On Measuring the Lattice of Commonalities Among Several Linked Datasets

Michalis Mountantonakis and Yannis Tzitzikas
Institute of Computer Science, FORTH-ICS, GREECE
Computer Science Department, University of Crete, GREECE
{mountant | tzitzik} @ics.forth.gr

ABSTRACT

A big number of datasets has been published according to the principles of Linked Data and this number keeps increasing. Although the ultimate objective is linking and integration, it is not currently evident how *connected* the current LOD cloud is. Measurements (and indexes) that involve more than two datasets are not available although they are important: (a) for obtaining *complete information* about one particular URI (or set of URIs) with provenance (b) for aiding *dataset discovery* and *selection*, (c) for *assessing the connectivity* between any set of datasets for quality checking and for monitoring their evolution over time, (d) for constructing *visualizations* that provide more informative overviews. Since it would be prohibitively expensive to perform all these measurements in a naïve way, in this paper we introduce indexes (and their construction algorithms) that can speedup such tasks. In brief, we introduce (i) a namespace-based prefix index, (ii) a sameAs catalog for computing the symmetric and transitive closure of the `owl:sameAs` relationships encountered in the datasets, (iii) a semantics-aware element index (that exploits the aforementioned indexes), and finally (iv) two lattice-based incremental algorithms for speeding up the computation of the intersection of URIs of any set of datasets. We discuss the speedup obtained by the introduced indexes and algorithms through comparative results and finally we report measurements about connectivity of the LOD cloud that have never been carried out so far.

1. INTRODUCTION

The ultimate objective of LOD (Linked Open Data) is linking and integration, for enabling discovery and integrated query answering and analysis. However, even some basic tasks are nowadays challenging due to the scale and heterogeneity of the datasets: the LODStats website provides statistics about approximately ten thousand discovered linked datasets until August 2014[1], and it keeps growing. To fill this gap, in this paper we focus on methods, supported by special indexes, for performing tasks and measurements that involve more than two datasets. Such indexes and measurements are important: (a) for obtaining *complete information* about one particular URI (or set of URIs) with their provenance, (b) for obtaining measurements that are useful for *dataset discovery* and *selection* [6, 17, 18], (c) for *assessing the connectivity* between any set of datasets for *quality checking* and for *monitoring their evolution* over time [14], (d) for constructing *visualizations* [3] that provide more informative overviews and could also aid dataset discovery. Overall, the aforementioned tasks can assist data scientists since according to several studies they currently spend most of their time in collecting and preparing unruly digital data.

An example of task (a) follows. Suppose that one user or an application wants to find all the available data associated with "http://www.dbpedia.org/Aristotle", including URIs being `owl:sameAs` with this entity (i.e., object coreference), coming from multiple sources. This is not currently possible. With the proposed approach this is possible, and one can also get the provenance of the returned triples. This task is not trivial since the `owl:sameAs` relationships model an equivalence relation and therefore its transitive closure has to be computed and this presupposes knowledge of all datasets. Dataset discovery and visualization, i.e. tasks (b) and (d), are emerging challenges for the web of data; for instance according to [8] "Being able to discover data from other sources, to rapidly integrate that data with one's own, and to perform simple analyses, often by eye (via visualizations), can lead to insights that can be important assets." Currently the community uses catalogs that contain some very basic metadata, and diagrams like the Linking Open Data cloud diagram[2], as well as LargeRDFBench[3] (an excerpt is shown in Figure 1). These diagrams illustrate how many links exist between *pairs* of datasets, however they do not make evident if *three or more* datasets share any URI or Literal! To fill this gap in this paper we show how we can make efficiently measurements that involve more than two datasets. The results can be visualized as Lattices, like that of Figure 2 which shows the lattice of the four datasets of Figure 1. From this lattice one can see the number of common real world objects in the triads of datasets, e.g. it is evident that the triad of *DBpedia*, *GeoNames* and *NYT* shares 1,517 real world objects, and there are 220 real world objects shared in all four datasets. Instead, the classical visualizations of the LOD cloud, like that of Figure 1, stop at

[1]http://stats.lod2.eu

[2]http://lod-cloud.net/

[3]http://github.com/AKSW/LargeRDFBench

This is a preprint of the article: Michalis Mountantonakis and Yannis Tzitzikas, On Measuring the Lattice of Commonalities Among Several Linked Datasets. Accepted for publication in Proceedings of Very Large Databases Conference (PVLDB) and will be presented in New Delhi, India 05/09/2016-09/09/2016.
.

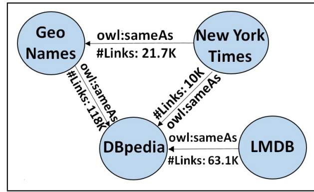

Figure 1: LargeRDFBench Cross Domain Datasets

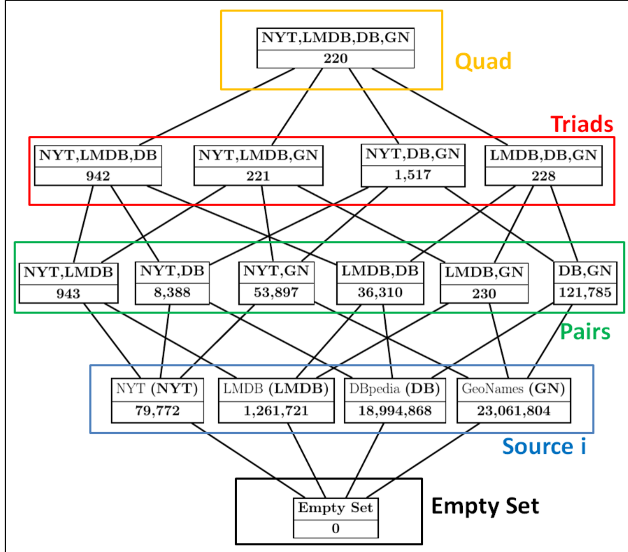

Figure 2: Lattice of four datasets showing the common Real world Objects

the level of pairs. As another example in the lower right corner of Figure 3 we show a visualization of 287 RDF datasets where each dataset is illustrated as a building and bridges are used to illustrate `owl:sameAs` relationships between two datasets where the volume of each bridge is equal to the number of such relationships. If the transitive closure is not computed then the visualization will suffer for missing bridges (i.e. missing connections between datasets) and smaller in size bridges. As *dataset discovery* is concerned, i.e. task (b), apart from visualization the proposed measurements can be directly used for answering queries like "find the K datasets that are more connected to a particular dataset". It is also important to be able to retrieve information about the same real world entity from several sources in order to verify or clean that information and eventually produce a more accurate dataset or "widen" the information for a URI. Technically, this reduces to constructing a dataset with high "pluralism factor", i.e., a consolidated dataset where the number of datasets that provide information about each entity is high. An indicative query for such a case follows: "find the K datasets that maximize the pluralism factor of the entities of a particular dataset".

As regards RDF data quality, i.e. task (c), according to [4] is important for determining the subset of the available data which are trustworthy to use in a Linked Data application while [25] categorizes quality metrics in various categories. The major dimension in which our work (common real world objects) belongs to is mainly *interlinking* and secondarily *relevancy*.

However, it is very expensive to perform all these measurements straightforwardly, because there are numerous datasets and some of them are quite big. Moreover, the possible combinations of datasets is exponential in number. To tackle this challenge, we introduce a set of indexes (and their construction algorithms) for speeding up such measurements. We show that with the proposed method it is feasible to perform such measurements even with one machine! More than one machines could be included for further speedup but this is not necessary since the task of measurement is not a daily activity. Nevertheless, the introduced method and its experimental analysis paves the way for effectively parallelizing the task. In brief in this paper:

- we introduce a *namespace-based prefix index* for speeding up the computation of the metrics,
- we introduce a *sameAs catalog* for computing the symmetric and transitive closure of the `owl:sameAs` relationships encountered in the datasets (the algorithm is based on incremental signatures allowing each pair of URIs to be read only once),
- we introduce a *semantics-aware element index* (that exploits the previous two indexes), and *two lattice-based incremental algorithms* for speeding up the computation of the intersection of URIs of *any set* of datasets (the lattice can be also used for the visualization of commonalities if the number of datasets is low, and as a navigation mechanism if the number of datasets is high),
- we measure the speedup obtained by the proposed indexes and algorithms (just indicatively they enable computing the sameAs closure of 300 datasets in 45 seconds and the lattice of measurements of all possible sets of datasets that share more than 30 URIs in 3.5 minutes and we report connectivity measurements for a subset of the current LOD that comprises 300 datasets.

We have published the measurements in *datahub.io*[4] using VoID [12] and VoIDWH [14] vocabularies, while **`http://www.ics.forth.gr/isl/LODsyndesis/`** offers a set of query services for dataset discovery and global entity lookup, plus a link to a prototype that exploits these measurements and provides an interactive 3D visualization.

The rest of this paper is organized as follows: §2 provides the required background, and describes related work, §3 states the problem, introduces the indexes and their construction algorithms, and §4 shows how to compute the metrics for any set of datasets. §5 reports measurements and experimental results over a big number of datasets of the LOD cloud, and discusses the speedup obtained with the introduced indexes. Finally, §6 concludes the paper and identifies possible directions for future research.

2. BACKGROUND AND RELATED WORK

2.1 Background

The Resource Description Framework (`RDF`) [13] is a graph-based data model for linked data interchanging on the web. RDF uses `Triples` in order to relate Uniform Resource Identifiers (`URIs`) or anonymous resources (`blank nodes`) where both of them denote a `Resource`, with other `URIs`, `blank nodes` or constants (`Literals`). Let $\mathcal{U}$ be the set of all URIs, $\mathcal{B}$ the set of all `blank nodes`, and $\mathcal{L}$ the set of all Literals. A *triple* is any element of $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$,

[4]http://datahub.io/dataset/connectivity-of-lod-datasets

while an *RDF graph* (or dataset) is any finite subset of $\mathcal{T}$. Linked Data refers to a method of publishing structured data, so that it can be interlinked and become more useful through semantic queries, founded on HTTP, RDF and URIs. The linking of datasets is essentially signified by the existence of common URIs, referring to schema elements (defined through RDF Schema[5] and OWL[6]), or data elements. Since we focus on the second, hereafter we will consider only URIs that are either *subjects* or *objects* of triples whereas we will ignore properties, since they are schema elements. However, we do consider `owl:sameAs`-triples, where `owl:sameAs` is a built-in OWL property for linking an individual to an individual meaning that both are equivalent i.e. they refer to the same real world object.

2.2 Related Work

Measurements in LOD Scale. The authors of [20] indexed thirty eight billion triples from over six hundred thousand documents, for cleaning them and providing statistics about the cleaned data. A key difference in our approach is that we take into account the semantics, specifically the `owl:sameAs` relationships. Moreover, our measurements concern connectivity, while [20] focuses on other aspects (like validity of documents) and they do not compute common URIs between three or more datasets. The authors of [15] created a portal for link discovery which contains mappings between pairs of 462 datasets. In comparison to our approach, they take into account only pairs of datasets and they do not index the URIs.

The authors in [21] focused on crawling a large number of datasets, and categorizing them into eight different domains (such as publications, geographical, etc.) and they provided measurements like the degree distribution of each dataset (how many datasets link to a specific dataset). Another work that analyzes a large number of linked datasets is LODStats [2] which provides useful statistics about each document such as the number of triples, number of `owl:sameAs` links and so forth. Comparing our work with the previous two approaches, we focus on connectivity of datasets URIs (meaning that we provide more refined measurements) and we measure the connectivity among two or more datasets. In [14] the authors performed measurements among more than two datasets but for small in number datasets (7 in number). The method that was used for performing these measurements (plain SPARQL queries) cannot be scaled up to large in number datasets. Finally, the work [23] focuses only on features of the semantic web schemas, not on datasets.

Indexes for search and queries. The work described in [9] proposes an object consolidation algorithm which analyses inverse functional properties in order to identify and merge equivalent instances in an RDF dataset. An index for the `owl:sameAs` relationships is adopted in order to find all the URIs for a real world object. This index is used in *YARS2* [7], a federated repository that queries linked data coming from different datasets. The system uses a number of indexes, such as a keyword index and a complete index on quads in order to allow direct lookups on multiple dimensions without requiring join. *YARS2* is part of the Semantic Web Search Engine (*SWSE*) [10] which aims at providing an end-to-end entity-centric system for collecting, indexing, querying, navigating, and mining graph-structured Web data. Another system is *Swoogle* [5] which is a crawler-based indexing and retrieval system for the semantic web. It analyzes a number of documents and provides an index for URIs and character N-Grams for answering user's queries or compute the similarity among a set of documents. A lookup index over resources crawled on the Semantic Web is presented in [24]. In particular, the authors construct an index for finding for each URI the documents it occurs, a keyword index and an index that allows lookup of resources with different URIs identifying the same real world object. The work [16] describes an engine for scalable management of RDF data, called *RDF-3X* which is an implementation of SPARQL [19]. This system maintains indexes for all possible permutations of an RDF triple members (s p o) in six separate indexes while only the changes between triples instead of full triples are stored. In comparison to our work, we mainly focus on finding how connected are the different subsets of the LOD Cloud and how to perform faster such measurements, while the focus of aforementioned approaches is not on speeding up measurements but on answering faster different types of user queries.

[5]http://www.w3.org/TR/rdf-schema/

[6]http://www.w3.org/TR/owl2-overview/

3. INDEXES FOR MEASURING THE CONNECTIVITY OF RDF DATASETS

3.1 Problem Statement

Let $\mathcal{D} = \{D_1, \ldots, D_n\}$ be a set of RDF datasets (or sources). For each D_i we shall use $triples(D_i)$ to denote its triples ($triples(D_i) \subseteq \mathcal{T}$), and U_i to denote the URIs that occur as subjects or objects in these triples. As running example we will use four datasets each containing six URIs as shown in Figure 3 (upper-left corner).

Common URIs in Datasets.
Let $\mathcal{P}(\mathcal{D})$ denote the powerset of $\mathcal{D}$, comprising elements each being a subset of $\mathcal{D}$, i.e. a set of datasets. Our objective is to find the *common URIs* in every element of $\mathcal{P}(\mathcal{D})$, meaning that for each set of datasets $B \in \mathcal{P}(\mathcal{D})$ we want to compute $cu(B)$ defined as:

$$cu(B) = \cap_{D_i \in B} U_i \quad (1)$$

Datasets containing a particular URI.
Another objective is to find all datasets that contain information about a particular URI u, i.e. to compute

$$dsets(u) = \{D_i \in \mathcal{D} \mid u \in U_i\} \quad (2)$$

Considering Equivalence Relationships.
So far we have ignored the `owl:sameAs` relationships. Below we shall see how to semantically complete the previous definitions. Let $sm(D_i)$ be the `owl:sameAs` relationships of a dataset D_i, i.e.:

$$sm(D_i) = \{(u, u') \mid (u, \texttt{owl:sameAs}, u') \in triples(D_i)\} \quad (3)$$

If B is a set of datasets, i.e. $B \in \mathcal{P}(\mathcal{D})$, we will denote with $SM(B)$ the union of the `owl:sameAs` relationships in the datasets of B, i.e. $SM(B) = \cup_{D_i \in B}\ sm(D_i)$. Notice that our running example includes five `owl:sameAs` relationships, shown in Figure 3 (upper-right). If R denotes a binary relation, we shall use $\mathcal{C}(R)$ to denote the transitive and symmetric closure of R. Consequently, $\mathcal{C}(sm(D_i))$ stands for the transitive and symmetric closure of $sm(D_i)$, while $\mathcal{C}(SM(B))$ is the transitive and symmetric closure of the `owl:sameAs` relationships in all datasets in B. The number of real world objects (in abbreviation *rwo*) in a dataset D_i

is the number of *classes of equivalence* of $\mathcal{C}(sm(D_i))$, plus those URIs of U_i that do not occur in $\mathcal{C}(sm(D_i))$ (in order not to count some URIs more than once), e.g. if $U_{temp} = \{u_1, u_2, u_3, u_4, u_5\}$ and there are two `owl:sameAs` relationships, $u_1 \sim u_3$ and $u_1 \sim u_4$, then their closure derives the following classes of equivalence $U_{temp}/\sim = \big\{\{u_1, u_3, u_4\}, \{u_2\}, \{u_5\}\big\}$. The number of real world objects in a set of datasets B is defined analogously.

We can now define the equivalent URIs (considering all datasets in B) of a URI u (and of a set of URIs U) as:

$$Equiv(u, B) = \{ u' \mid (u, u') \in \mathcal{C}(SM(B))\} \quad (4)$$

$$Equiv(U, B) = \cup_{u \in U} Equiv(u, B) \quad (5)$$

We are now ready to "semantically complete" the definitions (1) and (2):

Datasets containing a particular (or equivalent) URI. The set of datasets that contain information about u (or a URI equivalent to u) is defined as:

$$dsets_{\sim}(u) = \{D_i \in \mathcal{D} \mid (\{u\} \cup Equiv(u, \mathcal{D})) \cap U_i \neq \emptyset\} \quad (6)$$

Obviously, it holds $dsets(u) \subseteq dsets_{\sim}(u)$.

Common (or equivalent) URIs in Datasets. The common or equivalent URIs in the datasets in B are defined as:

$$cu_{\sim}(B) = \{ u \in U \mid dsets_{\sim}(u) \supseteq B\} \quad (7)$$

Obviously it holds $cu(B) \subseteq cu_{\sim}(B)$. Now the *rwo* that are in common in the datasets in B are the classes of equivalence of $cu_{\sim}(B)$, i.e. the set $cu_{\sim}(B)/\sim$. Therefore we define the number of common real world objects in the datasets of B as:

$$co_{\sim}(B) = |cu_{\sim}(B)/\sim| \quad (8)$$

In a nutshell, in this paper we focus on how to compute efficiently formulas 6 and 8, for any $u \in U$ and $B \subseteq \mathcal{D}$ respectively.

3.2 The Proposed Indexes

Figure 3 illustrates the proposed indexes over our running example. Let $U = U_1 \cup \ldots \cup U_n$ ($n = 4$ in our example). We propose three indexes:

- `PrefixIndex`: It is a function $pi : Pre(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$ where $Pre(\mathcal{D})$ is the set of prefixes of the datasets in $\mathcal{D}$, i.e. $Pre(\mathcal{D}) = \{ pre(u) \mid u \in U_i, D_i \in \mathcal{D}\}$, e.g. see step 1 of Fig. 3.
- `SameAsCat`: For each u that participates to $SM(\mathcal{D})$ this catalog stores a unique id. All URIs in the class of equivalence of u are getting the same id. Let SID denotes this set of identifiers. The `SameAsCat` is essentially a binary relation $\subseteq U \times SID$, e.g. see step 2 of Fig. 3.
- `ElementIndex`: For each element of $U \cup SID$ this index stores the datasets where it appears, i.e. it is a function $ei : U \cup SID \rightarrow \mathcal{P}(\mathcal{D})$ where $ei(u) = dsets_{\sim}(u)$, e.g. see step 3 of Fig. 3.

The rationale and the construction method for each one is given below.

3.3 Prefix Index

Rationale: Most data providers publish their data using prefixes that indicate their company or university (e.g., *DBpedia* URIs starts with prefix *http://dbpedia.org*).

A `PrefixIndex` can greatly reduce the cost of finding common URIs. First, there is no need to compare URIs containing different namespaces. Second, if a prefix p exists in one dataset only, it is not possible for the URIs starting with p to be found in another dataset.

Construction Method: For getting the prefixes of a D_i stored in a triplestore, one can either submit a SPARQL query or scan each U_i once. We also count the frequency of each prefix in the dataset. If nm is a namespace, $pi(nm)$ is the set of ids of the datasets that contain it. This set is stored in ascending order with respect to the frequency, e.g. in our running example we can see that for the prefix *dbp* the dataset *DBpedia* contains the most URIs, consequently, the ID of this source (i.e. 2) is in the last position of $pi(dbp)$. This ordering is beneficial for reducing the ASK queries as we shall see later in §3.5.

Moreover, `PrefixIndex` enables a fast method for finding the upper bound of $|dsets(u)|$ for a particular u (since $dsets(u) \subseteq pi(pre(u))$), e.g. in our running example a URI starting with prefix *en_wiki* can be possibly found in four datasets, because all four datasets contain this prefix. However, a URI with prefix *yg* can appear only in one dataset since this prefix appears only in *Yago*.

3.4 SameAs Catalog

Rationale: It is required for formulas 6, 7 and 8 as explained in §3.1.

Construction Method: We introduce a signature-based algorithm where each class of equivalence will get a signature (id) and the signature is constructed incrementally during the computation. After the completion of the algorithm, all URIs that belong to the same class of equivalence will have the same identifier. The algorithm assigns to each pair $(u, u') \in SM(B)$ an identifier according to the following rules:

1. If both URIs have not an identifier, a new identifier is assigned to both of them. E.g. Table 1 contains two classes of equivalence and four URIs. In the next step, a new `owl:sameAs` pair containing two URIs without identifier is inserted resulting to a new class of equivalence which is shown in Table 2.
2. If u has an identifier while u' has not, u' gets the same identifier as u. Table 3 shows such an example where a new URI (u_7) is assigned the identifier of an existing one (u_3).
3. If u' has an identifier while u has not, u gets the same identifier as u'.
4. If both URIs have the same identifier, the algorithm continues.
5. If the URIs have a different identifier, these identifiers are concatenated to the lowest identifier. In case of Table 4, both URIs exist in the `SameAsCat` and have a different identifier. For this reason, the classes of equivalence of these identifiers are merged.

We can say that the algorithm constructs incrementally chains of `owl:sameAs` URIs where each URI becomes a member of a chain if and only if there is a `owl:sameAs` relationship with a URI which is already member of this chain. Its correctness is based on the following proposition.

Prop. 1. If $(a, b) \in SM(B)$ and $(a, c) \in SM(B)$ then $(b, c) \in C(SM(B))$.

Proof. Firstly, $(b, a) \in C(SM(B))$ because of symmetry. By taking the transitive closure of $(b, a), (a, c)$, we get that $(b, c) \in C(SM(B))$. □

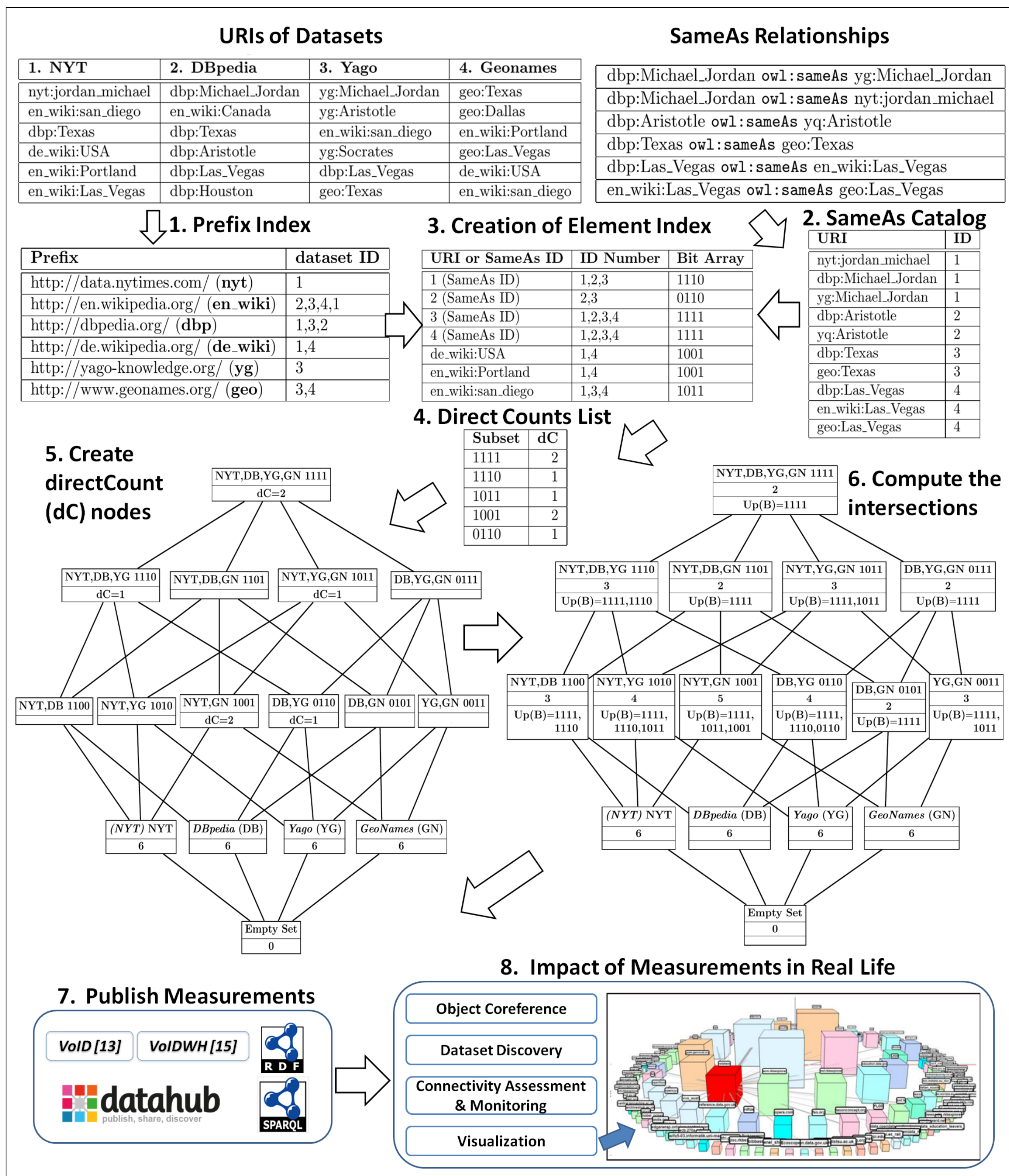

Figure 3: Running Example

Its benefit is that it needs one pass for each `owl:sameAs` pair in order to compute the transitive and symmetric closure, and its time complexity is $\mathcal{O}(n)$, where n is the number of `owl:sameAs` pairs. However, it is needed to keep in memory chains of `owl:sameAs` in order to connect such chains with new `owl:sameAs` relationships. Indeed, the space complexity is $\mathcal{O}(m)$, where m is the number of unique URIs that occur in `owl:sameAs` pairs, since in the worst case each URI

Table 1: Classes of Equivalence

ID	URIs
1	u_1, u_2
2	u_3, u_4

Table 2: Insert u_5 `owl:sameAs` u_6

ID	URIs
1	u_1, u_2
2	u_3, u_4
3	$\mathbf{u_5, u_6}$

Table 3: Insert u_3 `owl:sameAs` u_7

ID	URIs
1	u_1, u_2
2	$u_3, u_4, \mathbf{u_7}$
3	u_5, u_6

Table 4: Insert u_1 `owl:sameAs` u_3

ID	URIs
1	$u_1, u_2, \mathbf{u_3, u_4, u_7}$
~~2~~	~~u_3, u_4, u_7~~
3	u_5, u_6

is saved both in `SameAsCat` and in classes of equivalence until the end of the algorithm. Regarding the size of the catalog we store for each URI a distinct arbitrary number. Since each real world object is represented by exactly one identifier, the number of unique identifiers in the `SameAsCat` is equal to the number of unique real world objects of $SM(B)$. In our running example, the `owl:sameAs` catalog is shown in Figure 3 (step 2). In our implementation we use two HashMaps: the first for each URI (key) keeps its signature (value), while the second for each signature (key) keeps the set of URIs that have this signature (value). In case of executing rule 5, the URIs of the two signatures of the second HashMap are merged in the lowest signature, while the entry of the highest signature is removed. The signature of URIs which previously had the highest signature (of the two) should also be updated in the first HashMap.

Alternatively, one can use Tarjan's connected components (*CC*) algorithm [22] that uses Depth-First Search (*DFS*). The input of *CC* algorithm is a graph which should have been created before running the algorithm. For this reason, we have to read all the `owl:sameAs` pairs once $\mathcal{O}(n)$ (i.e., each `owl:sameAs` represents an edge) in order to construct the graph. The time complexity of *CC* algorithm is $\mathcal{O}(m+n)$. Regarding the space, the creation of graph requires space $n+2m$ because the graph is undirected and we should create bidirectional edges while the *CC* algorithm needs space $\mathcal{O}(m)$, since in the worst case it needs to keep all the nodes in *DFS* stack (i.e., unique URIs). However, the graph should be loaded in memory in order to run the *CC* algorithm, thereby the total space needed is $\mathcal{O}(n+m)$. In §5.1 we compare the execution time of the two aforementioned approaches.

3.5 Element Index

Rationale: `ElementIndex` is essentially a function $ei : U \cup SID \rightarrow \mathcal{P}(\mathcal{D})$ where $ei(u) = dsets_{\sim}(u)$ which is needed for finding fast the datasets to which a URI appears. To reduce its space we can avoid storing URIs that occur in only one dataset and this information can be obtained by the `PrefixIndex`. We can identify two basic candidate data structures for this index: (1) a bit array of length $|\mathcal{D}|$ that indicates the datasets to which each element belongs (each position in the bit string corresponds to a specific dataset), or (2) an IR-like inverted index [26], in which for each URI store the ID of the datasets (a distinct arbitrary number). A bitmap index keeps for each element of $U \cup SID$ a bit array of length $|\mathcal{D}|$, therefore its total size is $|U \cup SID| * |\mathcal{D}|$ bits. An inverted index for each element of $U \cup SID$ keeps a posting list of dataset identifiers. If ap is the average size of the posting lists, then the total size is $|U \cup SID| * ap * \log|\mathcal{D}|$ bits (where $\log|D|$ corresponds to the required bits for encoding $|\mathcal{D}|$ distinct identifiers). If we solve the inequality $Bitmap \leq InvertedIndex$, we get that the size of bitmap is smaller than the size of inverted index when $ap > \frac{|\mathcal{D}|}{\log|\mathcal{D}|}$.

Construction Method: Algorithm 1 creates the element index while considering the aforementioned indexes (this algorithm can be used for both types of data structures). With $Left(r)$ we denote the set of elements that occur in the left side of a binary relation or function r, and with $[u]$ the class of equivalence of u. Figure 3 (middle-right) shows the resulted index of our running example. The combination of the first and the second column of the element index represents the inverted index of the running example, while the combination of the first and the third represents the element index with a bit array of length n (instead of datasets IDs).

Returning to Algorithm 1, at first, if a URI u (of a dataset D_i) belongs to the `SameAsCat` (assuming that each class of equivalence involves URIs that occur in different datasets) an entry is added to the element index comprising the identifier of u (taken by the `SameAsCat`) and the dataset ID of D_i (lines 3-5). For instance, URI *yg:Aristotle* exists in `SameAsCat` (see Figure 3). Thereby, its identifier and dataset ID is added to the element index. When the identifier already exists in the element index, the corresponding entry is updated by adding only the dataset ID (line 7). When u does not exist in the `SameAsCat`, the next step is to check if u already belongs to the element index (because it was encountered previously). Then the index entry of u is updated by adding the dataset ID (lines 8-9).

The last step (if the two previous failed) corresponds to URIs that neither belongs to $Left($`SameAsCat`$)$ nor to $Left(ei)$. In this step, we exploit `PrefixIndex` by using the function *pi* for taking the datasets containing the namespace nm of u (line 12). One approach is to add $ei([u_j]) \leftarrow \{i\}$ if the nm of u_j exists in two or more datasets. In this way, at the end the *ei* could contain *URIs* that occur in one D_i. For this reason, such URIs should be deleted at the end (an extra step is required). Alternatively, one can perform an extra check for ensuring that a URI exists in at least two datasets and this is described in lines 12-17.

Let $ask(u, D_k) =$ "$ASK\ D_k\ \{\ u\ ?p\ ?o\ \}\ \cup\ \{\ ?s\ ?p\ u\ \}$" be an ASK query for a URI u and $answer(ask(u, D_k))$ a function which returns either true or false. At first, we find which datasets contain URIs starting with the namespace nm. In particular, we read the datasets IDs that $pi(nm)$ returned in reverse order and we send ASK queries only to a dataset D_k that contains more URIs starting with nm (starting from the D_k with the most URIs for nm). In case of $answer(ask(u, D_k)) = true$, a new entry is added to *ei* which is composed of u and the IDs i and k. For instance, the prefix *en_wiki* can be found in all datasets. However, for the URI *en_wiki:san_diego* of dataset NYT (with ID 1), we do not send a query since NYT dataset contains the most URIs for *en_wiki* prefix. For the same URI *en_wiki:san_diego* of dataset *Yago* (with ID 3) the first step is to send an ASK query to NYT source. In this case $answer(ask(en_wiki{:}san_diego, NYT)) = true$, therefore we

create a new entry to ei for this URI and we add both the IDs of NYT and $Yago$. On the contrary, for URIs starting with prefixes that exist only in one dataset (e.g., *yg:Socrates*), the algorithm continues without sending any ASK query (see §3.3). The approach with the ASK queries is the only one that can be used if the data cannot fit in memory. On the contrary, when the required memory space is available it is faster to keep the URIs until the end (and then remove those that do not occur in ≥ 2 datasets).

Algorithm 1 Element Index Creation

Input: A set of URIs U_i for each dataset, the `SameAsCat` and the `PrefixIndex`

Output: An element index ei of real world objects that exist in ≥ 2 datasets

```
 1: for all D_i ∈ 𝒟 do
 2:    for all u_j ∈ U_i do
 3:       if u_j ∈ Left(SameAsCat) then
 4:          if [u_j] ∉ Left(ei) then          ▷ signature of u_j
 5:             e_i([u_j]) ← {i}
 6:          else
 7:             e_i([u_j]) ← e_i([u_j]) ∪ {i}
 8:       else if u_j ∈ Left(ei) then
 9:          e_i(u_j) ← e_i(u_j) ∪ {i}
10:       else if u_j ∉ Left(SameAsCat) ∪ Left(ei) then
11:          nm ← namespace(u_j)
12:          for all D_k ∈ pi(nm) in reverse order do
13:             if D_k = D_i then
14:                break
15:             if answer(ask(u_j, D_k)) = true then
16:                e_i(u_j) ← {i, k}
17:                break
18: return ei
```

Algorithm 1 reads each $u \in U_i$ once for all $D_i \in \mathcal{D}$, thereby the complexity is $\mathcal{O}(y)$ where y is the sum of $|U_i|$. Alternatively, one can use a straightforward method (*sf*) that finds the intersections of all subsets in $\mathcal{P}(\mathcal{D})$. In a straightforward method, the URIs are sorted lexicographically in order to perform binary searches for finding the common URIs of any subset. In particular, for each subset $B \in \mathcal{P}(\mathcal{D})$, for all the URIs (e.g., U_1) of the smallest dataset (regarding the size of URIs) one or more binary searches are performed, starting from the second smallest dataset (e.g., U_2) and so forth. Regarding the complexity, in case of having n datasets inside the subset B, in the worst case we should perform for each URI in U_1 a binary search for each of the $n-1$ remaining datasets. For all subsets of $\mathcal{D}$ the complexity becomes exponential, since there exists $2^{|\mathcal{D}|}$ possible subsets, thereby the cost is $\mathcal{O}(2^{|\mathcal{D}|} n \log n)$. In §5.1, we show experiments for comparing the execution time of the index approaches and the straightforward method.

Ordering the Prefix Index for Reducing the ASK Queries. In Algorithm 1 we send ASK queries only to the datasets containing more URIs than D_i for prefix p. Here, we describe a) how different combinations of the dataset IDs sequence in prefix index produce different numbers of ASK queries and b) why the proposed ordering reduces the number of ASK queries. Let $U_p = \{u \in U_i \mid namespace(u) = p\}$ and $U'_i = U_i \cap U_p$. Let pos return the dataset ID of specific position for a prefix p list, i.e. it is a function $pos : \mathbb{Z} \rightarrow \mathbb{Z}_{>0}$ and let n be the number of the different datasets having $|U'_i| > 0$. Suppose that we store the dataset IDs for each prefix randomly. It means that we do not take into account the frequency of the URIs of a prefix in each dataset. In the example of Table 5 we can see the size of each U'_i for a prefix p. Table 6 shows the number of ASK queries for the worst case in which for each pair $\langle D_i, D_j \rangle$ we should check $\forall\ u_i\ s.t.\ u_i \in U'_i$ the result of $answer(ask(u_i, D_j))$. The formula that we use in Table 6 follows: $Asks = |U'_{pos(0)}| * (n-1) + |U'_{pos(1)}| * (n-2) + ... + |U'_{pos(n-1)}| * 0$.

Table 5: Frequency for a prefix p

U'_i	**Freq. of p**
U'_1	1,000,000
U'_2	5,000
U'_3	10,000

Table 6: ASKs per combination for the worst case

Position 0,1,2	**ASKs**
D_1,D_2,D_3	2,005,000
D_1,D_3,D_2	2,010,000
D_2,D_1,D_3	1,010,000
D_2,D_3,D_1	20,000
D_3,D_1,D_2	1,020,000
D_3,D_2,D_1	25,000

Concerning the sequence $\langle D_1, D_2, D_3 \rangle$, in the worst case the number of ASK queries is: $Asks = |U'_1| * 2 + |U'_2| = 2,005,000$ ASK queries. In particular, we send $2 * |U'_1|$ ASK queries in order to check for each URI u_1, where $u_1 \in U'_1$, if $answer(ask(u_1, D_2)) = true$ or $answer(ask(u_1, D_3)) = true$. Then, for each u_2 we send $|U'_2|$ Ask queries (e.g., for each $u_2 \in U'_2$ we check if $answer(ask(u_2, D_3)) = true$). Concerning the sequence $\langle D_2, D_3, D_1 \rangle$, in the worst case the number of ASK queries is: $Asks = |U'_2| * 2 + |U'_3| = 20,000$ ASK queries. In the aforementioned sequence the datasets are in ascending order w.r.t. the frequency of URIs starting with p in D_i (the order that we follow in `PrefixIndex`).

Additionally, the fact that we start to send ASK queries for a URI containing a prefix p from the dataset with the most URIs starting with p makes it more possible the answer of first ASK query to be true. In fact, the dataset containing the most URIs for a prefix is usually the dataset of the publisher of this prefix' URIs.

4. THE LATTICE OF MEASUREMENTS

After the creation of the element index, we can compute the commonalities between any subset of datasets in $\mathcal{D}$. For (a) speeding up the computation of the intersections of URIs and (b) visualizing these measurements (for aiding understanding), we propose a method that is based on a lattice (specifically on a meet-semilattice). If the number of datasets is not high, the lattice can be shown entirely, otherwise (i.e. if the number of datasets is high) it can be used as a navigation mechanism, e.g. the user could navigate from the desired dataset (at the bottom layer) upwards, as a means for dataset discovery. Specifically, we propose constructing and showing the measurements in a way that resembles the *Hasse Diagram* of the *poset*, partially ordered set, $(\mathcal{P}(\mathcal{D}), \subseteq)$. The lattice can be represented as a Directed Acyclic Graph $G = (V, E)$ where the edges point towards the direct supersets, i.e. each directed edge starts from a set B and points to a superset B', where $B \subset B'$ and $|B| = |B'| - 1$ (i.e., there exists exactly one element of B' which is not an element of B). The empty set is the unique `source` node of G (i.e., node with zero in-degree) and the set containing all the datasets (i.e. $\mathcal{D}$) is the unique `sink` node of G (i.e.,

node with zero out-degree). A lattice of $\mathcal{D}$ datasets contains $|\mathcal{D}|+1$ levels while the value of each level k ($0 \le k \le |\mathcal{D}|$) indicates the number of datasets that each subset of level k contains (e.g., level 2 corresponds to pairs). For computing the measurements that correspond to each node (of the $2^{|\mathcal{D}|}$ nodes), one could follow a straightforward approach, i.e. scan the entire element index once per each node, but that would require exponential in number scans, hence prohibitively expensive for high number of datasets. To tackle this problem, below we introduce (i) a *top-down* and (ii) a *bottom-up* lattice-based incremental algorithm that both require *only one scan* of the element index for computing the commonalities between any set of datasets. We should stress that it is not required to compute the entire lattice, one can use the proposed approach for computing only the desired part of the lattice and its incremental nature can offer significant speedups (as we shall see in §5.2).

4.1 Making the Measurements of the Lattice Incrementally

For a subset B, let *directCount*(B) denote its frequency in the element index, i.e.
$directCount(B) = |\{ u \in Left(ei) \mid ei(u) = B \}|$.
We can compute these counters by scanning the element index *once* (in our running example, the outcome is shown in Step 4 of Figure 3). Now let $Up(B) = \{B' \in \mathrm{P}(\mathcal{D}) \mid B \subseteq B', directCount(B') > 0\}$. The key point is that for a subset B, the sum of the *directCount* of $Up(B)$ gives the intersection of the real world objects of the datasets in B, i.e.

$$co_{\sim}(B) = \sum_{B' \in Up(B)} directCount(B') \qquad (9)$$

because the URIs belonging to the intersection of a superset certainly belong to the intersection of each of the subsets of this superset, as stated in the following proposition.

PROP. 2. *Let F and F' be two families of sets. If $F \subseteq F'$ then $\underset{S \in F'}{\cap S} \subseteq \underset{S \in F}{\cap S}$. (The proof can be found in [11].)*

Now we will describe the two different ways for computing the *entire lattice* or a *part of* it.

The **top-down** approach starts from the maximum level having at least one subset B where $B \in Left(directCount)$. At first, we add B to $Up(B)$ if $B \in Left(directCount)$ and then we compute $co_{\sim}(B)$. Afterwards, the list $Up(B)$ of the current node is "transferred" to each subset B' of the lower level since $B' \subseteq B$ implies $Up(B) \subseteq Up(B')$. For example, if $1110 \in Up(1110)$ then surely $1110 \in Up(1100)$. After finishing with the nodes of the current level, we continue with the nodes of the previous level, and so on. As an example, in Figure 4 the second (middle) box of each node indicates the order by which it is visited. As we can see, we start from the maximum level (i.e., level 4) and we continue with the triads and finally with the pairs. The dashed edges represent the edges that are created by the algorithm. In our running example (see Figure 3), we can observe in Step 6 the final intersection value of each node and all the Up(B) for each subset B. For instance, we observe that the $Up(1001)$ are the subsets **1111**, **1011** and **1001**. If we sum their *directCount*, we find $co_{\sim}(1001)$, which is 5.

The time complexity of this algorithm is $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of vertices ($|V| = 2^{|\mathcal{D}|}$) and $|E|$ the number of edges, and it holds that $|E| = |\mathcal{D}| * 2^{(|\mathcal{D}|-1)}$.

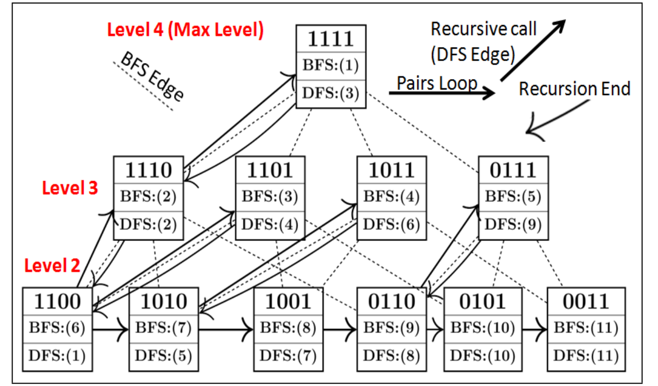

Figure 4: Lattice Traversal (BFS and DFS)

As regards memory requirements, this algorithm will create all edges for each node and it has to keep in memory each subset B (and $Up(B)$) of a specific level k (the number of nodes V_k of level k is given by $V_k = \binom{|\mathcal{D}|}{k}$) because the traversal is BFS (breadth-first search).

In the **bottom-up** approach we first assign the $Up(B)$ to each subset B of level two (the level that contains nodes of each pair of datasets) and we continue upwards. However, and in order to reduce the main memory requirements, instead of covering entirely each level before going to its upper level, we follow a kind of Depth-First Search (*DFS*). We will call it *DFS* although we could also call it "Height First Search" since it starts from the leaves and goes towards the root. The visiting order of the nodes of the example of Figure 4 is shown in the third (bottom) box of each node. Again, the $co_{\sim}(B)$ of a node is computed by adding the *directCount* of $Up(B)$. However, a part of the list (or the whole list) $Up(B)$ of the current node is "transferred" to each superset B' (where $B' \supset B$) of the next level that has not been visited yet. In this way, we visit each node once and this lead to a total cost of one incoming edge per node, i.e. $|V|$ edges ($|V| = 2^{|\mathcal{D}|}$ for the whole lattice) instead of the $|\mathcal{D}| * 2^{(|\mathcal{D}|-1)}$ edges of the top-down approach. The time complexity of this algorithm is $\mathcal{O}(|V|)$, where $|V|$ is the number of vertices. Indeed, it passes once from each node and it creates one edge per node ($|V|+|V|$). Moreover, it needs space $\mathcal{O}(d)$ where d is the maximum depth of the lattice (in our case d is the maximum level having at least one B where $co_{\sim}(B) > 0$). However, we should take into account the cost of checking which of the elements of $Up(B)$ belong to $Up(B')$ since we cannot transfer all the $Up(B)$ to B' (because $B' \supseteq B \not\Rightarrow Up(B) \subseteq Up(B')$), e.g. in our running example $1110 \in Up(0110)$ but $1110 \notin Up(0111)$.
Cost analysis. If B_i in $Left(directCount)$, let $bits_1(B_i)$ be the number of 1's in B_i (e.g. $bits_1(1110)$ is 3), and obviously $2 \le bits_1(B_i) \le |\mathcal{D}|$. Each such B_i belongs to the $Up(B)$ of $2^{bits_1(B_i)} - 1$ subsets (we subtract 1 for the empty set). Since for each such B_i, we have to perform $2^{bits(B_i)}$ checks (i.e., one check when traversing the list of *directCount* nodes and $|2^{bits_1(B_i)} - 1|$ checks when traversing the lattice), it follows that the total cost of such checks is $checkCost = |\sum_{i=1}^{C} 2^{bits_1(B_i)}|$, where C is the number of nodes occurring in *Left(directCount)*, i.e. $C = |\{ei(u) \mid u \in U \cup SID\}|$. It is essentially the cardinality of the codomain of ei and obviously, $C \le |U \cup SID|$ (as we shall see in the experiments $C \simeq 0.1\%$ of $|U \cup SID|$).

Comparison. Both algorithms pass from all nodes having $co_\sim(B) > 0$. The top-down approach requires creating $|\mathcal{D}|/2$ times more edges, however the bottom-up approach has the additional *checkCost*.

PROP. 3. *If the frequency of URIs to datasets follows a power-law distribution (specifically if we group and rank in descending order the directCount nodes according to their number of bits, and assume that the number of nodes of the n-th category ($2 \leq n \leq |\mathcal{D}|$) is given by $f(n) = k*(m/2)^{n-2}$, where k is the number of such nodes having $bits_1(B) = 2$, $m/2$ is the reduction factor ($1 < m \leq 2$), and assume $k = |\mathcal{D}^2|/4$), then the bottom-up approach is more efficient than the top-down if $|\mathcal{D}|^2 * \frac{m^{|\mathcal{D}|-1}-1}{m-1} < (|\mathcal{D}|-2) * 2^{|\mathcal{D}-1|}$.*

Proof sketch: The bottom-up approach is better than the top-down when *checkCost* < extra edges of the top-down. If we subtract the edges of the bottom-up approach (i.e., $2^{|\mathcal{D}|}$) from the edges of the top-down approach (i.e., $|\mathcal{D}| * 2^{|\mathcal{D}|-1}$) we get $(|\mathcal{D}|-2) * 2^{(|\mathcal{D}|-1)}$. Now let calculate *checkCost* for the assumed power-law distribution. The *n-th* category (each node B_n of the n-th category has $bits_1(B_n) = n$) of nodes contains $k * (m/2)^{|bits_1(B_n)|-2}$ nodes and we need $k*(m/2)^{|bits_1(B_n)|-2} * 2^{|bits_1(B_n)|}$ checks. The corresponding sum (i.e., of checks) leads to $checkCost = 4k * \frac{m^{|\mathcal{D}|-1}-1}{m-1}$. By assuming $k = |\mathcal{D}^2|/4$ (which is quite reasonable based on our experiments) we get the inequality of Prop. 3. □

As it is shown in Figure 5, it follows that (a) for $m \approx 1$ (i.e., nodes are reduced by half as categories grow), the bottom-up approach is better that the top-down when $|\mathcal{D}| > 6$, (b) for $m = 2$ (i.e., each category has the same number of nodes) top-down is always better, whereas (c) for $m = 1.6$ the bottom-up traversal is more efficient than the top-down for $|\mathcal{D}| \geq 17$. The experiments in §5.2 are explained by this analysis, and we identify the same trade-off for $m = 1.6$. As regards memory requirements, the top-down approach needs significantly more space since it keeps in the worst case in memory the nodes of a specific level k (i.e., $\binom{|\mathcal{D}|}{k}$ nodes) whose number can be huge for big lattices while the bottom-up keeps at most $|\mathcal{D}|$ nodes (i.e., maximum depth is $|\mathcal{D}|$). Finally, an alternative straightforward, but less efficient, way to compute the lattice is to use a *directCount* scan (*dcs*) approach for each subset. The complexity of *dcs* is $\mathcal{O}(|V|*C)$ (exponential in number *directCount* scans).

Computing a Part of the Lattice. Here we describe how one can compute only parts of the lattice.

- *Single Node.* For computing a specific node B, we can scan all $B_i \in Left(directCount)$ and sum all the $directCount(B_i)$ if $B_i \in Up(B)$. Its complexity is $\mathcal{O}(C)$.
- *Threshold-based Nodes.* For computing only the $co_\sim(B)$ of subsets that satisfy a specific threshold (e.g., $co_\sim(B) \geq 20$), it is beneficial to use the bottom-up approach. Indeed, fewer nodes will be created since we can exploit Prop. 2 to avoid creating nodes that are impossible to satisfy the threshold.
- *Lattice of a subset of Datasets.* For computing only the nodes of the lattice that contain datasets only from a particular subset D' ($D' \subset \mathcal{D}$), the previous analysis as regards the two types of traversal holds also in this case. The only difference is that here instead of using the original *directCount* we use $directCount(B, D')$ defined as $directCount(B, D') = |\{ u \in Left(ei) \mid B = ei(u) \cap D'\}|$ which can be produced by scanning once the element index.

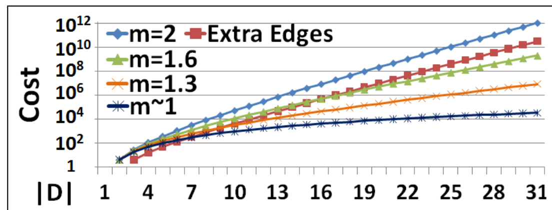

Figure 5: CheckCost vs extra edges for various m

Table 7: Datasets Statistics

Domain	$|\mathcal{D}|$	**|Triples|**	**|URIs|**
Cross Domain **(CD)**	19	293,129,862	103,281,343
Geographical **(GEO)**	14	155,591,494	34,169,442
Life Sciences **(LF)**	17	66,684,349	9,725,521
Government **(GOV)**	45	61,189,128	6,896,850
Publications **(PUB)**	76	53,930,138	10,932,689
Media **(MED)**	9	15,267,271	4,434,038
Linguistics **(LIN)**	8	9,128,072	2,059,465
Social Networking **(SN)**	96	2,451,093	561,686
User Content **(UC)**	16	1,059,255	308,193
All	300	658,430,662	172,369,227

- *All Nodes containing a particular dataset.* For computing only the nodes of the lattice that contain a particular subset D_i ($D_i \in \mathcal{D}$), we follow a similar approach, i.e. instead of using the original *directCount* we use $directCount(B, D_i)$ defined as $directCount(B, D_i) = |\{ u \in Left(ei) \mid ei(u) = B \text{ and } D_i \in B \}|$.

5. EXPERIMENTAL EVALUATION

Here we report the results of two kinds of experiments: a) interesting measurements over the entire LOD, and b) measurements that quantify the speedup obtained by the introduced techniques. We used a single machine having an i7 core, 8 GB main memory and 1TB disk space, and the triple-store *Openlink Virtuoso*[7] *Version 06.01.3127* for uploading the datasets and for sending SPARQL ASK queries.

Datasets. The set of datasets that was used in the experiments contains 300 datasets which were collected from the following resources: (a) the dump of the data which were used in [21], (b) online datasets from `datahub.io` website, and (c) subsets of (i) *DBpedia* version 3.9, (ii) *Wikidata*, iii) *Yago* and iv) *Freebase*. Table 7 shows the number of datasets, triples and URIs for each domain (in descending order w.r.t. their size in triples). Most datasets are from the social networking domain, however, most of them contain a small number of triples and URIs. On the contrary, 68% of triples and 79.7% of URIs are part of cross domain and geographical datasets although their union contains 33 (of 300) datasets. The selected set of datasets is quite representative and adequately large for the needs of this paper (658 million triples, powerset of $|\mathcal{D}| = 300$ containing 2^{300} elements).

5.1 Measurements over the Datasets

Statistics derived by the Indexes. Table 8 synopsizes some interesting statistics for the datasets and the creation of the element index. Firstly, according to the prefix index, each dataset's URIs contains on average 212 different prefixes. Element index contains 6.2 million real world objects (*rwo*). As one can see, there are 3,293,248 *rwo* (2.3% of all the *rwo*) which are part of three or more datasets. This

[7]http://virtuoso.openlinksw.com/

Table 8: Index Creation Statistics

Category	Value
Prefix Index Size	63,803
Unique Real World Objects	141,269,960
Element Index Size (*rwo*)	6,242,344
Element Index Size (URIs)	17,840,499
Asks Number	6,684,242
rwo in 3 or more D_i	3,293,248
URIs corresponding to *rwo* in 3 or more D_i	12,296,650
Num. of Lattice Nodes (threshold $\geq$ 30)	130,525,631
Num. of Lattice Nodes (threshold $\geq$ 20)	1,541,968,012

Table 9: SameAs Catalog Statistics

Category	Value
SameAs Triples	13,158,621
SameAs Catalog Size	18,789,593
SameAs Triples Inferred	19,450,107
Pairs sharing at least 1 real world object	6,708
New Pairs discovered due to SameAs Alg.	2,393
Triads sharing at least 1 real world object	74,432
New Triads discovered due to SameAs Alg.	48,658
SameAs Unique IDs	6,218,958

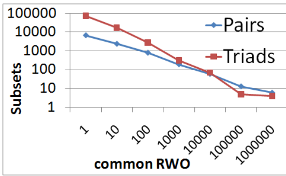

Figure 6: # of Pairs, Triads per Threshold

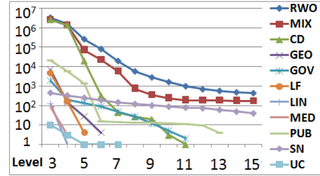

Figure 7: Unique(RWO) - Max Subset per Level

percentage corresponds to 12,296,650 URIs (8% of unique URIs). The number of unique B having $directCount(B) > 0$ are 5,399, i.e. 0.1% of ei size. We used the aforementioned $directCounts$ for computing the $co_{\sim}(B)$ of 130 million lattice nodes (all pairs, triads, quads, quintets and each subset B where $|B| \geq 6$ and $co_{\sim}(B) \geq 30$) in 3.5 minutes and the $co_{\sim}(B)$ of 1.5 billion nodes (having $co_{\sim}(B) \geq 20$) in 35 minutes by using the bottom-up traversal that described in §4.1. Finally, we excluded from this computation URIs belonging to rdf, $rdfs$, owl and popular ontologies such as $foaf$.

Gain from transitive and symmetric closure computation. Regarding the `owl:sameAs` catalog, Table 9 shows some statistics derived by the computation of transitive and symmetric closure. The computation of closure had as a result 2,393 new pairs (35.6% of the number of all connected pairs) and 48,658 new triads (65.3%) that have at least one common real world object (without taking into account URIs belonging in popular ontologies). Moreover, the algorithm found more than 19 million new `owl:sameAs` relationships. Indeed, the increase percentage of `owl:sameAs` triples was 147.8%. The unique URIs of the `owl:sameAs` catalog are 18,789,593 while the different *rwo* are 6,218,958. It means that on average there are 3 URIs for a specific real world object. Finally, Figure 6 shows how many pairs exist for a number of *rwo* threshold (e.g., threshold 10 means that two datasets shares at least 10 *rwo*) and the analogous measurement for the triads.

Common real world objects among three or more datasets. Table 10 shows the ten subsets of size three or more having the most common real world objects (e.g., in descending order according to the number of common *rwo*). The most connected triad contains three cross domain datasets. Particularly, the subset comprising of the datasets *DBpedia*, *Freebase* and *Yago* shares 2.7 million of *rwo* while the quad that contains also *Wikidata* (apart from these datasets) contains 1.43 million of *rwo*. Afterwards, combinations of cross-domain and geographical datasets follows. The first triad that does not contain one of the aforementioned datasets includes three datasets from the publication domain (*d-nb.info*, *bnf.fr*, *id.loc.gov*) which share approximately 21 thousand *rwo*.

Table 10: Top-10 Subsets ≥ 3 with the most common *rwo*

Datasets of subset B	$co_{\sim}(B)$
1: {DBpedia,Freebase,Yago}	2,709,171
2: {DBpedia,Freebase,Wikidata}	1,950,319
3: {DBpedia,Yago,Wikidata}	1,435,713
4: {Yago,Freebase,Wikidata}	1,434,407
5: {DBpedia,Yago,Freebase,Wikidata}	1,434,404
6: {DBpedia,GADM,Freebase}	107,968
7: {DBpedia,GeoNames,Freebase}	98,985
8: {DBpedia,GADM,Wikidata}	96,968
9: {GADM,Freebase,Wikidata}	96,968
10: {DBpedia,GADM,Freebase,Wikidata}	96,968

Figure 7 shows the unique real world objects and the maximum subset (e.g., subset with the most common *rwo*) per lattice level for each domain. The mix corresponds to subsets that possibly contain datasets from more than one domain. The most connected domain from level 3 to 6 is the cross domain whereas in the remaining levels (from 7 to 15) the domain with the most common *rwo* is the social networking domain. Moreover, regarding combinations with datasets from different domains, there are 8 datasets that shares approximately hundreds of *rwo* and 15 datasets sharing over a hundred of *rwo*. Most of these *rwo* predominantly refer to geographical places and to popular persons. Generally, cross domain datasets take part in the most combinations with datasets from different domains.

Top datasets containing frequent real world objects. Regarding the datasets having the most real world objects in a subset of three or more datasets, *DBpedia* is the biggest dataset as we can observe in Table 11 (in ascending order with respect to the number of *rwo* in three or more datasets). It is rational that *DBpedia* is first in this category since it is the biggest hub of the LOD Cloud while the three other popular cross domain datasets follow. The other datasets are predominantly from the geographical domain while there are datasets from the publications, media and lifescience domain containing more than ten thousand of such *rwo* objects. Additionally, 129 of 300 datasets (43%) contain at least hundreds of *rwo* that can be found in three or more datasets. Moreover, we can see the percentage of the *rwo* of each dataset that exist in three or more datasets. It is worth mentioning that *GeoNames* and *LinkedGeoData* have the lowest percentages regarding the datasets of Table 11 while the cross domain datasets have the bigger ones.

5.2 Efficiency of Measurements

Here we focus on measuring the speedup obtained by the introduced indexes and their construction.

Savings by Prefix Index. Regarding the prefix index, 89% of prefixes exist only in one dataset. However, this percentage corresponds only to the 10.8% of distinct URIs while 11% of the prefixes to the 89.2% of URIs. In any implemen-

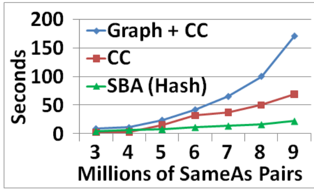

Figure 8: SameAs Catalog Construction time

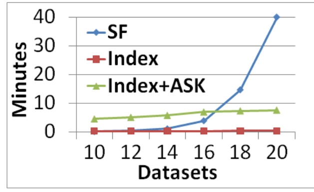

Figure 9: Comparison of different approaches

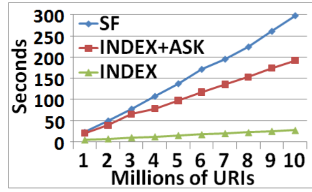

Figure 10: Comparison with stable $|\mathcal{D}| = 17$

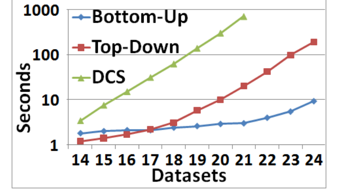

Figure 11: Execution Time of Lattice creation

Table 11: Top-10 datasets with the most *rwo* existing at least in 3 datasets

Dataset D_i	*rwo* in ≥ 3 D_i	(% of D_i *rwo*)
DBpedia	3,246,415	17.3%
Freebase	3,237,604	11.3%
Yago	2,712,930	48.0%
Wikidata	1,952,222	7.3%
GADM Geovocab	108,503	9.4%
GeoNames	102,747	0.4%
d-nb.info	65,076	4.2%
LinkedGeoData (LGD)	43,265	0.6%
Opencyc	34,313	26%
LMDB	30,225	2.3%

tation (e.g., index approach with or without ASK queries), the URIs starting with a prefix existing in one dataset can be ignored, therefore there is no need to compare these URIs with others (e.g., by sending an ASK query or by keeping them until the end). In our case, we ignored 16,689,866 URIs and we sent ASK queries for a URI of a specific dataset only to the other datasets having more URIs for each prefix (see subsection 3.5). For this reason (e.g., the optimized sequence of datasets in prefix index) 6.68 million ASK queries (1 ASK query per 19 URIs having a prefix that can be found in two or more datasets and does not belong to a sameAs relationship) sent where in any random case the number could be much bigger.

SameAs Signature-Based vs Connected Components Algorithm. Here we compare the signature-based algorithm (*SBA*) versus Tarjan's connected components (*CC*) algorithm [22] that uses Depth-First Search (*DFS*) and was described in §3.4. We performed experiments for 3 to 9 million randomly selected `owl:sameAs` relationships and the results are shown in Figure 8. As one can see, the experiments confirmed our expectations since the signature-based algorithm is much faster than the combination of the creation of graph and *CC* algorithm while it is even faster than the *CC* algorithm as the number of `owl:sameAs` pairs increases. Regarding the space, for 10 million or more pairs it was infeasible to create and load the graph due to memory limitations. For this reason we failed to run the *CC* algorithm, however, one can use techniques like those presented in [1] to overcome this limitation. The signature-based algorithm needed only 45 seconds to compute the closure of the `owl:sameAs` pairs of our experiments (in number 13 million).

Index Approach vs Straightforward Method. Here we compare the execution time of the following three methods that described in §3.5: (a) an index approach without ASK queries (*Index*) (b) an index approach with ASK queries (*Index+ASK*) and (c) a straightforward method (*sf*). For the index approaches we include in the execution time the creation of the prefix index and the calculation of lattice nodes by using the bottom-up approach that described in §4. Figure 9 shows the execution time in minutes for the aforementioned approaches for varying number of $|\mathcal{D}|$. The datasets of this experiment belong to the publications domain and the number of URIs is approximately nine millions. As the number of datasets grows, the execution time of the *sf* method increases exponentially. On the contrary, the execution time of the *Index* approach ranges from 17 to 23 seconds. The *Index+ASK* approach needs more time comparing to the other two approaches for 10-16 datasets. However, it is faster than the *sf* method for $|\mathcal{D}| > 16$ since its execution time does not increase so much when new datasets are added. In another experiment that is shown in Figure 10, we report the efficiency for various values of $|URIs|$ but with stable number of $|\mathcal{D}| = 17$. The number of URIs for these 17 datasets varies from 8,000 to 1,000,000. As one can see, the execution time of both *sf* and *Index+ASK* methods increases linearly as the number of URIs grows. In this experiment, the number of ASK queries increased linearly when more URIs added. Indeed, the execution time of *Index+ASK* approach highly depends on the number of ASK queries and their response time. Consequently, in case of adding millions of URIs having a prefix that can be found only in one dataset, the number of ASK queries will not be increased. Therefore, the execution time will be increased less than linearly in that case. Finally, the *Index* approach is again very fast and its execution time ranges from 5 to 27 seconds. In these experiments it is evident that the *Index* approach is always faster than the other approaches. However, in the experiments of §5.1, where the data was infeasible to fit in memory, we used the *Index+ASK* approach.

Computation of power set intersections. Here we compare the performance of the two lattice incremental algorithms and the *directCount* scan (*dcs*) approach for each subset which described in §4.1. We selected 24 datasets that are highly connected (i.e., each subset of lattice has $co_{\sim}(B) > 0$) and the number of *directCount* nodes for the lattice of these 24 datasets is approximately 1,000 (from over 4 million *rwo*). In Figure 11, we can see that each incremental approach is faster than the *dcs* approach. Concerning the incremental approaches, one can clearly see the trade-off between the two approaches. Indeed, for lower number of datasets the top-down approach is faster, since the cost of edges creation is lower than the cost for checking the $Up(B)$ of each subset B. As the number of datasets grows (for ≥ 17 datasets), the bottom-up approach is faster, since the number of edges (and their cost) increases greatly compared to the cost of checking the $Up(B)$. Moreover, for $|\mathcal{D}| \geq 25$, it was infeasible to use the top-down approach due to memory limitations while with the bottom-up approach we computed the $co_{\sim}(B)$ of more than 2^{30} subsets as mentioned in §5.1.

6. CONCLUDING REMARKS

Existing approaches for measurements over the LOD either report measurements between pairs of datasets, or focus on document-based features (not dataset-based), or do not focus on indexes for speeding up such measurements. Since it would be prohibitively expensive to compute measurements that involve more than two datasets without special indexes, in this paper we have introduced various indexes (and their construction algorithms) that can speedup such measurements. We introduced a namespace-based *prefix index*, a *sameAs catalog* for computing the symmetric and transitive closure of the `owl:sameAs` relationships encountered in the datasets, a semantics aware *element index* (that exploits the aforementioned indexes) and two lattice-based incremental algorithms for speeding up the computation of the intersection of URIs of any set of datasets. We showed that with the proposed algorithm it takes only 45 seconds to compute the reflexive and transitive closure for 13 millions of `owl:sameAs` relationships. As regards the computation of intersections we showed that the combination of the element index and the bottom-up incremental lattice-based algorithm is much faster (even 100 times faster for 20 datasets) than a straightforward method whose time complexity increases exponentially as the number of datasets and their size increase. We exploited the introduced indexes for making various measurements over the entire LOD. A few indicative follow. The measurements showed that a dataset contains on average URIs with 212 different prefixes, and that 89% of the prefixes are used in 10.8% of URIs and occur in only one dataset. Concerning `owl:sameAs` relationships, for a real world object exist on average 3 URIs. The transitive and symmetric closure of the `owl:sameAs` relationships of all datasets yielded more than 19 million new `owl:sameAs` relationships, and this increases the connectivity of the datasets: 35.6% of the 6,636 connected pairs of datasets are due to these new relationships. Regarding the *rwo* of element index, 60% of them exist in ≥ 3 datasets.

The measurements also reveal the "sparsity" of the current LOD cloud and make evident the need for better connectivity. Only 2.3% of real world objects (in number 3,293,248) are part of three or more datasets. Most of these real world objects (belonging in ≥ 3 datasets) are part of cross domain datasets such as *DBpedia* and geographical datasets like *GeoNames*. Additionally, many datasets from the social networking domain are highly connected.

There are several topics that are worth further research. An interesting direction is to investigate how to make the lattice of measurements adaptive (in a way similar in spirit with [27] for data series). Another direction is to investigate the speedup that can be achieved by parallelizing the measurements in a MapReduce context, or by developing methods for approximate measurements.

Acknowledgments. This work was partially funded by the BlueBRIDGE project (H2020 Research Infrastructures, 2015−2018, Project No: 675680).

7. REFERENCES

[1] C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *Proceedings of the VLDB Endowment*, 2(1):862–873, 2009.

[2] S. Auer, J. Demter, M. Martin, and J. Lehmann. LODStats - an Extensible Framework for High-Performance Dataset Analytics. In *Proceedings of EKAW*, volume 7603, pages 353–362. 2012.

[3] N. Bikakis and T. K. Sellis. Exploration and visualization in the web of big linked data: A survey of the state of the art. In *LWDM*, 2016.

[4] C. Bizer, P. Boncz, M. L. Brodie, and O. Erling. The meaningful use of big data: four perspectives–four challenges. *ACM SIGMOD Record*, 40(4):56–60, 2012.

[5] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle: a search and metadata engine for the semantic web. In *Proceedings of CIKM*, pages 652–659, 2004.

[6] M. B. Ellefi, Z. Bellahsene, S. Dietze, and K. Todorov. Dataset recommendation for data linking: an intensional approach. In *Proceedingss of ESWC*, volume 9678, pages 36–51, 2016.

[7] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *ISWC*, volume 4825, pages 211–224, 2007.

[8] J. Hendler. Data integration for heterogenous datasets. *Big data*, 2(4):205–215, 2014.

[9] A. Hogan, A. Harth, and S. Decker. Performing object consolidation on the semantic web data graph. In *Proceedings of the WWW Workshop I^3*, 2007.

[10] A. Hogan, A. Harth, J. Umbrich, S. Kinsella, A. Polleres, and S. Decker. Searching and browsing linked data with swse: The semantic web search engine. *Web semantics: science, services and agents on the world wide web*, 9(4):365–401, 2011.

[11] T. Jech. *Set theory*. Springer Science & Business Media, 2013.

[12] M. H. Keith Alexander, Richard Cyganiak and J. Zhao. Describing linked datasets with the VoID vocabulary, W3C interest group note, 2011.

[13] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[14] M. Mountantonakis, N. Minadakis, Y. Marketakis, P. Fafalios, and Y. Tzitzikas. Quantifying the connectivity of a semantic warehouse and understanding its evolution over time. *IJSWIS*, 12(3), 2016.

[15] M. Nentwig, T. Soru, A.-C. N. Ngomo, and E. Rahm. Linklion: A link repository for the web of data. In *The Semantic Web: ESWC Satellite Events*, volume 8798, pages 439–443. 2014.

[16] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[17] D. Oguz, B. Ergenc, S. Yin, O. Dikenelli, and A. Hameurlain. Federated query processing on linked data: a qualitative survey and open challenges. *The Knowledge Engineering Review*, 30(5):545–563, 2015.

[18] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal*, 25(2):243–268, 2016.

[19] E. Prud' Hommeaux, A. Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.

[20] L. Rietveld, W. Beek, and S. Schlobach. LOD lab: Experiments at LOD scale. In *Proceedings of ISWC*, volume 9367, pages 339–355, 2015.

[21] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *Porceedings of ISWC*, volume 8796, pages 245–260. 2014.

[22] R. Tarjan. Depth-first search and linear graph algorithms. In *Twelfth Annual Symposium on Switching and Automata Theory*, pages 114–121, 1971.

[23] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On graph features of semantic web schemas. *Knowledge and Data Engineering*, 20(5):692–702, 2008.

[24] G. Tummarello, E. Oren, and R. Delbru. Sindice.com: Weaving the open linked data. In *Proceedings of ISWC*, volume 4825, pages 547–560, 2007.

[25] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, and S. Auer. Quality assessment for linked data: A survey. *Semantic Web Journal*, 7(1):63–93, 2016.

[26] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM computing surveys*, 38(2):6, 2006.

[27] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the ACM SIGMOD*, pages 1555–1566, 2014.