a new paradigm in analyzing links on the Data Web. Our approach uses Bloom filter[2] (BF) to compare and count links among datasets solving the problem of meta-data generation describing *linksets*. A contribution that this work brings is the new process for link discovery and counting, which is made by comparing objects with subjects of different datasets, as well as ontologies. In particular, the subjects are indexed in BF vectors, which provide memory efficiency and high performance in the comparison process. Moreover, we store sufficient data about datasets and *linksets* to create a cloud diagram and show scalability.

In order to identify datasets and distributions, this work relies on DataId[4][2] files, which are used to get the meta-data description of datasets to be included in the diagram. A DataId file provides meta-data about multiple properties of a dataset, including subsets and its distributions. Based on this, our approach is able to enrich these DataId files with data about linksets, i.e. adding meta data about how many links a described distribution has with a different dataset. After the enrichment process, provenance of the newly generated meta-data is guaranteed since the Prov-O[3] ontology is used.

Our last contribution is the creation of a Graphical User Interface which allows users to add DataId files and visualize datasets in a cloud diagram format. The graph visualization is made using Data Driven Documents (D3) JavaScript library[4]. Our visualization graph provides a dynamic and interactive way for users to see links either on dataset level or on distribution level. This application is scalable enough and was tested with a small cloud with more than a billion of triples. Even with hundreds of datasets, it was possible to update the diagram in a few minutes for each newly included dataset.

Dynamic-LOD is available at `http://dynamiclod.dbpedia.org` and its implementation is open source and available at GitHub[5].

This work is structured as follows: We provide a description of *linksets* and Bloom filters in Section 2, followed by a motivational problem definition and relations to previous work in Section 3. Section 4 contains the Dynamic-LOD overview and implementations details. In Section 5 we describe results and evaluation obtained using our approach, and, in Section 5.3 and Section 6 we discuss about results and conclusions.

[1] `http://www.w3.org/TR/void/#linkset`
[2] `https://github.com/dbpedia/dataid/blob/master/ontology/dataid.ttl`
[3] `http://www.w3.org/TR/prov-o/`
[4] `http://d3js.org/`
[5] `https://github.com/AKSW/dynamiclod`

2 Background

2.1 Linkset Definition

In order to clarify the definition of the term *linkset* used in this work, a brief explanation is following given. Let ID be a dataset (as given by DataID), S_{ID} the set of subsets of the dataset ID, D_{ID} the set of distributions of the dataset ID and $D_{S_{ID}}$ the set of distributions of subset S of dataset ID. Each distribution d consists of a set of RDF triples in the form $< s, p, o >$. We define that a link occurs from distributions d_1 to distribution d_2 if d_1 contains a triple $t_1 =< s_1, p_1, o_1 >$ and d_2 contains a triple $t_2 =< s_2, p_2, o_2 >$ and $o_1 = s_2$. We then call t_1 a link (independently of the property used) and say that the distributions are linked to each other (cf. Section 3.2). Furthermore, let a linkset $L_{d_1 \to d_2}$ be the set of links in d_1.

From this definition it easily follows that linksets between distributions, subsets, datasets can be aggregated in a straightforward manner. We omit the details and just mention that a dataset id_1 is linked to another dataset id_2, if a non-empty linkset from any distribution $D_{S_{id_1}}$ to $D_{S_{id_2}}$ exists. For practical reasons (noise reduction) we later consider only linksets with more than 50 links: $|L_{d_m \to d_n}| \geq 50$ or $|L_{id_m \to id_n}| \geq 50$ respectively.

2.2 Bloom Filter

Bloom filter is a probabilistic data structure created by Burton H. Bloom in 1970 [2]. The main goal is to check whether an element x exists in a set S. In this data structure with 100% recall, *false negative* (fn) matches are not possible, while a small percentage of *false positives* (fp) are condoned. Queries will always return either "possibility the element x exists in set S" or "sure element x does not exists in set S". The proposal of Bloom filter is to handle with search algorithms in a large amount of data in a much more performative way, with a margin of error viable within certain applications in many different domains [6,14]. Usually fewer than 10 bits per element are necessary for a 1% of fp probability, independent of the size of the set S [3]. The weakness of Bloom filter is the possibility for a *false positive* (fp), which are elements that are not part of S but are reported being in the set by the algorithm.

The Bloom filter was chosen according to [18,20]. Besides, the space and time advantages makes this data structure more coherent on this scenario rather than binary search trees, hash tables, arrays or linked lists, for example. A benchmark can be found at Putze's work [18].

Bloom filter is an essential part of this paper because *linkset* indexing relies on it. Dynamic LOD uses this model in order to identify the links among distributions of datasets loaded from DataId files.

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:

$$m = -\frac{n \log p}{(\log 2)^2} \tag{1}$$

Besides, an optimal number of hash functions is given by:

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

The $false\ positive\ probability$ (fpp) is calculated according to the dataset size. Equation 3 defines the fpp value used in our experiments. Our application uses a *properties* file which allows to customize fpp equation. An fpp of 0.9/distributionSize guarantees an expected value (EV) of finding 0.9 links per distribution that are not links (false positives).

$$fpp = \begin{cases} 0.9/distributionSize, & \text{if } size > 1000000, \\ 0.0000001, & \text{otherwise.} \end{cases} \tag{3}$$

3 Problem Definition and Related Work

3.1 Link Granularity

Exploring linked datasets becomes more and more important since new datasets are often released. A general visualization of the relations among these datasets is needed in order to have a comprehension of the current structure of Linked Data. Several works such as [15,19,13] explore different ways to create diagrams and represent datasets and their relations. However, a major drawback of these approaches is the lack of a granular model of dataset meta-data, which serves as a prerequisite for accurate link counting. Such a model is provided by DataId[4], a DCAT[6] extension similar to a VOID file, which provides URIs for datasets that can be used as identifiers. Furthermore, DataID provides a mechanism to define subsets of datasets.

Figure 1 shows a complete overview of links in different levels of granularity regarding *linkset* representation. Datasets are represented by ID_n (as given by DataID), subsets represented by S_n and distributions represented by D_n. L_{real} is a *linkset* containing links between two distributions as measured on the intersection of subjects and objects (cf. Section 2.1). Note that it is irrelevant whether the distribution belongs to the same dataset or not for the measurement. The distributions represent the basic building blocks for aggregation. The *linksets* L_1 to L_4 can be generated by calculating the union of the linksets between all distributions of the respective subsets and datasets.

Existing approach like [19] assume pay-level domain and sub-domains as the basis for dataset definition and are using granularity level of L_4. Taking advantage of DataId descriptions, our approach allows to use different levels of granularity in a visualization graph and expand datasets to their subsets and distributions.

[6] `http://www.w3.org/TR/vocab-dcat/`

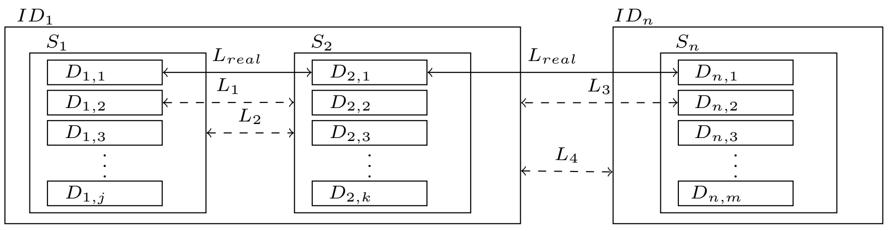


Fig. 1: The full arrows (L_{real}) represents links between distributions. The dotted arrows are: L_1 distribution to subset, L_2 subset to subset , L_3 distribution to dataset, L_4 dataset to dataset

3.2 Linking Predicates

Another classification of linkage can be done by analyzing the linking predicate. While `owl:sameAs` has well-defined formal semantics and is the predicate closest to traditional deduplication or record linkage or object reconciliation in the database area, counting `owl:sameAs` links exclusively provides a very limited view of the Data Web and has been critized for being used carelessly and too frequent [8].

Several other properties have been proposed with `rdfs:seeAlso` and `skos:` { `exact | close | broad | narrow | related`} `Match` being the most common. Furthermore, domain-specific properties such as `http://rdvocab.info/RDARelationshipsWEMI/manifestationOfWork` or `http://dbpedia.org/ontology/birthplace` are employed.

In our work, we are lenient and consider all predicates for linking. While for crawling link direction is important – although DBpedia is the largest authority [19], no backlinks are included – we argue that linking properties are often either symmetrical [7] or it is feasible to assume that an inverse property exists or could be easily created, e.g. following a `birthplace`↔`isBirtplaceOf` pattern or simply `birthplace`$^{-1}$.

To the best of our knowledge, we have not encountered predicates expressing negative links yet (e.g. `notLinkedTo`).

Vocabulary Links . Another aspect of linking properties that is often neglected are links to vocabularies and links between vocabularies. Especially, the linkage via `rdf:type` has not yet been visualized in a cloud diagram and is often not included in link analysis. Reusing the

3.3 Link Validity

It's possible to discover and count links using web crawlers, for instance, using LDSpider[11] framework and dereference linked data URLs. The disadvantage

[7] and highly unlikely to be asymmetric

of using crawlers is that it takes long time to crawl large number of datasets, even using multiple threads creating HTTP connections for target datasets. An advantage is that link targets are validated, i.e. resolve in a linked data way. Our approach doesn't validate all links of a *linkset*. However, Dynamic LOD takes a statistical sample set for each *linkset* created, and makes the validation. Thus, we can express with certain interval of confidence the validity of a given *linkset*.

3.4 Bloom Filter on Linked Data

Even accepting a false positive rate, Bloom filters reduces the I/O activity during operations. For Linked Data, Bloom filters has been used in several approaches with different purpose.

As an example, in [21], Bloom filters are used as a SPARQL extension for testing blank nodes membership. It's possible to notice that in certain circumstances Bloom filters has the potential to reduce the overall bandwidth requirements of making a query, which in some cases queries were reduced by 97% of the size. In [10] the authors use same size Bloom filters to retrieve data from filters intersection. The evaluation shows clear improvement over overlap-oblivious queries, and preserved the perfect recall of almost queries. The authors in [16] present an approach for answering queries through an evolutionary search algorithm which uses Bloom filter for rapid approximate evaluation of generated solutions. Bloom filters don't allow elements removal from the set, thus, variations of the original Bloom filter had been used such in [17] where the author chooses Counting Bloom filters to filter non-matches candidates to a query.

3.5 Linked Open Data Cloud Diagrams

The LOD cloud diagram[19] is one of the main motivation for us, since there are several problems which can be pointed out. One of the main disadvantages of LOD cloud diagram, is that one assumes that every distribution in the same pay-level domain or sub-domains belongs to the same dataset. Thus, the LOD cloud currently fails to generate a precise diagram about the described datasets. Another weak point is the up-to-dateness, which means that new releases of the LOD cloud diagram can take years, thus it's not possible to know what's the real current state of the cloud diagram. Finally, another important point is the lack of filters as the result is a static image with no significant interaction between user and the diagram.

LODLive [5] is an environment that queries predefined SPARQL endpoints. This work allows to make filters in a dynamic way, however *linksets* are shown only within resources of a dataset, and they are not extensible to others datasets or subsets. Since it's not possible to query multiples datasets, LODLive does not provides methods for link discovery.

Protovis [1] was already been analyzed before in [12], and it's a graphical approach to visualizing the LOD Cloud diagram. The bubble colors reflect the CKAN rating, however the only user interactivity is when clicking on any bubble, the user is taken to the package details page at CKAN. The main advantage is that the diagram is automatically updated, since it uses CKAN API.

Nevertheless the cited diagrams has advantages which make them so important to community. It's important to stress that the visualization issue led to exists more approaches which are not cited here, however they have their importance in the different Linked Data domains.

4 Dynamic-LOD

4.1 Overview

Figure 2 shows an overview of the framework architecture. The service was written in Java and uses Apache Jena[7] to parse and write RDF data. MongoDB[8] has shown scalability and is used to store non-RDF data regarding to statistical data, internal paths, domain resources, etc. To implement Bloom filter, the Dynamic LOD uses Google Guava Library[9].

A Java class named Manager is the central controller of the service and is responsible for communication with the HTTP Servlet (which makes interface with the GUI) and manages the process of streaming dump files, Bloom filter creation and link counting. A Servlet interfaces with the GUI and a REST API. The GUI has two roles, first let the user add DataId files and second provide a diagram representing datasets and *linksets*.

Moreover, the GUI contains a log window fed by websockets providing real time information about the progress of datasets insertion and link counting. The REST API allows users to a) send a DataId, a dataset URL or a dataset distribution URL to be streamed and compared with previously loaded datasets and return meta-data in RDF about a specific dataset distribution, subset or dataset and b) post a list of dataset URLs and retrieve a LOD diagram.

The whole process was divided in four main stages which consists on parsing DataId file, streaming distributions, creating filters and link counting. The provided meta-data by our service can be used to for DataId enrichment or for diagram visualization. The following section provides a detailed description of each of the four stages.

1. **Parse DataId file and check for modifications**. In order to get a list of dump files to stream, Dynamic-LOD parses the received DataId file and searches for distributions using `dataid:Distribution` class or equivalent classes. Then the application fetches the `dcat:downloadURL` object and checks HTTP headers (Last-Modified and Content-Length) for each fetched distribution. This is made in order to verify whether data has been modified.
2. **Stream Distributions**. In case there are modifications or it's a new distribution, the Dynamic-LOD application streams the new dump file. Statistic calculations are made at this stage and subject and object of each triple are separated and saved in different files.

[8] https://www.mongodb.org/
[9] https://code.google.com/p/guava-libraries/

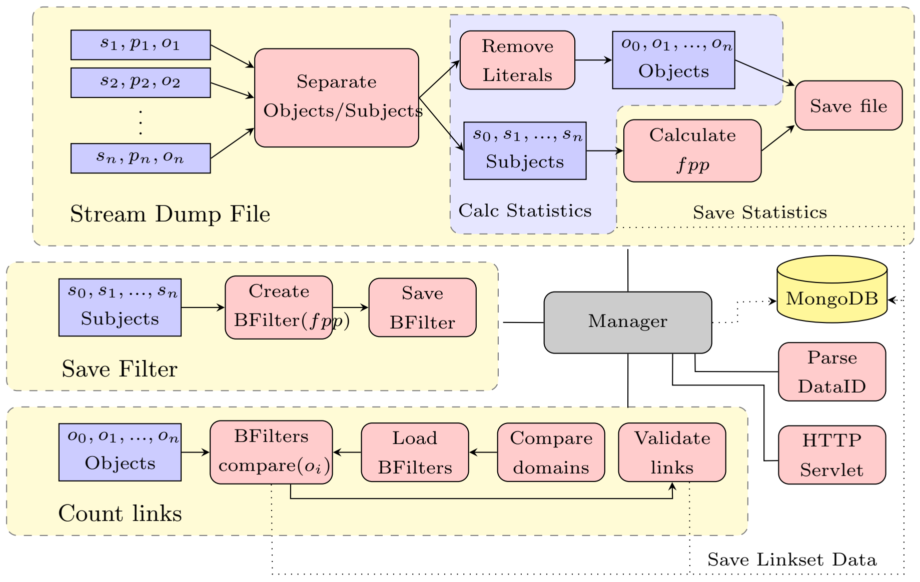


Fig. 2: Service overview

3. **Create Bloom filter**. For each distribution a Bloom filter is created using subjects stored by the previously step.
4. **Count Links and Update RDF**. At this stage, the FQDN filter loads the Bloom filters which might have positive values, then, objects of the distribution are compared with the Bloom filter vectors. Finally, a set of links is tested in order to verify whether the resources are available.

4.2 Implementation

We describe the parsing process of DataIds in Algorithm 1. Each DataId contains one or more datasets ID having zero or more subsets S, which can recursively have subsets of their own. Let *DataID distribution* D be defined as $D_x = \{t_1, t_2, ..., t_n\}$ where elements t_i are RDF triples. We use ID_s to denote *DataID subsets* where $D \subset ID_s$. *DataID datasets* ID are defined by $ID_s \subset ID$.

Algorithm1 demonstrates the process for parsing a DataId file. A set ID gets all parents datasets described in the file. For each subset $ID_s \subset ID$ the function to parse the dataset is called. Case the subset ID_s has distributions, and a new MongoDB object is stored saving meta-data about D. The status of D is changed to "WAITING_FOR_STREAMING", which means that the current distribution should be streamed before create the Bloom filter. If ID_s has subsets, the function to parse dataset is recursively called.

Algorithm 1 Parse DataID file

```
1: $ID$ ←readParentDatasetsFromDataId();
2: for all $ID_s$ in $ID$ do
3:     parseDataset($ID_s$)
4:     saveMongoDBObject($ID_s$))
5: end for
6: function parseDataset($ID$)
7:     saveMongoDBObject($ID$));
8:     if $ID$ has distributions then
9:         $D \leftarrow ID[distribution]$
10:        saveMongoDBObject($D$);
11:        setStatus($D$, "WAITING_FOR_STREAMING");
12:    end if
13:    for all Subsets $s$ of $ID$ do
14:        parseDataset($s$)
15:    end for
16: end function
```

The MongoDB contains collections that describe datasets, subsets, distributions and *linksets*. Here, MongoDB collections (except for linksets) are updated with meta-data fetched from the DataId file.

The next process stream distributions and is described in algorithm 2. For each distribution D which has a status that allows the distribution be streamed, a *newConcurrentAtomicQueue* is created. The *queue* necessary since multiple threads will be consuming triples. The first *thread* t_1 uses regular expressions to separate s_j and o_j. All literals are removed and two files are created, the first containing *subjects* and the second with *objects*. The second *thread* t_2 analyses and creates a list of *fully qualified domain name* (FDQN) plus the first string of the path (in the format `http://<authority>/<string>/`) for each s_j and o_j. The analysis of *fully qualified domain name* is important because will avoid unnecessary comparison between distributions that don't have common domains. Then, the D_i is streamed and each triple s_j, p_j, o_j is added to the *queue*. Finally, fpp is calculated based on the number of loaded *subjects*.

At this point, for each streamed distribution the application has two files, the first with the list of objects and the second with Bloom filter vector (which contains hashes from subjects).

Algorithm3 is the last to be described. At this stage the FQDN are compared and the necessary Bloom filters are loaded. Now, Bloom filters are effectively used to compare sets and count links among them. First for each distribution D a set P is created which contains the intersection of FQDN from the $D_i[objects]$ and $D[subjects]$. Note that subjects are from all other D which are different from the current distribution D_i since it's not necessary to compare a distribution with itself. Having P it's possible to know which Bloom filters should be loaded that might contain links with the D_i, and load them to a set of *threads*. Next, a $buffer$ is created containing the objects from the current distribution. The

Algorithm 2 Streaming new distribution

```
1: for all $D$ which should be streamed do
2:     $queue \leftarrow new\ concurrentAtomicQueue()$;
3:     $th_1 \leftarrow$ new Thread(separateObjectsAndSubjects($queue$);
4:     $th_2 \leftarrow$ new Thread(checkDomains($queue$));
5:     while $streaming(uncompress(D_i))$ do
6:         for all $s_j, p_j, o_j \in D_i$ do
7:             $queue.add(s_j, p_j, o_j)$
8:         end for
9:     end while
10:    $fpp \leftarrow calculateFPP(th_1.getNumberOfSujects())$
11:    updateMongoDBObjects(statistics);
12: end for
13: return $fpp$
```

threads of the set are started using the $buffer$ of objects to compare with the previously loaded Bloom filters. Finally, when all *threads* finish counting links, the data is saved as a MongoDB object and
set is tested to check links availability.

It is unfeasible to validate all links of a linkset and check whether they resolve. To determine the sample set size to be tested with certain confidence score, we extracted randomly (random sampling) values from the *linksets*, where the sample size is defined having 95% of confidence interval and $p < 2\%$. Future work will include a more complex statistics estimation process such as Wald's and bootstrapping methods and Monte Carlo simulation.

At this stage all the meta-data needed to create the LOD cloud diagram is stored in the MongoDB. The database contains enough data regarding to subsets, datasets and distributions (all meta-data taken by the DataId description). In addition, data about *linksets* are now available making possible to visualize links among different datasets. Regarding to figure 1 the representation might be done for L_1, L_2, L_3 and L_4.

It's important to stress that the application reads and retrieve, however doesn't store any RDF data. All RDF triples are created on the fly reading documents from MongoDB and using Jena to create RDF.

5 Evaluation

In order to demonstrate why Bloom filter (BF) was chosen, we first compared BF with two classical search structures: HashMap Search (HS) and Binary Search Tree (BST). We evaluate three parameters: memory usage for each structure, time to compare a set with 10 millions objects varying the structure, size and time to compare different datasets with a fixed structure size. Second, we measured the time of BF to compare and count links for 513 distribution files. Finally,

Algorithm 3 Counting links

```
 1: for all $D$ already streamed do
 2:     $P \leftarrow FQDN(D_i[objects]) \cap FQDN(\forall D[subjects] | D_i \notin D)$
 3:     for all subjectsBloomFilter $\in P$ do
 4:         $threads[i] \leftarrow$createNewThread($subjectsBloomFilter_j$);
 5:         $i \leftarrow i + 1$;
 6:     end for
 7:     while $buffer = read(D_i[objects])$ do
 8:         for all threads do
 9:             $createLinksets(buffer, thread_k)$
10:         end for
11:     end while
12:     while $threadsStillRunning()$ do
13:         $wait()$;
14:     end while
15:     $updateMongoDBObjects(linksets)$
16:     $newThread(testSample(linksets))$
17: end for
```

we adapted Dynamic-LOD to parse VoID files description[10] and measured the performance of link counting.

All experiments were made using a Intel(R) Core(TM) i7-3720QM (4 cores, 8 threads), 16GB DDR3 and a 250GB SSD Sata III drive.

5.1 Data Structure

Space Efficiency. The result (Figure 3) shows that the main advantages of using BF shows the memory usage for varying the number of objects for each structure. The difference from HS and BST to BF is notable. Storing 8 millions of objects HS and BST use more than 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 are unfeasible. It's 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 targets. BF fulfills its function using less then 34Mb of memory, performing in average 12x better than HS and 10x better then BST.

Varying the Structure Size. In the next experiment, we loaded to each structure from 10 to 75 millions of subjects, and we compared with a fixed set of 10 millions of objects, and the results are shown in the Figure 4a. For less than 65M of objects, the best performance was made by HS followed by BF and BST. Above of 65M, BF has the best performance, and the degradation of performance of HS and BST is given by the amount of memory used. HS and BST grows faster than BF, forcing the operational system to use swap memory

[10] http://lod-cloud.net/data/void.ttl

impacting directly on the performance. The same doesn't happen with BF, and it's possible to observe that for BF, the time remains almost stable even varying the size from 25M to 75M of objects.

Varying the Set Size. For Figure 4b, we added 10 millions of subjects in each structure, and we varied from 10 to 100 millions objects to be compared. The fastest approach is HS, following by BF and BST. The recall of all three approaches is 100%, since none of them allows false negatives. Using the algorithm described in Equation (3) for BF, the precision stays at 100%. The same is not true for HS, where from 1M the precision is 0.99987 and 10M is 0.99125. This means that for 100M triples using HS we might have more than 800 thousand of false positives which is unfeasible.

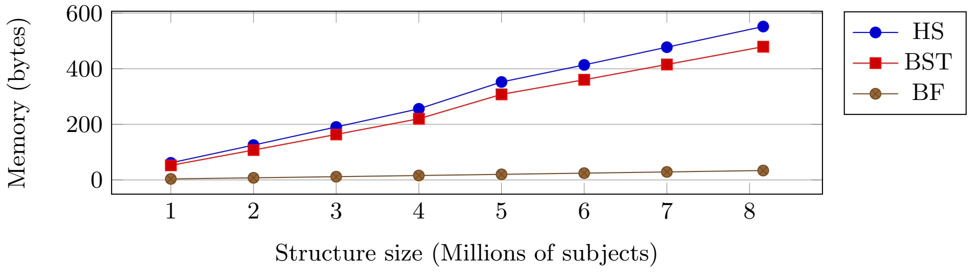
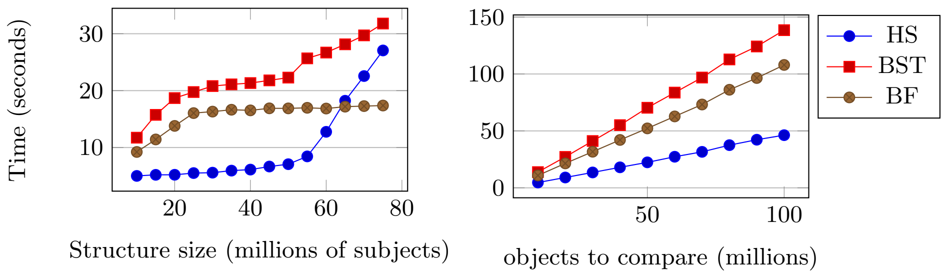


Fig. 3: Memory usage per indexed subjects



(a) Time to compare 10M objects (b) Time to search with different sizes

Fig. 4: In (a) we fixed number comparison to 10M objects, and in (b) we fixed the structure size to 10 million subjects

5.2 Quantitative Evaluation

Counting Links. To verify the impact of BF false positives, we made five experiments. For the first experiment, we added the 48 English DBpedia distributions[11] and counted links among them. Then, we added all 358 vocabularies available in LOV[12] (referred as round 2), and again we counted links among the vocabularies and DBpedia distributions. In the last three experiments, we added three NIF[9] converted corpora: Reuters128[13], RSS-500[14] and Brown Corpus[15] (which contains 123 distributions). We measured false positive rate and time to count links. All Bloom filters were created with fpp described in Equation (3). The results are resumed in Table 1. For DBpedia, up to 812 millions of triples in 48 distributions were analyzed. There are more than 9 billions of links among the 48 distributions, and BF had an average precision of 0,99999985 with 1.354 false positives total. 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 intersections with most of the distributions.

The time to count these links was 1h43m20s, and the amount of disk space occupied by BF was 771MB. In the second experiment, we added up to 862k triples from 358 datasets available in LOV. BF counted more than 31 millions of links having a precision of 0,999997518 with 77 false positives. To compare LOV datasets with the previously loaded DBpedia, took only 72 seconds.

In the last three experiments, we added three NIF corpora and for Reuters-128 and RSS-500 no fp were detected. For Brown corpora, the precision was 0.99999966 with 3 false positives.

Table 1: Adding new datasets

Round	Dataset	Dist.	Triples	*tp*	*fp*	Precision	F-Measure	Time
1	English DBpedia	48	812M	9.013.302.727	1.354	0,99999985	0,999999925	01:43:20
2	LOV Vocabularies	358	862K	31.027.759	77	0,99999752	0.999998759	00:01:12
3	Routers128	1	7k	2501	0	1	1	00:00:21
4	RSS-500	1	10k	1265	0	1	1	00:00:33
5	Brown Corpus	123	3.4M	890.760	3	0.99999663	0.999998316	00:04:04

Lod-cloud VoID Description. Although we don't have a DataId file to describe all datasets from lod-cloud, we adapted our application to parse the avail-

[11] http://downloads.dbpedia.org/3.9/en/
[12] http://lov.okfn.org/lov.nq.gz
[13] https://github.com/AKSW/n3-collection/blob/master/Reuters-128.nt
[14] https://github.com/AKSW/n3-collection/blob/master/RSS-500.nt
[15] http://brown.nlp2rdf.org/

able VoID file[16]. Thus, we can estimate how long it would take to create links in a wider scale among all available resources described by `void:dataDump` property. Only 62% of the resources were available to download, and the list of downloaded distribution can be found at our GitHub Webpage[17].

It took us around 1 hour and 56 minutes to download and create BF for all distributions. Naturally, the external factors (e.g. download speed) directly affects the performance of the system, considering that a considerable work is made while streaming the distributions. However, we measured the time to count links by comparing distributions which took us 49 minutes to create the *linksets*. This measure doesn't consider the time to validate dereferencability of the links, and the reason is that again, this depends of external factors (e.g. *timeout* of connections).

5.3 Discussion

We evaluated Dynamic-LOD in three different aspects: firstly by analyzing data structure performance comparing BF with HS and BST, secondly a quantitative evaluation regarding fp, speed to count links in a dense scenario like DBpedia and thirdly on a large scale based on lod-cloud distributions. In fact, all three evaluations indicates that BF is a good choice for what our work proposes.

The main advantage of BF over HS and BST is the memory efficiency. This importance is given because to compare datasets, usually it's necessary to load multiple filters in the RAM memory. Another interesting property of BF is the ability to control the false positive probability. An unexpected result was that the number of links can be higher than the number of triples due to the fact that –unlike linked data – distributions can be linked to several distribution according to our link definition. The Table 1 contains 531 distributions (which are dump files and vocabularies), and its possible to notice that the precision and F-Measure remain at a high level.

Even with this high precision level, BF uses only a few bits per entry, which justifies our choice for this structure. This also had sown when testing our approach on LOD.

6 Conclusions

For efficient comparison of datasets we used a probabilistic data structure called Bloom filter to count links between datasets and create a scalable implementation of a link analysis between dataset distributions. Therefore we can notice, according to the evaluation section that using Bloom filters has a good performance for comparing sets and storing data while accepting a low trade-off for correctness. While we approximate the count of linking, our definition of links is accurate (up to validation of the online status). Although the Bloom filter

[16] http://lod-cloud.net/data/void.ttl
[17] https://github.com/AKSW/dynamiclod

parameter that controls the false positive probability can be adjusted to the size of the dataset, we discovered that this assumption is not appropriate for distributions that are linked to many other distributions.

In this initial experiment and proof-of-concept implementation, we have shown that the performance of the Bloom filter data structure, the creation of *linkset* becomes feasible. We are providing up to date measurements and diagrams on `http://dynamiclod.dbpedia.org`

We are exploring the possibility of using Dynamic-LOD not only a stand alone service, but interface it with a DataId SPARQL endpoint to centralize and enrich DataId's. Moreover, we are aware that only accepting dump file is a limitation to our approach, future works would include reading SPARQL endpoint to get the data from distributions.

Acknowledgement. CAPES foundation (Ministry of Education of Brazil) for the given scholarship.

References

1. Protovis: Linked open data graph. 2012. available in: http://inkdroid.org/lod-graph/.
2. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
3. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*, pages 684–695, 2006.
4. 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.
5. D. V. Camarda, S. Mazzini, and A. Antonuccio. Lodlive, exploring the web of data. In *Proceedings of the 8th International Conference on Semantic Systems*, I-SEMANTICS '12, pages 197–200, New York, NY, USA, 2012. ACM.
6. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
7. M. Grobe. Rdf, jena, sparql and the 'semantic web'. In *Proceedings of the 37th Annual ACM SIGUCCS Fall Conference: Communication and Collaboration*, SIGUCCS '09, pages 131–138, New York, NY, USA, 2009. ACM.
8. H. Halpin, P. J. Hayes, J. P. McCusker, D. L. McGuinness, and H. S. Thompson. When owl: sameas isn't the same: An analysis of identity in linked data. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Z. 0007, J. Z. Pan, I. Horrocks, and B. Glimm, editors, *International Semantic Web Conference (1)*, volume 6496 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010.
9. S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer. Integrating nlp using linked data. In H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J. Parreira, L. Aroyo, N. Noy, C. Welty, and K. Janowicz, editors, *The Semantic Web – ISWC 2013*, volume 8219 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 2013.

10. 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.
11. R. Isele, J. Umbrich, C. Bizer, and A. Harth. Ldspider: An open-source crawling framework for the web of linked data. In *Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts, Shanghai, China, November 9, 2010*, 2010.
12. R. Kaiser. Visualizing open data. 2011. available in: http://courses.iicm.tugraz.at/ivis/surveys/ss2011/g4-survey-open-data-vis.pdf.
13. T. Käfer, J. Umbrich, A. Hogan, and A. Polleres. Towards a dynamic linked data observatory. In *In LDOW at WWW*, 2012.
14. A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
15. L. Matteis. VoID-graph: Visualize linked datasets on the web, 2014. `http://arxiv.org/abs/1408.6691`.
16. 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.
17. S. Paturi. *A new filtering index for fast processing of SPARQL queries.* PhD thesis, Faculty of the University Of Missouri-Kansas City, 2013.
18. 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.
19. M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *ISWC 2014*, pages 245–260, 2014.
20. S. Tarkoma, C. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys Tutorials, IEEE*, 14(1):131–155, First 2012.
21. G. T. Williams. Supporting identity reasoning in sparql using bloom filters. In *Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008).*