LODVader: LOD Visualization, Analytics and DiscovEry in Real-time

Ciro Baron Neto Kay Müller Martin Brümmer
Dimitris Kontokostas Sebastian Hellmann
Leipzig University, AKSW, http://aksw.org
Leipzig (Germany)
(cbaron|kay.mueller|bruemmer|kontokostas|hellmann)@informatik.uni-leipzig.de

ABSTRACT

The Linked Open Data (LOD) cloud is in danger of becoming a black box. Simple questions such as "What kind of datasets are in the LOD cloud?", "In what way(s) are these datasets connected?" – albeit frequently asked – are at the moment still difficult to answer due to the lack of proper tooling support. The infrequent update of the static LOD cloud diagram adds to the current dilemma, since there is neither reliable nor timely-updated information to perform an interactive search, analysis or in particular visualization in order to gain insight into the current state of Linked Open Data. In this paper, we propose a new hybrid system which combines LOD Visualisation, Analytics and DiscovEry (LODVader) to aid in answering the above questions. *LODVader* is equipped with (1) a multi-layer LOD cloud visualization component comprising of a light and a dark side, (2) dataset analysis components that extend the state of the art with new similarity measures and efficient link extracting techniques and (3) a fast search index that is an entry point for dataset discovery. At its core, *LODVader* employs a timely-updated index using a complex cluster of Bloom Filters (BF) as a fast search index with low memory footprint. This BF cluster is able to efficiently perform analysis on link and dataset similarities based on stored predicate and object information, which – once inverted – can be employed to discover broken links by displaying the Dark LOD Cloud. By combining all these features with a versioning system, we allow for an up-to-date, multi-dimensional LOD cloud analysis, which – to the best of our knowledge – was not possible before.

Keywords

Linked Open Data, Linksets, Bloom filter, RDF diagram

1. INTRODUCTION

The amount of datasets in the Linked Open Data (LOD) cloud have grown significantly in the last couple of years.[1] Due to the growing number of available datasets, identifying relevant linked datasets becomes more and more difficult. In addition the analysis, visualization and exploration of these linked datasets requires more computational resources. Currently, most available LOD cloud frameworks are specialised on specific tasks such as visualization of the whole LOD cloud or collecting statistical information about the datasets. Furthermore, searching for datasets in terms of used vocabularies, tuples, or available resources requires custom solutions that most part of the time, imply downloading large amounts of datasets. Due to the growing number of available datasets this use-case will become more common for many linked data applications. Another frequent problem in the area of Linked Data Analysis, is to provide fast and accurate methods to detect and extract links between datasets. Existing approaches mainly rely on counting *fully qualified domain name* (FQDN) in the-https://www.sharelatex.com/project/55ed874fe8dff2d67afb8032 Linked Data space. However, this method is not accurate, since it is unfeasible to check if all links are truly being described in the *source* and in the *target* dataset. Crawling the Linked Datasets providing up-to-date data, is computationally expensive and requires powerful hardware to scale with the growing cloud.

Given the described challenges, we present *LODVader*[2], a framework which allows the user to explore a new paradigm in analysing and working with linked datasets. In order address and overcome the described issues, *LODVader* was created with following key features:

1. Link Extraction: *LODVader* uses a novel approach to detect and extract links between datasets using Bloom filters (BF) [2]. The extraction if performed on the fly, when the datasets are being streamed.
2. Dataset Statistics: Due to the vast amount of data which is stored in each dataset, it is important to collect statistical information. Accurate statistical analysis of each dataset regarding the top N used properties, links, relations and similarities is performed and made available for further analysis. In order to keep the

[1] `http://lod-cloud.net/#history`
[2] `http://lodvader.aksw.org`

computational and storage overhead to a minimum, the application uses BFs which have proven to work efficiently in the LOD domain (cf. Section 5.1).

3. Visualization: The visualization of the LOD cloud has been proven to be an important tool when dealing with linked data, since the user is able to comprehend the current structure and connectedness of the available linked datasets. Therefore, *LODVader* supports a multi-layer graph visualization interface which visualizes datasets and their respective relations. Moreover, in many cases it is important to identify dataset links which are not connected within the imported dataset cloud. Therefore we introduce the novel notion of the Dark Cloud. Using the above features makes it possible to create a new LOD diagram showing broken links between source and destination datasets. The broken links discovery also relies on BFs.

4. Dataset comparison: In many cases it is important to find similar datasets within the imported dataset cloud. As a proof of concept the Jaccard Index has been chosen for the initial implementation which is used to identify similar datasets.

5. Simple Search Index: Based on the BF a simple search index is created by indexing subjects and objects as BF vectors, thus allowing an efficient access to this data for comparison and search operations. In addition, *LODVader* allows to search and filter datasets or ontologies by subject, property and objects.

6. Support for RDF Streams: *LODVader* supports the ability to deal with RDF streams, which enables the support of different kinds of RDF input sources. Example RDF data sources are RDF dump files, SPARQL endpoints or other RDF data streams.

7. Versioning: The *LODVader* framework supports a simple version control system, which allows the user to perform even deeper analysis of one or more datasets and its development over time.

LODVader is capable of consuming RDF descriptions of datasets according to DCAT [15], VoID [3] and DataID [4]. Based on the description file, the application fetches and consumes RDF datasets by streaming them statement by statement. *LODVader* can further enrich these description files with `void:Linkset`. After the enrichment process, provenance of the newly generated metadata is annotated using the Prov-O ontology [16].

Besides to be able to read and write RDF data, *LODVader* doesn't uses triplestores. Instead, *LODVader* uses a MongoDB as a database to store and fetch relevant data. MongoDB has shown fast and scalable enough to be used with BF as a central index for linked datasets. Given the above-mentioned features, we have high hopes that *LODVader* has great potential to become an important tool for the Semantic Web community.

The remainder of this work is structured as follows: We provide a description of *linksets*, Bloom filters and Jaccard index in Section 2, followed by the *LODVader* overview and 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.

[3]`http://www.w3.org/TR/void/`

2. BACKGROUND

In this section we provide the necessary definitions needed to understand the *LODVader* lifecycle.

2.1 Dataset Metadata Vocabularies

In order to identify which resources should be streamed or processed, this work relies on vocabularies such as DCAT [15], VoID [4] and DataID [4]. These vocabularies are used to represent metadata descriptions of datasets. They provide information about multiple properties of a dataset, including subsets and distributions. A subset is a distinct part of a dataset that can be differentiated for a number of reasons, such as differences in provenance, publication dates, accessibility or language[5]. Distributions describe the specific files or resources by which the datasets might be accessed or acquired[6].

2.2 Linkset Definition

Linksets are RDF descriptions of relations between datasets 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. 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_{ID}: the set of subsets, described by `void:subset` of given dataset ID
- $< s, p, o >$: 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
- $D_{S_{ID}}$: the set of distributions of subset S of dataset ID
- *linkset* $L_{d_1 \to d_2}$: the set of existing links, having d_1 as source distribution and d_2 as target distribution.
- *darkLinkset* $L_{d_1 \to d_2}$: the set of existing bad links, having d_1 as source distribution and d_2 as target distribution.
- NS: The namespace of an URI.

[4]`http://www.w3.org/TR/void/`
[5]`http://www.w3.org/TR/void/#subset`
[6]`http://www.w3.org/TR/vocab-dcat/#class-distribution`

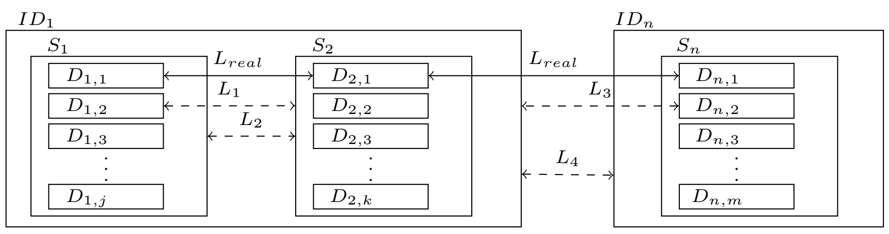

Figure 1: The full arrows (L_{real}) represents links between distributions. The dotted arrows are: L_1 distribution to subset, L_2 subset to subset , L_3 distribution to dataset, L_4 dataset to dataset

We define that a link occurs from a distribution d_1 to a distribution d_2 whenever d_1 contains $t_1 =< s_1, p_1, o_1 >$ and d_2 contains $t_2 =< s_2, p_2, o_2 >$ where $o_1 = s_2$. We then call t_1 a link (regardless of the used property) and say that the distributions are linked with each other (cf. Section 2.6). From this definition it easily follows that *linksets* between distributions (subsets or datasets) can be aggregated in a straightforward manner. 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.

We define as a *darkLinkset* from distribution d_1 whenever $t_1 =< s_1, p_1, o_1 >\in d_1$ and $\exists t_i =< s_i, p_i, o_i >\in d_2 \wedge (NS(o_1) = NS(s_i)) \wedge \nexists t_j =< s_1, p_j, o_j >\in d_2$. An *unverified darkLinkset* from distribution d_1 exists whenever $t_1 =< s_1, p_1, o_1 >\in d_1$ and $\nexists t_2 =< s_2, p_2, o_2 >\in d_2 \wedge (NS(o_1) = NS(s_2))$.

2.3 Link Granularity

The LOD cloud diagram[21] assumes *Pay-Level Domain (PLD)* [13] and sub-domains as the basis for a dataset definition. It consequently only depicts inter-dataset relations as links. *LODVader* also offers visualisation and analysis of intra-dataset relationships, for example between subsets and distributions, featuring a higher *link granularity.* Figure 1 shows an overview of links at different levels of granularity regarding a *linkset* representation. Datasets are represented by ID_n, subsets are represented by S_n and distributions are represented by D_n. L_{real} is a *linkset* containing links between two distributions which are measured on the intersection of subjects and objects (cf. Section 2.2). 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.

2.4 Bloom Filters

Bloom filters (BF) are used create the search indices for the *LODVader* framework and are therefore crucial for the search and analytical component. A 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. Internally BF use a combination of bit arrays and hash functions to store and retrieve data. This data structure has 100% recall and *false negative* (fn) matches are impossible, while a small percentage of *false positives* (fp) are condoned. The work [6, 12] has shown that Bloom filters fit well with search algorithms for large corpora. Furthermore [6, 12] adds that BF have a low memory footprint and search queries can be executed very efficiently. The weakness of Bloom filters is the likelihood of the existance of *false positive* (fp) values. These elements are not part of S, but they are found in the set by the algorithm. However, the fp margin of error can be adjusted in advance. The low probability of *false positives* makes BF viable for many applications in different domains. Independent of the BF set S size, less than 10 bits per element are required for a 1% of fp probability. [3]. The $false\ positive\ probability$ (fpp) is calculated according to the dataset size. Equation 1 defines the fpp value used in our experiments. Our application uses a $configurationfile$ 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).

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

Considering that datasets might contain millions of links, an algorithm with good performance and low memory footprint is fundamental for our application. BF provide us with the desired performance and memory requirements. BF allows us to compare URIs fast and to keep the linksets directly in memory. The process of selecting which BF is loaded in memory at a given time is performed by comparing the NS of the triple's subject and object URIs with all the collections which are stored in a database. Because this data structure offers space and time advantages in our scenario, we prefer it compared to the more commonly used binary search trees, hash tables, arrays or linked lists. A detailed benchmark can be found at Putze's work [20].

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:

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

An optimal number of hash functions is given by:

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

2.5 Jaccard Coefficient

As mentioned above, one of the key features of the *LODVader* framework is the ability to compare two of more datasets with each other. We currently implement the simple Jaccard Coefficient (JC) or Jaccard Index to measure the similarity between two sets. It is defined as:

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|}, \tag{4}$$

where A and B are two sets and the result $JC(A, B)$ is a number between 0 and 1. With the help of the Jaccard coefficient it is possible to measure the overlap of two sets A and B, where both sets do not have to the same size. If both sets are the same, then the result is 1. If both sets are disjoint, then the result is 0.

2.6 Linking Predicates

Common approaches for linking analysis rely on the inspection of the predicates. `owl:sameAs` has well-defined formal semantics and is the predicate which is closest to traditional deduplication. For record linkage or object reconciliation in the database area, counting `owl:sameAs` links exclusively provides a very limited view of the Web of Data and does not provide a reliable model [9].

Several other properties have been proposed with `rdfs:seeAlso` and `skos: { exact | close | broad | narrow | related} Match` being the most common. In our work, we are tolerant and consider all predicates for linking. While for crawling link direction is important – although DBpedia is the largest authority [21], no backlinks are included – we argue that linking properties is often either symmetrical (and highly unlikely to be asymmetric) or it is feasible to assume that an inverse property exists or could be easily created, i.e., following a `birthplace`$\leftrightarrow$`isBirtplaceOf` pattern or simply `birthplace`$^{-1}$.

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

3. LODVADER ARCHITECTURE

In the previous chapters the requirements and the background to the *LODVader* framework have been discussed. This chapter goes into more detail how all the requirements have influenced the architecture of this novel LOD analysis framework.

As one can see in the architecture diagram in figure 2, the framework consists of the following components: *Manager*, *RDF Streaming Module*, *Analytics Backend*, *MongoDB*[7] and a user frontend which is connected via a REST API to the *Manager*. *LODVader* was implemented in Java and uses the Apache Jena[8] library to stream, parse and write RDF data. *MongoDB* is used as a backend storage service for storing non-RDF data such as analytics data, distribution namespaces, Bloom filter results, etc. One can try out *LODVader* following this link `http://lodvader.aksw.org`. The implementation is open source and available on GitHub. [8]

The *Manager* is the central controller of the service and is responsible to serve the API calls and coordinate the processing components. Once a user submits a dataset for processing, the *Manager* dispatches the call to the *RDF streaming module* (cf. subsection 3.1). The output of this step is an array of triples $< s, p, o >$ that is dispached to the *Analytics Backend*. The *Analytics Backend* contains two units: the *Linking Analytics* (cf. subsection 3.2) and the *Dataset Analytics* (cf. subsection 3.3). Using efficient data structures, the analytis results are stored in MongoDB and are available from the *RESTAPI*.

Besides the dataset submission, the *LODVader RESTAPI* communicates with the *Manager* and the database and provides: 1) customizable LOD cloud visualizations, 2) dataset searching based on subject, predicate and objects fields, 3) dataset analytics 4) RDF exports in the *linksets* format, and 5) real time information about the progress of the RDF streaming and the link detection and extraction.

Bloom filters (BF) is a core data structure in *LODVader*. For every distribution that is processed, two BFs are created, one that holds all the subjects of a dataset and one that holds all the non-literal & non blank node objects. The BFs are stored in the MongoDB and are associated with the current distribution. BFs allow fast and reliable response due the possibility of set fpp and the small index size. Therefore they play a crucial part of the search and *Linking Analytics* module.

3.1 Dataset Streaming

In order to get a list of RDF distribution download links, *LODVader* parses the received description file and searches for instances of `dcat:Distribution`. All these distributions are a *source distribution*. The application fetches the `dcat:downloadURL` or `void:dataDump` object. Before the download of the *source distribution* is started, it is checked whether the dataset has already been imported into the system. If the dataset is known, the system reads the *Last-Modified date* and *Content-Length* in the HTTP header to verify whether the dataset has not been changed. If there are modifications, the old data is moved to an archive, in order to use it for versioning reasons. Once the streaming starts, we detect the serialization type, possibly decompress the stream and parse the RDF triples.

3.2 Linking Analytics

For every triple that is streamed, the *Linking Analytics* modules discards the predicate and takes only the subject and the object as input ($< s, o >$). if the object is a literal or a blank node the tuple is discarded. As a final filtering step, we reject tuples with malformed IRIs. The tuples that pass the filtering step, enter a processing pipeline:

1. **Tuple splitting**. *subjects* and *objects* of each tuple are separated and saved in two queues. The queues contain resources which will compared with BFs.

[7] `https://www.mongodb.org/`

[8] `https://github.com/AKSW/lodvader`

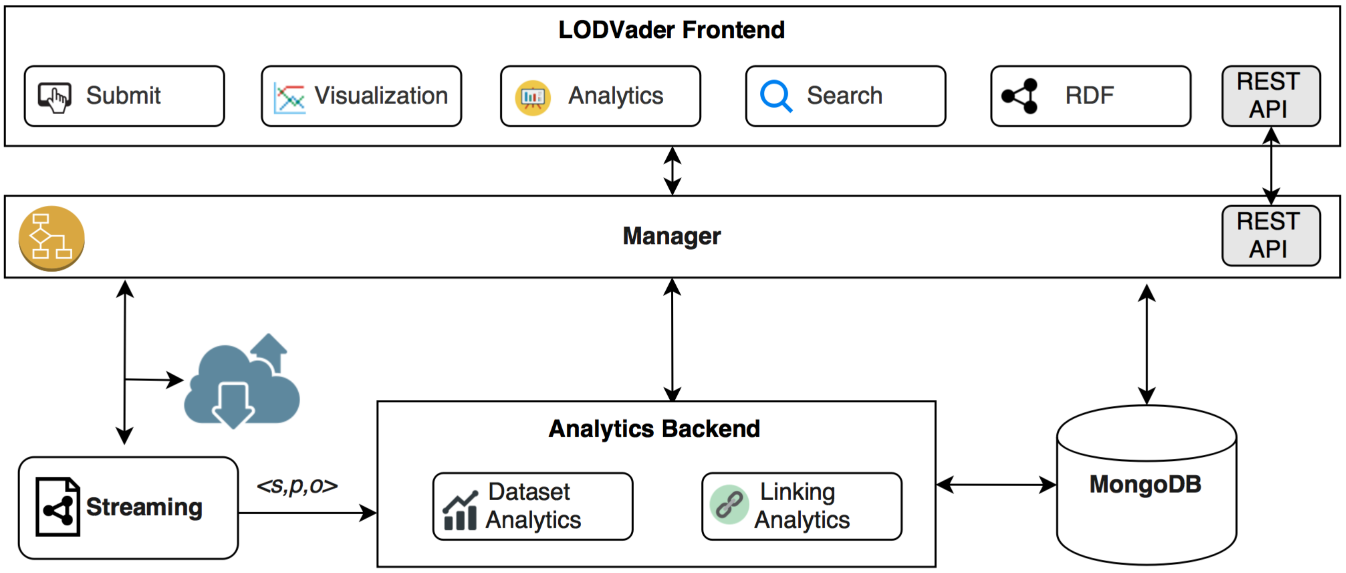

Figure 2: LODVader Architecture

2. **BF Fetching**. we extract the namespace of each resource to compare and assign the resource to a respective BF which will represent a *target distribution*. For every namespace we encounter, we fetch all the existing BFs that are processed and stored in a cache memory.

3. **Link Extraction**. objects and subjects of the *source distribution* are compared with the in-memory BFs of the *target distributions*. If an *object* of the *source distribution* exists in the BF of the target distribution as a *subject* we count one link between the *source distribution* and the *target distribution*. If the opposite way happens, i.e if *subject* of the *source distribution* exists in the BF of the target distribution as a *object* we count one link between the *target distribution* and the *source distribution*. Non-existence is counted as a bad link between the *source distribution* and the *target distribution*, and will be added to the dark cloud.

At the end of the pipeline two new BF are created, one with all subjects and a second containing all objects of the *source distribution*. These BF will represent the current distribution and might be used later when other *sources distributions* are streamed.

3.2.1 Computational Complexity

It is important to understand the impact of using BFs and estimate the computational complexity of discovering links by comparing distributions. The bottleneck of the pipeline is the *link extraction* stage, where the execution is made in parallel and consists of querying a chunk of data (a queue of *subjects* or *objects*) against multiple BFs. The following equation describes the time t to process a chunk of data C, where f_C is the time to fetch from the MongoDB database the Bloom filters the describe the resources of the chunk, l_B is the time to load a set of BFs B from the storage drive.

$$t_C = f_C + l_B + ((m/n)\log 2) \times size(C) \times size(B) \qquad (5)$$

Determining witch BFs should be loaded is done only once, since after loaded to main memory the *linking analytics* module can use it. Each thread holds one Bloom filter, thus the real effort of each thread is determined by the number of hash functions used to compare to the BF times the number of the chunk size:

$$t_C = ((m/n)\log 2) \times size(C) \qquad (6)$$

In other words, equation 6 determines that the time of comparing a chuck of data with a Bloom filter is directly proportional to the number of hashes needed to query the Bloom filter.

3.2.2 Implementation details

In this section we characterize important details regarding the viability and implementation of our approach. Algorithm 1 describes the process of streaming distributions, and Algorithm 2 gives a detailed description of the novel process of how BFs are used to count links.

For each distribution D which has a status that allows it to be streamed (i.e. a new distribution added by a user), a *queue* is created. The *queue* necessary since multiple threads will be consuming triples. The first *thread* t_1 uses an RDF parser to separate s_j and o_j, storing in the *tupleQueue*. *threads* t_2 fetches the BFs needed to compare with the *tupleQueue*. The *threads* t_3 uses the *tupleQueue* s_j and o_j to extract links comparing with BFs. The *extractLinks* function is described in Algorithm 2. 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 triples and the Bloom filter and the statistical data for *subjects*, *predicates* and *objects* are created.

Notice that only after streaming a distribution, the BFs for this particular distribution will be created. It is not possible to create BFs on the fly because it is necessary to know how many URIs will be stored in the filter. In algorithm 2 namespaces (NS) are compared and only the necessary BFs

Algorithm 1 Streaming new distribution

```
1: for all $D$ which should be streamed do
2:     $queue \leftarrow new\ queue()$;
3:     $tupleQueue \leftarrow new\ queue()$;
4:     $th_1 \leftarrow$ new Thread(SPLITTUPLE($tupleQueue$,$queue$);
5:     $th_2 \leftarrow$ new Thread(FETCHBF($resourceQueue$);
6:     $th_3 \leftarrow$ new Thread(EXTRACTLINKS($resourceQueue$));
7:     while $streaming(uncompress(D_i))$ do
8:         for all $s_j, p_j, o_j \in D_i$ do
9:             $queue.add(s_j, p_j, o_j)$
10:        end for
11:    end while
12:    $fpp \leftarrow calculateFPP(th_1.getNumberOfTriples())$
13:    CREATEBLOOMFILTER($subjects$,$objects$,$fpp$);
14:    CREATEANALYSIS($subjects$,$predicates$,$objects$);
15: end for
16:
```

are loaded and the resources are effectively used to be compared with BFs. In Algorithm 2, for each distribution D a set P is created which contains the intersection of namespaces from the $queue_i[objects]$ and $D[subjects]$. Having P it is possible to know which BFs that might contain links with the $queue_i$ should be loaded to a set of $threads$ (notice that if BF is not cached they must be loaded to the main memory, and the equation 5 applies). Next, a $buffer$ is created containing the objects from the current distribution. The $threads$ of the set are started using the $buffer$ of objects to compare with the previously loaded Bloom filters. If BFs return true values, a link in incremented, else, a bad (or an unverified bad) link is incremented. Essentially, in this manner, *LODVader* computes the links for the LOD cloud and for the Dark LOD could.

It is important to stress that our model reads and retrieve, however does not store any RDF data. Our implementation creates RDF on the fly reading documents from MongoDB and using Apache Jena[8] to create RDF models.

3.3 The Analytics Module

LODVader allows to fetch analytical data through the Analytics Module. We address four different types of dataset analysis:

1. **TOP N most similar distibutions**. It's possible to compare distributions using three different of approaches: Similarity by `rdf:type`, `rdfs:Class` or by all predicates.
2. **TOP N most linked distributions**. *LODVader* retrieves information about the top N distributions linked with each other.
3. **TOP N predicates**. *LODVader* retrieves different types of statistical data regarding to a specific distribution. This include information about the top N most used predicates, `rdf:type` and `owl:Class`.
4. **General distribution details**. *LODVader* retrieve general details of the streamed distributions, including the number of triples, serialization format, and the status of the distribution by checking whether the resource is available to be downloaded or not.

Algorithm 2 Extract links

```
1: function EXTRACTLINKS($queue$)
2:     $P \leftarrow NS(queue_i[objects]) \cap NS(\forall D[subjects] | queue_i \notin D)$
3:     for all BF $\in P$ do
4:         $threads[i] \leftarrow$createNewThread($BF_i$);
5:         $i \leftarrow i + 1$;
6:     end for
7:     while $buffer = read(queue_i[objects])$ do
8:         for all threads do
9:             $results \leftarrow compare(buffer, thread_k)$;
10:        end for
11:        for all result $\in results$ do
12:            if result then
13:                $incrementLink()$
14:            else
15:                if $badLinkVerified(result)$ then
16:                    $incrementBadLink()$
17:                else
18:                    $incrementUnverifiedBadLink()$
19:                end if
20:            end if
21:        end for
22:    end while
23: end function
```

The data regarding of the TOP N features are directly stored in the MongoDB database. Hence, the search and calculation are flexible allowing the creation of further queries w.r.t statistical data. The similarity comparison between distributions is calculated with Jaccard Coefficient. Our approach enables the possibility to compare two distributions side-by-side in respects to the data retrieved from the similarities values.

3.4 The Search Module

LODVader takes advantage of Bloom filters and provides a search engine that allows users to find datasets querying by subjects, predicate or objects. The lifecycle of the search engine consists in three stages:

1. **Extracting Namespaces**. In order to fetch which datasets describe a particular URI, the first step is to extract its namespace. Notice that this process is done only for search by *subjects* or *objects*.
2. **Loading Bloom filters**. After extracting the namespace of the URI, the application will fetch MongoDB for Bloom filters which describe the namespace. Then, for BF that had never been loaded before, they are loaded to a cache memory that will allow the application to further access them more quickly.
3. **Querying Bloom filters**. The last step consists in querying the original URI against the loaded Bloom filters. Evidently that BF that represents subjects or objects must be chosen according with the type of query. If BF returns a positive value, means that the dataset associated with this BF is describing the URI.

3.4.1 Computational complexity

We can estimate the computational effort needed for the search engine. The equation 5 is applicable in this case. However, there is no chunk size since the query is only a URI, and the Bloom filters are only loaded once. Thus, equation 5 is simplified to:

$$t_C = ((m/n)\log 2) \times size(B) \qquad (7)$$

This is the main reason for the search engine to be fast. The query is made against Bloom filters, which means that only a few hashes are created, making no graph search or intensive databases query, keeping the search engine simple and fast.

3.5 The LOD Diagram

Our API provides JSON objects which characterize the current condition of the links between the distributions. Hence, it provides a new visualization of the LOD-Cloud diagram using multiple layers of data, i.e. distributions and subsets are within datasets and are represented by different edges of the graph. An example can be seen in figure 3. In addition to the classical view, where vertexes represents the amount of links between datasets, the *LODVader* framework allows different dataset visualizations which can help the user to perform analysis operations on the imported LOD cloud. For example it is possible to filter datasets based on their similarities, by datasets and ontologies. Furthermore, subsets and distributions of a dataset are shown and grouped as clusters of bubbles.

Moreover *LODVader* introduces the concept of the Dark LOD diagram. The Dark LOD diagram, which can be seen in figure 4 and visualizes links between objects of a *source distribution* which are not described as subjects in a *target distribution*. We distinguish two different types of these links: 1. The first type is called *bad link* and it can be annotated as such, if all the dataset distributions have been imported into the system. 2. A *unknown link* is similar to the *bad link*, but in this case not all of the distributions of a dataset are loaded. Hence it is not possible to annotate the link as a bad link, since one of the not imported dataset might contain a link to the currently unconnected object.

In order to distinguish between *bad links* and *unknown links* the metadata information of a dataset can be used, since it describes all the distributions which belong to a dataset. For simplicity reasons, the current version of the framework treats both types the same and we are planing to introduce this distinction in a future release. Furthermore, this feature can help a user to find out what part of the LOD cloud needs refinement, since *bad links* exist. In addition to the cited features, *LODVader* contains archive data for the versioning purpose. This makes it possible to follow the diagram growth over time.

4. EVALUATION

In order to demonstrate the performance of the *LODVader*, we first compared indexing data using BF with HashMap Search (HS) and Binary Search Tree (BST) w.r.t. memory usage for each structure to index distributions. Second, we measured the number of false positives generated by queries against BF. Thirdly, we compared *LODVader* with Open-

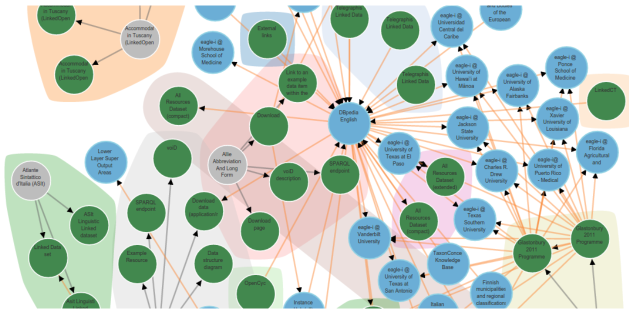
Figure 3: LODVader LOD cloud

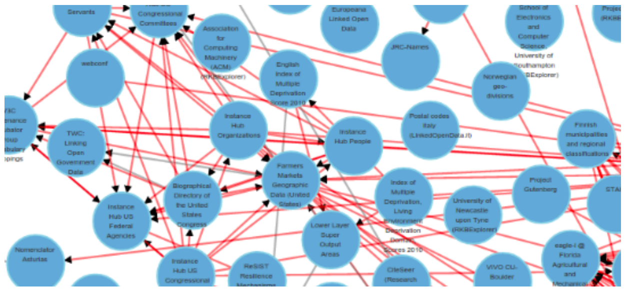
Figure 4: LODVader Dark LOD cloud

Link Virtuoso[9] w.r.t. time to load and index triples, time to make searches, amount of data on the storage. Then we check how well the framework can be used to find relevant dataset distributions in the imported LOD cloud.

All experiments were made using a Intel(R) Core(TM) i7-5600U @ 2.6GHz, 16GB DDR3 and SSD drive.

4.1 Index Data Structure

The results (Figure 5) show 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 5 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 then 34MB of memory, performing on average 12 times better than HS and 10 times better than BST.

4.2 Quantitative Evaluation

As shown in the background section 2, BFs can produce false positives (FP). This section verifies the impact of BF false positives by conducting five experiments. In order to measure the *tp*, and compare with BF, we used BST. For the first experiment, we added the 48 English DBpedia distributions[10] and counted links among them. Then, we added all 358 vocabularies available in LOV[11], and again we counted

[9]http://virtuoso.openlinksw.com/
[10]http://downloads.dbpedia.org/3.9/en/
[11]http://lov.okfn.org/lov.nq.gz

Dataset	Triples	*tp*	*fp*	Precision	F-Measure
English DBpedia	812M	9.013.302.727	1.354	0,99999985	0,999999925
LOV Vocabularies	862K	31.027.759	77	0,99999752	0.999998759
Routers128	7k	2501	0	1	1
RSS-500	10k	1265	0	1	1
Brown Corpus	3.4M	890.760	3	0.99999663	0.999998316

Table 1: BF Index Precision

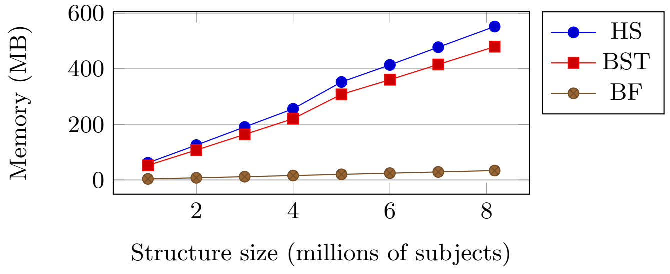

Figure 5: Memory usage per indexed resource

Parameter	Virtuoso	*LODVader*	Performance
Load time	24:02:36h	06:32:01h	3.67x faster
Disk usage	84,28Gb	4,03Gb	95.2% less pace

Table 2: Triple load time and disk usage of *LODVader* compared to Virtuoso. Virtuoso outperforms *LODVader*

the links among the vocabularies and DBpedia distributions. In the last three experiments, we added three NIF[10] corpora: Reuters128[12], RSS-500[13] and Brown Corpus[14] (which contain 123 distributions). We measured the false positive rate and the time to count links. All Bloom filters were created with *fpp* described in Equation (1). The results are shown in Table 1. For DBpedia, up to 812 million triples in 48 distributions were analyzed. There are more than 9 billion 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). Hence, if a DBpedia URI occurs as an object, it is highly likely to have an intersections with most of the distributions. In the second experiment, we added up to 862.000 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. 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.

Apart from measurig the precision of BFs, we used OpenLink Virtuoso to compare the performance in other aspects. We are aware that *LODVader* is not a triplestore, and do not store sufficient data do make a SPARQL query. However, making complexes queries is out of the scope of our approach. Moreover, considering the huge gain in the matter of performance of making simples searches, and storage space, *LODVader* is appropriate to index and search large amount of RDF data.

[12]https://github.com/AKSW/n3-collection/blob/master/Reuters-128.nt
[13]https://github.com/AKSW/n3-collection/blob/master/RSS-500.nt
[14]http://brown.nlp2rdf.org/

	LODVader	Virtuoso	Difference
Subjects only	3.813s	2.221s	1.7x faster
Properties only	1.158s	0.540s	2.1x faster
Objects Only	5.468s	3.738s	1.5x faster
Triples	4.158s	2.540s	1.6x faster

Table 3: Average search performance of *LODVader* compared to Virtuoso

To compare the efficiency of *LODVader* with OpenLink Virtuoso, we loaded 491 datasets with a total of 1,092,182,333 triples. There is a slightly difference between the number of loaded triples on *LODVader* and Virtuoso. *LODVader* loaded 1,092,182,333 triples and Virtuoso 1,034,808,229. This difference is due the fact that *LODVader* stores triples regardless whether they are repeated or not, dealing with all triples as unique. On the other hand, triplestores will not store the same triple twice, considering that it's a graph structure. The problem is emphasized since each framework deals differently with erroneous data. For example, a bad RDF structure might be accepted by a particular framework, but completely ignored by others. Hence, when we deal with large amount of triples, it's difficult to have the exact number of resources in different frameworks.

Table 2 shows the results w.r.t for loading and indexing triples and the total space used in the hard drive. OpenLink Virtuoso loaded triples in 24:02:23, while *LODVader* 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 *LODVader*.

The search performance is shown in Table 3. First, we made 800 queries searching for particular subjects, predicates and objects against and then searching by particular triples. Again, we compared *LODVader* with OpenLink Virtuoso. All experiments regarding of search speed, *LODVader* was slower then OpenLink Virtuoso. For subjects queries, there was a difference of 1.7x of speed, for properties 2.1x, for objects 1.5x, and for searching full triples the difference was 1.6x. Despite the difference of speed, the *LODVader* compensates the performance on the disk utilization. The list of the the used datasets is available at our web page[15].

[15]https://github.com/AKSW/lodvader

4.3 Similarity Evaluation

In this section we want to evaluate how well the framework can be used to find similar or related dataset distributions within the imported LOD cloud. Therefore, we picked 2 dataset distributions: *RSS-500 NIF NER CORPUS* and *Public Spending in Greece (2012 4th Quarter)*. In order to identify whether a dataset distribution is similar to other ones, we used the Jaccard Coefficient (based on predicates) and the number of links which are shared between two distributions. These are the results we found for each distribution:

1. RSS-500 NIF NER CORPUS: For this distribution we were able to identify two other matches. The first one is the *Reuters-128 NIF NER Corpus* distribution. Here the Jaccard Coefficient is *0.47* and the number of shared links is 2501. The second match is *NLP Interchange Format vocabulary*. Here the JC is 0.025 and the number of shared links is 2501. The fact that the first match has a higher similarity than the second one is not a surprise, since the second distribution only contains information about the actual *NIF* format and both other distributions store actual *NIF* annotation data.
2. Public Spending in Greece (2012 4th Quarter): We were able to find the 4 other distributions which belong to the same dataset. It was interesting to see that some distributions had a high similarity of around 0.5 and a low link count. One distribution had a low similarity value and a very high link count. This shows that both values are important factors.

These results essentially show that searching distributions with high similarity value, contains comparable structure, and high number of linkets the datasets contains complementary data. Using the feature consists in a advantageous way to find possible datasets that can be used for enrichment purposes.

4.4 Discussion

We evaluated *LODVader* in three different aspects: Firstly by comparing the performance differences between BF with HS and BST datastructures. Secondly by performing a quantitative evaluation regarding fp, in a dense scenario like DBpedia. Thirdly by comparing performance of hard drive utilization and query time. In fact, all three evaluations indicate that *LODVader* is a good choice for the creation of a central index of Linked Datasets. The main advantage of *LODVader* over regular triplestrores is the disk space utilization and good performance regarding memory utilization by reason of the utilization of BF.

We compared OpenLink Virtuoso w.r.t. stored data efficiency and search time. *LODVader* stored data 20 times more efficient and is over 3 times faster indexing the same RDF data. However, *LODVader* does not store sufficient data to create SPARQL queries, making our framework limited for detailed queries. Therefore, the scope of our approach is to provide details about the relations between datasets, and not perform a deep and detailed search on the LOD graph. This is the main difference between *LODVader* and a regular triplestore.

The Jaccard Coefficient has shown a reasonable approach to compare distributions. The outcome of the similarity test looks very promising and shows the great potential of this framework. It can be used to find related dataset distributions quite easily. In regards to our false positive results, we will have to perform an error analysis and the system in a future release.

5. RELATED WORK

LODVader is a novel framework, thus it can be considered as a hybrid between different LOD cloud analysis applications. In order to make it easier for the reader to relate the different parts to each other, we have divided the related work section into different sub-sections.

5.1 Bloom Filters on Linked Data

For Linked Data, Bloom filters have been used in several works in the LOD domain. As an example, in [22], Bloom filters are used as a SPARQL extension for testing blank nodes membership. In addition the paper shows that it is possible to notice 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 [11] 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 [18] present an approach for answering queries through an evolutionary search algorithm which uses Bloom filter for rapid approximate evaluation of generated solutions. Bloom filters do not allow elements removal from the set, thus, variations of the original Bloom filter have been used such in [19] where the author chooses Counting Bloom filters to filter non-matches candidates to a query. In the light of these results Bloom filters were chosen to create a simple and fast index with a small memory footprint which serves the requirements of many LOD cloud use cases.

5.2 Linked Open Data Cloud Diagrams

The LOD cloud diagram [21] is one of the main motivation for this work. One of the main disadvantages of the LOD cloud diagram is the fact that it is based on the assumption that every distribution in the same pay-level domain or subdomain belongs to the same dataset. Therefore the LOD cloud diagram currently fails to generate a precise visualization of the existing LOD cloud datasets. Since an update of the LOD cloud diagram is only published every couple of years, the actual state of the diagram is unclear. Finally the inability to filter on dataset is a problem, since the diagram is a static image which does not allow any significant user interactions. Due to the fast growth rate of the LOD cloud the highlighted issues will gain in importance in the next couple of years.

LODLive [5] is an environment that queries predefined SPARQL endpoints. This work supports dynamic filter query generation, however *linksets* are only shown within resources of a dataset. In addition they are not extensible to others datasets or subsets. Since it is not possible to query multiples of datasets, LODLive does not provides methods for link discovery.

Since it is widely accepted that visualizations have an undeniable advantages when dealing with LOD dataset data, *LODVader* uses LOD dataset visualizations as a central key feature of the application and it addresses the above mentioned issues.

5.3 RDF Analytics

Due to strong growth of the LOD cloud it is obvious that there is a demand for LOD cloud analytical frameworks. Some statistical information can be found together with the LOD cloud diagram [21]. Unfortunately the statistical information are also static. The framework loupe [16] goes a step further. It gives many different statistical information about datasets and it allows the user to find information about classes, properties, the ontology and much more. In addition, it is combined with a simple visualizer for ontology classes. Although it creates a great user interface, it focus on s selected dataset and does not allow the comparision between different datasets within the LOD cloud.

Another good example is Aether [17]. It supplies the user with many different statistical information for datasets when supplied with a SPARQL endpoint address. It is even possible to compare different SPARQL endpoints, which can be useful if two different endpoints should be analyzed. Although this framework supplies the user with great statistical information and pie charts, it is only developed for comparing the content between two SPARQL endpoints.

LOD-Laundromat[1] provides an uniform way to publish and clean datasets. Different statistical data is publised, like duplicated triples, amount of triples, dataset size and other. The LOD-Laundromat contains over 38 bilio of triples, however the issue is that they do not provide metadata regarding dataset labels, name or title, making the whole graph visualization a hard task.

As one can see from the related work, many different solutions were developed to address different issues. To our knowledge no other LOD framework tries to address all of the use cases which are addressed with this application. No other application brings together an intuitively usable LOD cloud visualizer, a fast search and analytics interface, which offers the processing of unprecedented LOD use-cases which were not possible before.

5.4 Search for the Semantic Web

Search engines for the Semantic Web have increased in importance over the last couple of years. In [7] the authors give an overview about the different type of Semantic Web Search engines and they group them into two main categories. The first category covers Ontology Search Engines which can be used to find matching ontologies and datasets. The second category covers Semantic Search Engines which are extensions of regular text based search engines. They use semantic information to increase the search accuracy of the system. The *LODVader* system belongs to the first category, since *LODVader* can be used to find datasets within an imported group of LOD datasets. When it comes to datasets, it is possible to use a corpus of datasets metadata or a corpus of a RDF ontology data to perform a search. Examples of a framework which uses metadata information of datasets is DataHub [17]. *LODVader* currently uses some metadata information (i.e. title property, distribution link), but it can easily be extended to search for more. An example for a search engine, which uses an ontology corpus is swoogle [14]. *LODVader* is also capable of performing a search across subjects, predicate and objects (and a combination of them) of dataset. On top of that *LODVader* is also capable of finding similar datasets. This feature is particulary useful, when dealing with a large set of datasets.

[16]http://loupe.linkeddata.es/loupe/

[17]http://datahub.io/

6. CONCLUSIONS

In this paper we present a LOD framework which combines visualization, search, versioning and analytics components in a novel way. Over the last chapters we were able to show that this framework supports many important features for dealing with the LOD cloud, but as the conducted experiments have shown, it outperforms many state of the art approaches and systems. *LODVader* was developed to be a central index for Linked Data datasets. For the efficient comparison of datasets we used a probabilistic data structure called Bloom filter which is used to count links between them and to create a scalable implementation of a link analysis between dataset distributions. For discovery of similarities of the datasets, we used Jaccard Coefficient on `rdf:type`, `owl:Classes` and general predicates. Moreover we presented a LOD cloud which contains all the valid links among distributions and a Dark LOD diagram which shows the links which are not being described. In addition, the *LODVader* search module is able to quickly find datasets based on their subject, predicate and subjects.

In this initial proof-of-concept implementation, we have shown that the performance of *LODVader* with over one billion triples has proven to be a feasible approach. *LODVader* provides the user the chance to find similar datasets, and more than that, it can be used to find datasets with complementary data. We are providing up to date measurements and diagrams on `http://lodvader.aksw.org`

We are exploring the possibility of using *LODVader* not only as a standalone service, but as an interface with a DataID SPARQL endpoint to centralize and enrich DataID files. Moreover, we are aware that only accepting dump file is a limitation to our approach, future work will include reading SPARQL and LD Fragments endpoints to get the data from distributions. In order to ensure the quality of the imported data, we are planing to evaluate the intregration of the laundromat [1] into our pipeline.

Acknowledgement. This paper's research activities were funded by grants from the FP7 & H2020 EU projects ALIGNED (GA 644055), and LIDER (GA-610782) and the CAPES foundation (Ministry of Education of Brazil) for the given scholarship.

7. REFERENCES

[1] W. Beek, L. Rietveld, H. Bazoobandi, J. Wielemaker, and S. Schlobach. Lod laundromat: A uniform way of publishing other people's dirty data. In P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble, editors, *The Semantic Web – ISWC 2014*, volume 8796 of *Lecture Notes in Computer Science*, pages 213–228. Springer International Publishing, 2014.

[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*, 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] K. Esmaili and H. Abolhassani. A categorization scheme for semantic web search engines. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 171–178. IEEE, 2006.

[8] M. Grobe. RDF, Jena, SparQL and the 'Semantic Web'. In *Proceedings of the 37th Annual ACM SIGUCCS Fall Conference*, SIGUCCS '09, pages 131–138, New York, NY, USA, 2009. ACM.

[9] 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, L. Z. Mika, Peter, J. Z. Pan, I. Horrocks, and B. Glimm, editors, *International Semantic Web Conference (1)*, volume 6496, pages 305–320. Springer, 2010.

[10] 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*, pages 98–113. 2013.

[11] 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.

[12] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[13] O. Lehmberg, R. Meusel, and C. Bizer. Graph Structure in the Web: Aggregated by Pay-level Domain. In *Proceedings of the 2014 ACM Conference on Web Science*, WebSci '14, pages 119–128, New York, NY, USA, 2014. ACM.

[14] D. Li, F. Tim, J. Anupam, P. R. Rong, C. Scott, P. Yun, R. Pavan, D. Vishal, and S. Joel. Swoogle: a search and metadata engine for the semantic web. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 652–659. ACM, 2004.

[15] F. Maali and J. Erickson. Data Catalog Vocabulary (DCAT). W3C recommendation, W3C, Jan. 2014.

[16] D. McGuinness, T. Lebo, and S. Sahoo. PROV-O: The PROV Ontology. W3C recommendation, W3C, Apr. 2013.

[17] E. Mäkelä. Aether – generating and viewing extended void statistical descriptions of rdf datasets. In *Proceedings of the ESWC 2014 demo track, Springer-Verlag*, 2014.

[18] 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.

[19] S. Paturi. *A new filtering index for fast processing of SPARQL queries*. PhD thesis, Faculty of the University Of Missouri-Kansas City, 2013.

[20] 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.

[21] 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.

[22] G. T. Williams. Supporting Identity Reasoning in SPARQL Using Bloom Filters. In *Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*, 2008.