Scalable Methods for Measuring the Connectivity and Quality of Large Numbers of Linked Datasets

MICHALIS MOUNTANTONAKIS and YANNIS TZITZIKAS, Institute of Computer Science, FORTH-ICS, Greece & Computer Science Department, University of Crete, Greece

Although the ultimate objective of Linked Data is linking and integration, it is not currently evident how *connected* the current Linked Open Data (LOD) cloud is. In this article, we focus on methods, supported by special indexes and algorithms, for performing measurements related to the connectivity of more than two datasets that are useful in various tasks including (a) *Dataset Discovery* and *Selection*; (b) *Object Coreference*, i.e., for obtaining *complete information* about a set of entities, including provenance information; (c) *Data Quality Assessment and Improvement*, i.e., for assessing the connectivity between any set of datasets and monitoring their evolution over time, as well as for estimating data veracity; (d) *Dataset Visualizations*; and various other tasks. Since it would be prohibitively expensive to perform all these measurements in a naïve way, in this article, we introduce indexes (and their construction algorithms) that can speed up 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. For enhancing scalability, we propose parallel index construction algorithms and parallel lattice-based incremental algorithms, we evaluate the achieved speedup using either a single machine or a cluster of machines, and we provide insights regarding the factors that affect efficiency. Finally, we report measurements about the connectivity of the (billion triples-sized) LOD cloud that have never been carried out so far.

CCS Concepts: • **Information systems** → Resource Description Framework (RDF); Information integration;

Additional Key Words and Phrases: Data quality, dataset discovery, dataset selection, linked data, connectivity, lattice of measurements, big data, mapreduce, spark

ACM Reference format:
Michalis Mountantonakis and Yannis Tzitzikas. 2018. Scalable Methods for Measuring the Connectivity and Quality of Large Numbers of Linked Datasets. *J. Data and Information Quality* 9, 3, Article 15 (January 2018), 49 pages.
https://doi.org/10.1145/3165713

1 INTRODUCTION

The management of the plethora of available linked datasets poses various challenges. There is a need for methods that can deal with large quantities of linked data (volume), to accommodate

This work has received funding from the General Secretariat for Research and Technology (GSRT), the Hellenic Foundation for Research and Innovation (HFRI), and the European Union's Horizon 2020 research and innovation programme under the BlueBRIDGE project (Grant No. 675680).
Authors' addresses: M. Mountantonakis and Y. Tzitzikas, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece; emails: {mountant, tzitzik}@ics.forth.gr.

https://doi.org/10.1145/3165713

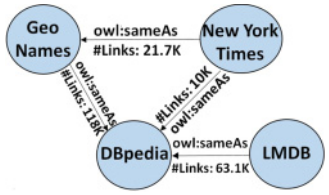

Fig. 1. LargeRDFBench cross-domain Datasets.

the dynamic aspects of data (velocity), to be able to uniformly deal with data originating from different domains and sources (variety), to assess and improve the accuracy of data (veracity), and to provide an indication of the impact of data quality, both for decision making and monetary aspects (value). For tackling these challenges in this article, we introduce methods for assessing the connectivity of large numbers of linked datasets, even if they come from different domains and sources (variety), for assessing their connectivity (integration value) and for providing value added services (like global lookup services). One distinctive characteristic is that we aim at enabling such measurements and services that can manage *all* datasets of the Linked Open Data (LOD) cloud (volume). Such "global scale" semantic indexing is beneficial also for veracity, since the collection of all information about an entity, and the cross-dataset inference that is feasible, allows us to spot the contradictions that exist and at the same time provides information for data cleaning or for estimating and suggesting which data are probably correct or more accurate.

Although the ultimate objective of LOD is linking and integration, for enabling discovery and integrated query answering and analysis, even some basic tasks are nowadays challenging due to the scale and heterogeneity of the datasets: the LODStats website provides statistics about approximately 10,000 discovered linked datasets until August 2014,[1] and it keeps growing. In this article, we focus on methods, supported by special indexes and algorithms, for performing measurements related to the connectivity of more than two datasets that are useful in various tasks including the following: (a) *Dataset Discovery* and *Selection* [13, 37, 40]; (b) *Object Coreference*, i.e., for obtaining *complete information* about one particular entity (identified by a URI) or set of entities, including provenance information; (c) *Data Quality Assessment and Improvement*, i.e., for *assessing the connectivity* between any set of datasets and *monitoring their evolution* over time [32], as well as for *estimating data veracity*; and (d) *Dataset Visualizations* [4], i.e., for providing more informative overviews that could also aid dataset discovery.

For instance, for *dataset discovery*, i.e., task (a), 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. In this article, we show how we can make efficient measurements that involve more than two datasets. The results can be visualized as lattices, like that of Figure 2 that 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 the level of pairs. Concerning task (b), it is not trivial to find all the available URIs for an entity, since this presupposes knowledge of all datasets, whereas the `owl:sameAs` relationships model is an

[1] http://stats.lod2.eu.
[2] http://lod-cloud.net/.
[3] http://github.com/AKSW/LargeRDFBench.

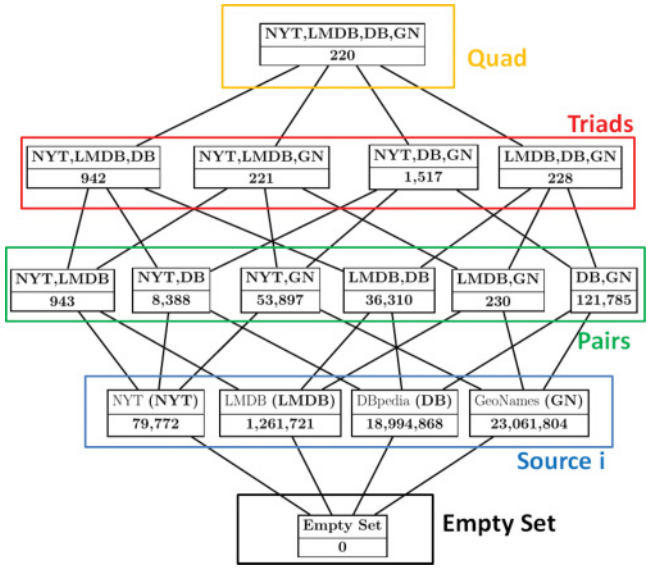

Fig. 2. Lattice of four datasets showing the common real-world objects.

equivalence relation, and therefore its transitive closure has to be computed. This task is challenging in global scale, since most of the algorithms that compute the transitive closure require a lot of memory, i.e., one should keep in memory all the binary relationships (in our case, the `owl:sameAs` relationships) during the computation of closure. More concrete motivating examples of the aforementioned tasks are given in Section 2.2.

It is not hard to see that it is very expensive to perform all these measurements and computations straightforwardly, because there are numerous datasets and some of them are quite big. Moreover, the possible combinations of datasets are exponential in number. To tackle this challenge, we introduce a set of indexes (and their construction algorithms) for speeding up such tasks. We show that with the proposed method it is feasible to perform such measurements for millions of triples even with one machine. Moreover, in comparison to Reference [33], in this article, we introduce parallelized versions of the corresponding processes that can leverage more than one machine for being able to exploit a cluster of machines and further speed up these processes. This allowed us to perform measurements over billions of triples. In brief, in this article:

- We introduce a *namespace-based prefix index* for speeding up the process.
- 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).

In comparison to Reference [33], in this article:

- We introduce parallel methods, implementable over big data frameworks such as *MapReduce* [8] and *Spark* [48], for constructing the indexes for billions of triples and performing the

measurements even for trillions of lattice nodes, while we introduce a new index called `SameAsPrefixIndex`.
- We measure the speedup obtained by the proposed indexes and algorithms by using (a) one machine (just indicatively, they enable computing the sameAs closure of 13 million `owl:sameAs` relationships in 45s and the lattice of measurements for more than 1 billion nodes in 35min) and (b) a cluster of machines (indicatively, by using 64 machines, 22min are needed for the construction of all the indexes and approximately 1min for computing a lattice of 1 billion nodes).
- We report connectivity measurements for a subset of the current LOD that comprises 302 datasets and billions of triples (in comparison to Reference [33] here we include approximately 3 times more triples, URIs and `owl:sameAs` relationships).
- We provide comparative results by using two big data frameworks, i.e., *MapReduce* [8] and *Spark* [48], and we show how the algorithms can be exploited for supporting different Resource Description Framework (RDF) features.

The indexes and measurements that are introduced in this article are exploited by a research prototype, i.e., http://www.ics.forth.gr/isl/LODsyndesis/, that 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 three-dimensional (3D) visualization. Furthermore, the up-to-date measurements concerning the commonalities among RDF datasets have been published in *datahub.io*[4] in RDF format by using VoID [29] and VoIDWH [31, 32] vocabularies.

The rest of this article is organized as follows: Section 2 provides the required background, introduces motivating examples, and describes related work, Section 3 states the problem and introduces the indexes and their construction algorithms, and Section 4 shows how to compute the metrics for any set of datasets. Section 5 discusses the speedup obtained with the introduced indexes. Afterwards, Section 6 introduces scalable indexes and methods by using *MapReduce* framework [8], while Section 7 reports measurements and experimental results over a big number of datasets of the LOD cloud and provides comparative results showing the scalability obtained with the parallel algorithms. Then, Section 8 discusses how the proposed approach can be implemented over different frameworks (Spark) and how it can support other RDF features, and, finally, Section 9 concludes the article and identifies possible directions for future research.

2 BACKGROUND, MOTIVATION, AND RELATED WORK

2.1 Background

The RDF [30] is a graph-based data model for linked data interchanging on the web. RDF uses `Triples` 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 $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{B})$, 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

[4] http://datahub.io/dataset/connectivity-of-lod-datasets.
[5] http://www.w3.org/TR/rdf-schema/.
[6] http://www.w3.org/TR/owl2-overview/.

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 Motivating Scenarios

Here, we introduce motivating scenarios for the four tasks described in the introductory section.

(a) Dataset Discovery and Selection. In many scenarios, dataset discovery is the first step of an integration process, and it is really difficult to find whether there are related datasets. Sometimes the metadata of these datasets are not enough. The work that is described in this article can assist the integration process, since it enables content-based dataset discovery services. The proposed measurements can be directly used for answering queries such as "find the K datasets that are more connected to a particular dataset." Such a query is important for finding related datasets (containing the same entities), either for constructing a warehouse or for mediator-based query answering. For instance, suppose that you publish a dataset and for connecting it to the rest datasets of the LOD cloud you establish relationships with DBpedia (by having triples that refer to URI's from DBpedia). Without the proposed approach, you would get that only one dataset is connected to your dataset (i.e., only DBpedia). With the proposed indexes and measurements (that include the computation of the transitive closure of owl:sameAs relationships), you could get much more datasets, i.e., datasets that you could not easily discover, because they could have in common only a few, or even no, URIs with your dataset. In many cases, it is also important to collect information about the same real-world entities from several sources to "widen" the information for the entities of interest. Such functionality can be exploited in the context of a Machine Learning task, specifically for finding more features about a given set of entities, which in turn could improve the accuracy of predictions (as was evidenced in Reference [34]). 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 discovering relevant datasets for such a scenario follows: "find the K datasets that maximize the pluralism factor of the entities of a particular dataset."

(b) Object Coreference. Suppose a scenario where one user or an application wants to find all the available data associated with "http://www.dbpedia.org/resource/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. Additionally, it is not hard to see that all these equivalence relationships can be exploited for improving the effectiveness of instance matching [39], which is an important requirement for effective integration [7]. We note that many linked datasets contain few external links, suggesting that the manual creation and maintenance of links is imposing a cost on publishers that they may be reluctant to bear.

(c) Data Quality Assessment And Improvement. As regards data quality, according to [5] quality is important for determining the subset of the available data that are trustworthy to use in a linked data application while [50] categorizes quality metrics in various categories. The major dimension in which our work belongs to is mainly *interlinking*, i.e., we measure the degree to which the datasets are linked to other ones by finding how many real-world objects they have in common and, secondarily, *relevancy*, i.e., we support "content-based" relevancy, since by having all the URIs for a given URI (or entity), one can find easily additional relevant information about this entity.

As it is stated in Reference [11], a mandatory component for resolving conflicts and improving the correctness (and in this way the quality) of data is to collect all the records (in our case URIs) referring to the same real-world entity, to fuse them into a single representation and then to apply

dedicated algorithms over the "integrated" content. Indeed, the "global scale"-like semantic indexing that we propose can be beneficial for estimating the veracity of data. The key point is that the collection of all information about an entity, and the cross-dataset inference that is offered, aids spotting the contradictions that exist and on the same time provides information for data cleaning or for estimating and/or suggesting which data are probably correct or more accurate. After having spotted such cases, several options are possible for improving quality: to inform the owners of the datasets for correcting the spotted errors or to produce an aggregated dataset with no conflicts by adopting a method for estimating the probability of correctness of each fact and/or selecting the more probable one (as it is also stated in References [10, 11]). Specifically, errors can occur in the values of specific real-world objects. For instance, in *d-nb.info*[7] it is written that *Aristotle's* birth year was 384 BC while in *DBpedia*[8] the value of the property *birthDate* for *Aristotle* is -383-1-1. The aggregation of all information enables estimating the more probable one.

Another kind of error is the inaccurate equivalence connections between two entities [17, 38], e.g., suppose that two entities representing two different real-world objects are connected through an inaccurate `owl:sameAs` or `skos:exactMatch` relationship (produced by an automatic instance matching method). Figure 3 shows a real example that contains 10 `owl:sameAs` relationships and eight URIs, where four of them refer to the entity *New York* (the URIs that are in bold in step 1) and the other four refer to *Constantia town* (which belongs to New York state). From the 10 `owl:sameAs` pairs, there is only a unique erroneous `owl:sameAs` relationship (see the fifth `owl:sameAs` pair in step 1) denoting that the entities *New York* and *Constantia town* are exactly the same. Due to the symmetric and transitive closure of `owl:sameAs` relationsips, all the URIs that refer to these two different real-world objects will be considered that they describe the same entity, making the problem more obvious. By exploiting the indexes that are introduced in this article, one could apply content-based algorithms for detecting possible erroneous relationships, i.e., one can first find all the triples for each of these URIs and compare their contents through similarity functions [6]. Alternatively, by having both the collection of all the `owl:sameAs` relationships and the URIs for a specific entity, one could exploit various graph-based metrics for detecting such cases [16, 27]. In the example of Figure 3, notice that every shortest path between a pair of instances u_1, u_2, where u_1 refers to New York and u_2 to Constantia town, passes from the erroneous edge. On the contrary, without having computed the transitive and symmetric closure, one should check every pair of `owl:sameAs` for detecting errors. Finally, in step 4 of Figure 3, we can observe how clustering (through a similarity function) could be used for providing evidence that half of the URIs refer to *New York* and the remaining half of URIs to *Constantia town*.

(d) Dataset Visualizations. Dataset visualization is an emerging challenge for the web of data; for instance, according to Reference [19], "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." A possible visualization exploiting the proposed measurements is shown in Figure 2. Such a visualization can be very useful for the users to identify the commonalities among a small set of datasets (as in the case of MarineTLO-based semantic warehouse [32]); however, in the case where there are many datasets, then a straightforward visualization is not suggested (since the combination of possible subsets of datasets is exponential in number). Another way to exploit the measurements for visualizing the commonalities among the datasets is shown in the lower right corner of Figure 4. In particular, we show a visualization of 287 RDF datasets that has been constructed by using the results of the measurements that are described in this article. This visualization has been constructed through a research prototype, called

[7] http://d-nb.info/gnd/118650130, accessed date: November 1, 2017.
[8] http://dbpedia.org/resource/Aristotle, accessed date: November 1, 2017.

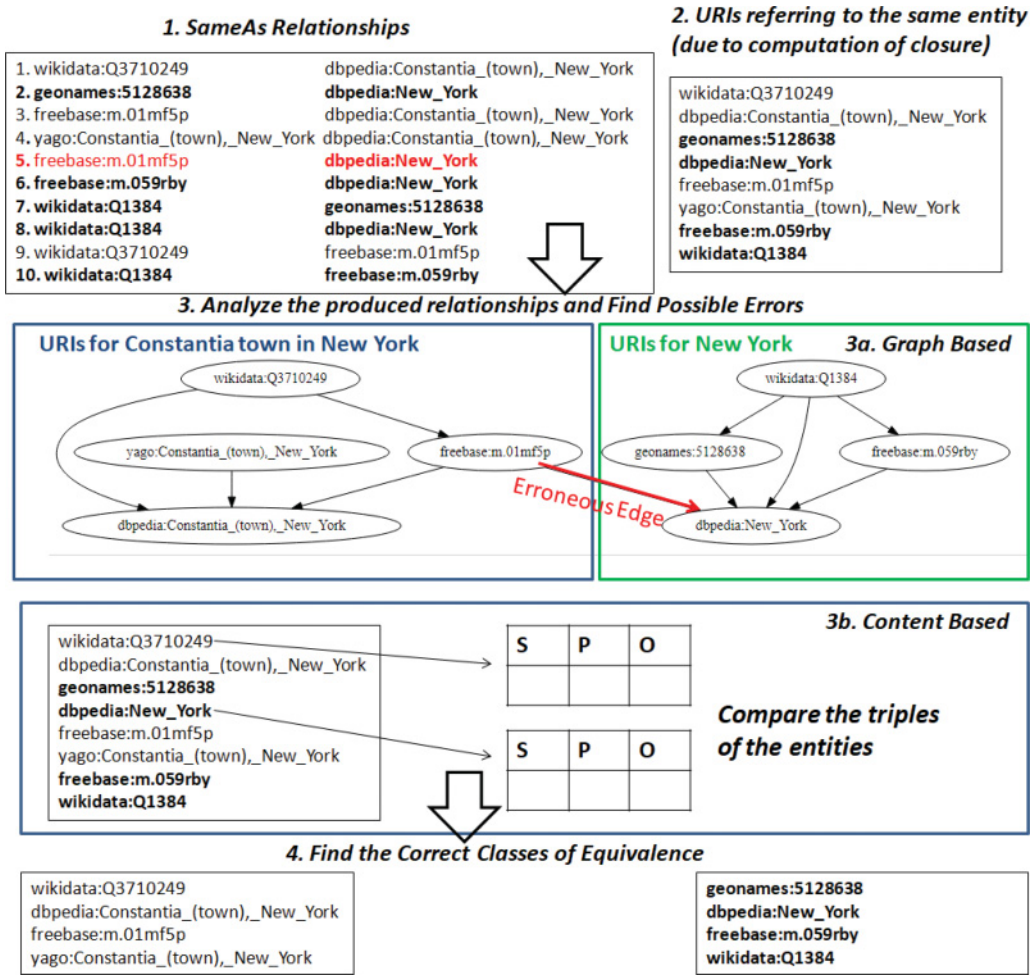

Fig. 3. Exploiting indexes for improving veracity.

3DLod,[9] which offers an interactive 3D visualization of LOD datasets. In particular, 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.

2.3 Related Work

Measurements in LOD Scale. The authors of Reference [43] indexed 38 billion triples from over 600,000 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 Reference [43] focuses on other aspects (like validity of documents), and they do not compute common URIs between three

[9] http://www.ics.forth.gr/isl/3DLod/.

or more datasets. The authors of Reference [35] created a portal for link discovery that 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 Reference [44] 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 [3], 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 Reference [14], the authors computed the PageRank for 319 datasets that was derived from LODLaundromat [43], where they found that the popularity of a specific source does not correlate to the size of a dataset. Such a measurement can show the popularity and to which extent other datasets trust a specific one; however, one cannot find the commonalities between a set of datasets. In Reference [32], 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, Reference [46] focuses only on features of the semantic web schemas, not on datasets.

Accuracy of RDF Datasets. A tool called LinkQA [16] uses a number of metrics (predominantly network metrics) to assess the quality of `owl:sameAs` linked data mappings regarding the dimension of *interlinking*. It can be used for detecting pathological cases, such as bad quality links, before they are published. Moreover, in Reference [27], the authors proposed network methods (such as clustering and edge betweenness) on top of instance matching systems to find errors in `owl:sameAs` relationships. Our indexes can be easily exploited for performing such metrics, since by having both all the `owl:sameAs` relationships and the "inferred" paths (due to the transitive and symmetric closure), one can apply such metrics for identifying incorrect links that produce inaccurate mappings between different entities. Another system, called ALEX [12], exploits user feedback on queries over linked data to remove incorrect links and discover new relationships that did not exist between pairs of datasets. They used real datasets for the measurements and they discovered new links while they improved the precision and recall for many pairs, especially because of the removal of incorrect links. In References [1] and [49], one objective of the authors was to check the correctness of links from *DBpedia* to a number of other sources. They keep only the correct links with the aid of user feedback to improve the precision and recall of such links. Furthermore, in Reference [6], the authors tried to improve the quality of `owl:sameAs` links by proposing an algorithm that uses a similarity matrix with pairwise similarity values. The aforementioned approaches can be assisted by our indexes, i.e., by having all the equivalent URIs for a given one and all the triples for these URIs, it is easier for the users to provide feedback by identifying disagreements with other datasets, since they can find all the available information for this set of URIs. Finally, the authors in Reference [10] identify as a key step the discovery of all the equivalent records for a given entity and the collection of all the available information about it to improve the trustworthiness of data.

Indexes for search and queries. The work described in Reference [20] proposes an object consolidation algorithm that analyses inverse functional properties to identify and merge equivalent instances in an RDF dataset. An index for the `owl:sameAs` relationships is adopted to find all the URIs for a real-world object. This index is used in *YARS2* [18], 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 to allow direct lookups on multiple dimensions

without requiring join. *YARS2* is part of the Semantic Web Search Engine (*SWSE*) [21], 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* [9], 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 Reference [47]. In particular, the authors construct an index for finding for each URI the documents in which it occurs, a keyword index, and an index that allows lookup of resources with different URIs identifying the same real-world object. Reference [36] describes an engine for scalable management of RDF data, called *RDF-3X*, which is an implementation of SPARQL [41]. 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 such measurements faster, while the focus of aforementioned approaches is not on speeding up measurements but on answering faster different types of user queries.

2.4 Services for Large in Number Linked Datasets

In this section, we focus on services over hundreds of linked datasets. We use the name *LODsyndesis* to refer to the indexes and measurements that are introduced in this article. As can be seen in Table 1, we categorize the available services according to the number of datasets that they include and to the services that they offer, while we compare all these services with *LODsyndesis*.

LODsyndesis [33] provides query services and measurements that are useful for several important tasks like (a) object co-reference, (b) dataset discovery, (c) visualization, and (d) connectivity assessment and monitoring. In addition, this page provides measurements that concern the commonalities of linked datasets by taking into account the semantics (e.g., equivalence relationships between the entities).

LODLaundromat [43] offers fetching and transforming datasets for cleaning them and providing statistics about them (like validity of documents). In this service, one can find 38 billion triples from over 600,000 documents. Similarly to our work, it offers a Global URI Lookup where one can find the documents where a URI or a namespace exists. On the contrary, by using the Global URI Lookup Service of *LODsyndesis*, one can also find all the equivalent URIs of a given one. Moreover, one can discover URIs and triples (and datasets) containing a specific keyword, since *LODLaundromat* offers a Global Keyword Search Engine (called *Lotus* [22]), while by using *LODsyndesis* one can discover the most connected subsets (e.g., pairs, triads, quads) for any given dataset.

LODStats [3] analyzes a big number of linked datasets and provides useful statistics about each document like the number of triples, number of `owl:sameAs` links, and so forth. LODsyndesis covers a subset of the datasets that can be found in *LODStats*. By using *LODStats*, one can find datasets through a keyword search and metadata for datasets, schema elements and namespaces. On the contrary, we index the URIs and we offer services for finding where a real-world object occurs and that datasets are the most connected to a given one.

Datahub[10] provides searching of data, registering published datasets, and creating and managing groups of datasets. One can find some major statistics for each dataset (e.g., number of triples and number of links to other datasets), while one can fetch datasets provided in dumps and get updates from datasets and groups that one is interested in. *LODsyndesis* offers services for a subset of the RDF datasets that have been published in *Datahub*. However, *Datahub* is a generic portal

[10] datahub.io.

Table 1. Categorizing Existing RDF Services for Big Number of Datasets, *Documents instead of datasets; Mil.=million, Bil.=billion, Unk.=unknown

Tool/Service	Total Triples	Include >10 Datasets	Include >100 Datasets	Include >1000 Datasets	Include >10000 Datasets	Global URI Lookup	Dataset Discovery	Dataset Visualization	Connectivity	Fetching Datasets	Keyword Search	Dataset Analysis	Query Datasets	Dataset Dynamics
LODsyndesis [33]	658 Mil.	✓	✓			✓	✓	✓	✓					
LODLaundromat [43]	38 Bil.	✓*	✓*	✓*	✓*	✓	✓			✓	✓			
LODStats [3]	130 Bil.	✓	✓	✓			✓					✓		
Datahub.io	Unk.	✓	✓	✓			✓			✓		✓		
LinkLion [35]	77 Mil.	✓	✓						✓	✓				
Europeana [23]	1.6 Bil.	✓	✓	✓									✓	
DyLDO [25, 26]	Unk.	✓*	✓*	✓*	✓*									✓
LODCache	4 Bil.	✓*	✓*	✓*	✓*						✓		✓	
LODCloud [44]	Unk.	✓	✓	✓				✓	✓			✓		
sameAs.org [15]	Unk.	✓	✓			✓								

containing datasets (and their metadata) from many organizations and different formats, while we mainly focus on measurements over a big number of RDF datasets.

LinkLion [35] is a portal that contains mappings between pairs of 462 datasets. In that repository, one can fetch mappings between pairs of datasets and metadata concerning *Dataset Interlinking*. Similarly to our work, one can find relationships (e.g., `owl:sameAs`) between two datasets while most of the datasets covered by *LODsyndesis* can be found also in *LinkLion*. In comparison to our approach, they take into account only pairs of datasets, and they do not index the URIs. Furthermore, *LODsyndesis* offers 119% more mappings for pairs of datasets, i.e., in *LODsyndesis*, one can find mappings for 7,119 pairs of datasets, whereas *LinkLion* offers mappings for 3,247 pairs of datasets.

Europeana [23] combines data from than 3,000 institutions across Europe while these data were transformed into linked data. One can query the integrated content by using SPARQL queries. *Europeana* contains a large collection of datasets for cultural heritage, while *LODsyndesis* contains datasets from many domains and covers a small subset of datasets offered by *Europeana*.

DyLDO [25, 26] is a Dynamic Linked Data Observatory to monitor linked data over an extended period of time. This service collects frequent, continuous snapshots of a subset of RDF documents and provides interesting insights in how linked data evolve over time. *DyLDO* offers the above service for many datasets that can be also found in *LODsyndesis*. Moreover, it can be combined with *LODsyndesis*, since it would be very helpful to detect the datasets that change over time to update the indexes (and the measurements).

LODCache[11] crawls data and collects them in a SPARQL endpoint. One can query the content while it offers a URI and label lookup service. Both *LODCache* and *LODsyndesis* have many datasets in common; however, *LODCache* does not find the equivalent URIs for a given one and focuses mainly on answering users' queries.

LOD Cloud [44] provides statistics about the datasets and the domains where they belong. Except for the statistics, one can find the popular *LOD Cloud Diagram*[12] that shows the links that exist between pairs of datasets. *LODsyndesis* covers a subset of *LOD Cloud* datasets, i.e., 302 of 1,139 datasets that the current *LOD Cloud Diagram* contains.

SameAs.org[13] [15] is a global URI lookup service containing approximately 203 million URIs, while in our approach we have indexed approximately 380 million URIs (86% more URIs).

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 a running example, we will use four datasets each containing six URIs as shown in Figure 4 (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:

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

[11] http://lod.openlinksw.com.
[12] http://lodcloud.net.
[13] http://sameAs.org.

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

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

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.,

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

If B is a set of datasets, i.e., $B \in \mathcal{P}(\mathcal{D})$, then 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 4 (upper right). If R denotes a binary relation, then we shall use $C(R)$ to denote the transitive and symmetric closure of R. Consequently, $C(sm(D_i))$ stands for the transitive and symmetric closure of $sm(D_i)$, while $C(SM(B))$ 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 $C(sm(D_i))$, plus those URIs of U_i that do not occur in $C(sm(D_i)$ (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: $U_{temp}/\sim = \{\{u_1, u_3, u_4\}, \{u_2\}, \{u_5\}\}$. 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:

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

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

We are now ready to "semantically complete" 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, after having merged all the URIs referring to the same entity (derived by the transitive and symmetric closure of owl:sameAs relationships) to a single object, in all 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 article, 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 4 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 Figure 4.
- `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 Figure 4.
- `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 Figure 4.

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, then $pi(nm)$ 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 Section 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 Section 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, then a new identifier is assigned to both of them. For example, Table 2 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 3.
(2) If u has an identifier while u' has not, then u' gets the same identifier as u. Table 4 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, then u gets the same identifier as u'.
(4) If both URIs have the same identifier, then the algorithm continues.

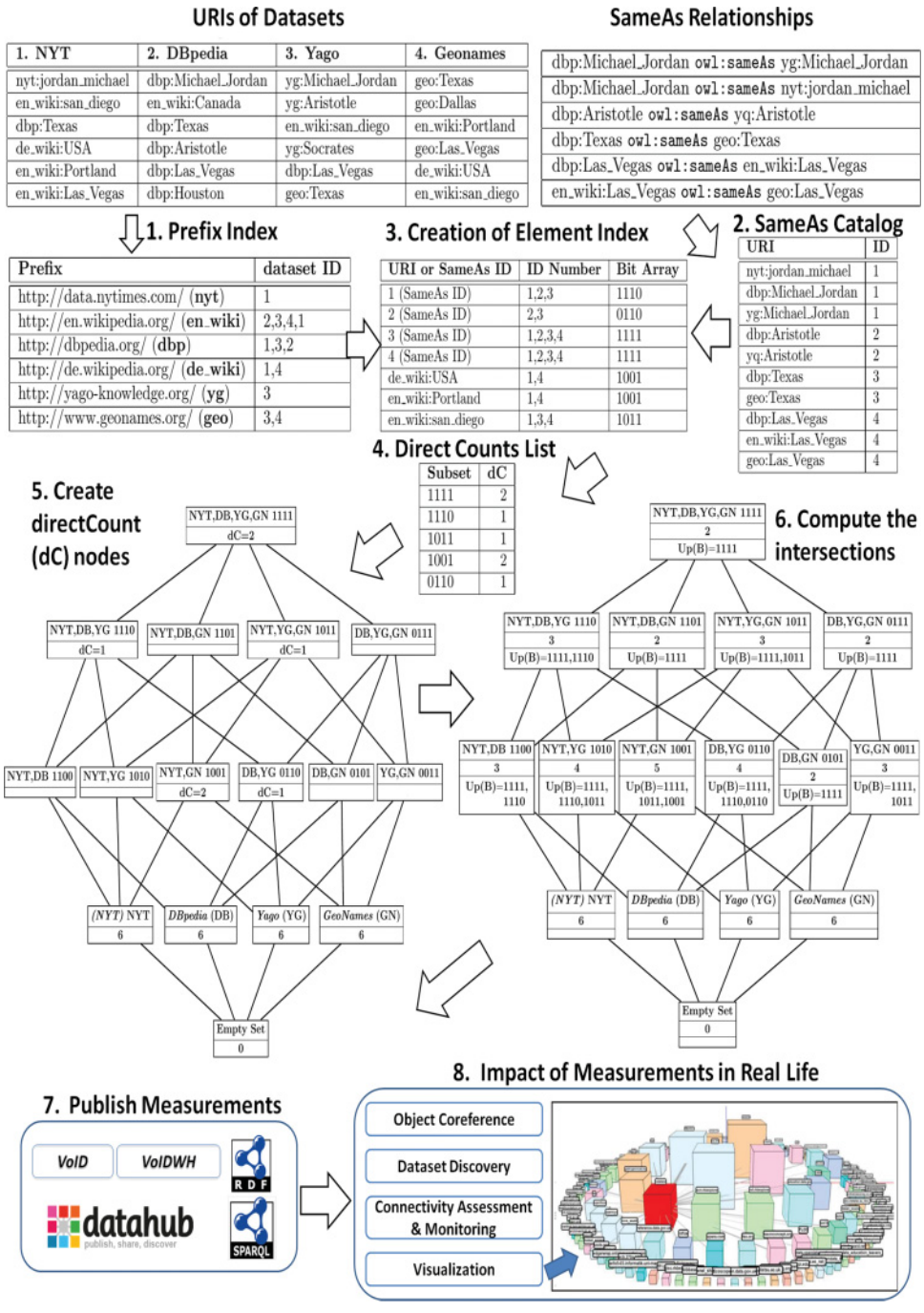

Fig. 4. Running example.

Table 2. Classes of Equivalence

ID	URIs
1	u_1, u_2
2	u_3, u_4

Table 3. Insert u_5 owl:sameAs u_6

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

Table 4. 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 5. Insert u_1 owl:sameAs u_3

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

(5) If the URIs have a different identifier, then these identifiers are concatenated to the lowest identifier. In case of Table 5, 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 that 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. First, $(b, a) \in C(SM(B))$ because of symmetry. By taking the transitive closure of $(b, a), (a, c)$, we get that $(b, c) \in C(SM(B))$. □

Its benefit is that it needs one pass for each owl:sameAs pair to compute the transitive and symmetric closure, and its time complexity is $O(n)$, where n is the number of owl:sameAs pairs. However, it is needed to keep in memory chains of owl:sameAs to connect such chains with new owl:sameAs relationships. Indeed, the space complexity is $O(m)$, where m is the number of unique URIs that occur in owl:sameAs pairs, since in the worst case each URI 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 4 (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 that 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 [45] that uses Depth-First Search (*DFS*). The input of *CC* algorithm is a graph that should have been created before running the algorithm. For this reason, we have to read all the owl:sameAs pairs once $O(n)$ (i.e., each owl:sameAs represents an edge) to construct the graph. The time complexity of *CC* algorithm is $O(m + n)$. 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 $O(m)$, 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 to run the *CC* algorithm, thereby the total space needed is $O(n + m)$. In Section 5.1, we compare the execution time of the two aforementioned approaches.

It is worth noting that the above algorithms can be also exploited for computing the closure of any property that is transitive and symmetric, e.g., one can apply the algorithms for finding the transitive and symmetric closure for the instances that are connected with skos:exactMatch property. Furthermore, variations of such algorithms can also be exploited for properties that are transitive and not symmetric (e.g., rdfs:subClassOf) or only symmetric.

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 in which a URI appears. To reduce its space, we can avoid storing URIs that occur in only one dataset, and this information can be partially 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 [51], 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 $|\mathcal{D}|$, and 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 $Bitmap \leq InvertedIndex$, then 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 4 (middle-right) shows the resulted index of our running example. The combination of the first and the second columns 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), then 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 *yq:Aristotle* exists in SameAsCat (see Figure 4). 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 and 9).

The last step (if the two previous failed) corresponds to URIs that belong neither to $Left(ei)$ nor to $Left(\texttt{SameAsCat})$. 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 that 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 the *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

ALGORITHM 1: Element Index Creation In a Single Machine

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  forall the D_i ∈ 𝒟 do
2      forall the u_j ∈ U_i do
3          if u_j ∈ Left(SameAsCat) then
4              if [u_j] ∉ Left(ei) then
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             forall the 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
```

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 Section 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 reads each $u \in U_i$ once for all $D_i \in \mathcal{D}$, thereby the complexity is $O(y)$, 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 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, and thereby the cost is $O(2^{|\mathcal{D}|} n \log n)$. In Section 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 6. Frequency for a Prefix p

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

Table 7. 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

Table 6, we can see the size of each U'_i for a prefix p. Table 7 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 7 follows: $Asks = |U'_{pos(0)}| * (n-1) + |U'_{pos(1)}| * (n-2) + \cdots + |U'_{pos(n-1)}| * 0$.

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 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 the first ASK query is 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, then 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' that 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 ($0 \leq k \leq |\mathcal{D}|$) 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, and hence be 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

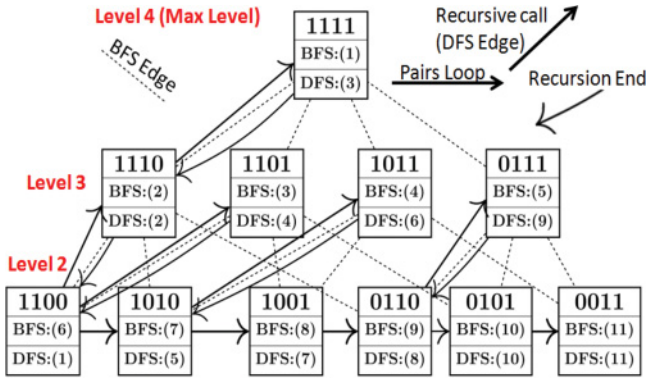

Fig. 5. Lattice traversal (BFS and DFS).

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 Section 5.1).

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 4). Now let $Up(B) = \{B' \in \mathcal{P}(\mathcal{D}) \mid B \subseteq B', directCount(B') > 0\}$. 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.,

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

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 $\bigcap_{S \in F'} S \subseteq \bigcap_{S \in F} S$. (The proof can be found in Reference [24].)*

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 5 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 4), 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 $directCount$, then we find $co_{\sim}(1001)$, which is 5.

The time complexity of this algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices ($|V| = 2^{|\mathcal{D}|}$) and $|E|$ the number of edges, and it holds that $|E| = |\mathcal{D}| * 2^{(|\mathcal{D}|-1)}$. 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 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 5 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 $O(|V|)$, 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 $O(d)$, 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 $B' \supseteq B \nRightarrow Up(B) \subseteq Up(B')$), e.g., in our running example $1110 \in Up(0110)$ but $1110 \notin Up(0111)$.

Cost analysis. If B_i in *Left*(*directCount*), then let $bits_1(B_i)$ be the number of 1's in B_i (e.g., $bits_1(1110)$ is 3), and obviously $2 \leq bits_1(B_i) \leq |\mathcal{D}|$. 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., $C = |\{ei(u) \,|\, u \in U \cup SID\}|$. It is essentially the cardinality of the codomain of ei and obviously, $C \leq |U \cup SID|$ (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 nth 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., $|\mathcal{D}| * 2^{|\mathcal{D}|-1}$) , then we get $(|\mathcal{D}| - 2) * 2^{(|\mathcal{D}|-1)}$. Now let us calculate *checkCost* for the assumed power-law distribution. The *nth* category (each node B_n of the nth 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

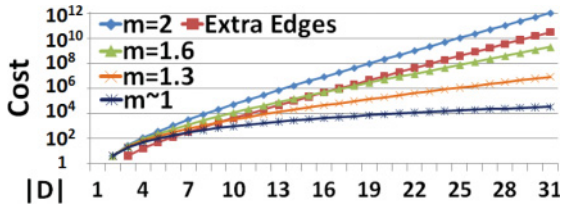

Fig. 6. CheckCost vs. extra edges for various m.

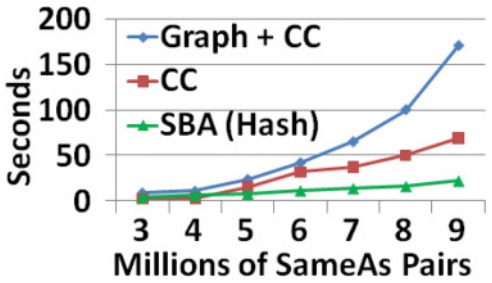

Fig. 7. SameAs catalog construction time.

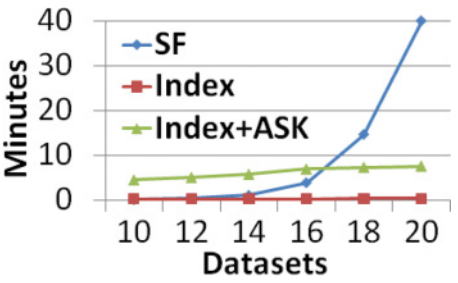

Fig. 8. Comparison of different approaches.

$checkCost = 4k * \frac{m^{|\mathcal{D}|-1}-1}{m-1}$. By assuming $k = |\mathcal{D}^2|/4$ (which is quite reasonable based on our experiments), we get the inequality of Proposition 3. □

As shown in Figure 6, 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) the 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 Section 5.1 are explained by this analysis, and we identify the same tradeoff 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 $O(|V| * C)$ (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 $O(C)$.
- *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 Proposition 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.

- *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—SINGLE MACHINE

Here, we report measurements that quantify the speedup obtained by the introduced techniques. We used a single machine having an i7 core, 8GB main memory and 1TB disk space and triplestore *Openlink Virtuoso*[14] *Version 06.01.3127* for uploading the datasets and for sending SPARQL ASK queries. For the below measurements, we used a subset of datasets and URIs of Section 7 (300 datasets, 658 million triples, and 172 million URIs) that are described in Reference [33].

5.1 Efficiency of Measurements

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

Savings by Prefix Index. In any implementation (e.g., index approach with or without ASK queries), the URIs starting with a prefix existing in one dataset can be ignored, and 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 Section 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 `owl: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 [45] that uses Depth-First Search (*DFS*) and was described in Section 3.4. We performed experiments for 3 to 9 million randomly selected `owl:sameAs` relationships and the results are shown in Figure 7. 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 Reference [2] to overcome this limitation. The signature-based algorithm needed only 45s to compute the closure of 13 million `owl:sameAs` pairs.

Index Approach vs. Straightforward Method. Here, we compare the execution time of the following three methods described in Section 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 described in Section 4. Figure 8 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 million. 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 17s to 23s. The *Index+ASK* approach needs more time compared 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

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

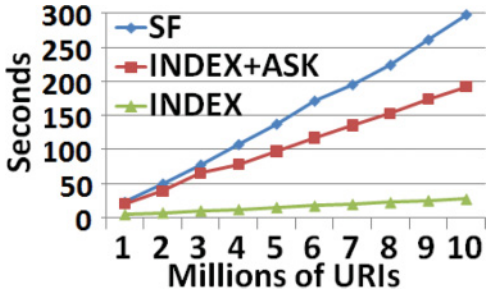

Fig. 9. Comparison with stable $|\mathcal{D}| = 17$.

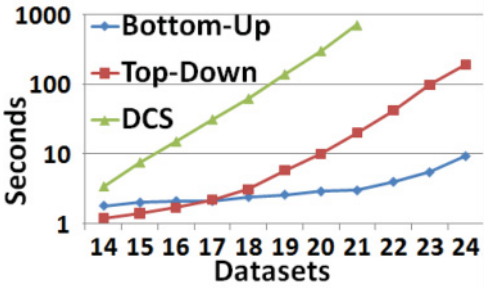

Fig. 10. Execution time of lattice creation.

increase so much when new datasets are added. In another experiment that is shown in Figure 9, 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 the subset of data that we used for the single machine process [33], where the data were infeasible to fit in memory, we used the *Index+ASK* approach, and the total execution time needed was approximately 7h.

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 described in Section 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 10, we can see that each incremental approach is faster than the *dcs* approach. Concerning the incremental approaches, one can clearly see the tradeoff between the two approaches. Indeed, for a 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 in approximately 35min.

6 PARALLELIZATION BY USING MAPREDUCE ALGORITHMS

Here, we show how to parallelize the creation of the indexes and how to parallelize the lattice-based measurements by using the *MapReduce* framework [8]. It is worth mentioning that all the algorithms that are presented in this section can be also implemented by using other frameworks such as *Spark* [48]. The key contribution is how to partition the problem to smaller ones, and we show how we can do it using MapReduce. The limitations of single-node process and therefore motivation for parallelization using several machines are the following: The signature-based algorithm needs a lot of memory, and thereby we were unable to compute the closure for over

14 million owl:sameAs relationships. Regarding the construction of ElementIndex, since we load the SameAsCat in memory, it is infeasible to scale to more than 14 million owl:sameAs triples. Moreover, for the construction of the ElementIndex, we send millions of ASK queries (for reducing the space), and the whole execution time increases as the number of ASK queries increase (e.g., it needs approximately 1min for 15,000 ASK queries). Finally, the computation of lattice measurements needs a lot of time as the number of datasets increases, because the number of lattice nodes increases exponentially. Therefore, in this section, we introduce parallel versions of the index construction algorithms and lattice-based algorithms. Moreover, we exploit the Hash-to-Min [42] algorithm for computing the symmetric and transitive closure of owl:sameAs relationships, while we introduce a new index, called SameAsPrefixIndex, which is useful in the case of the parallel computation of SameAsCat . Figure 11 shows the running example (containing the same datasets as the running example of singe-node version, i.e., Figure 4) for the parallel version of the algorithms.

6.1 MapReduce Framework

MapReduce [8] is a popular distributed computation framework that is widely applied to large-scale data-intensive processing, primarily in the big-data domain. The computation takes as input a set of key-value pairs and produces a set of output key-value pairs. Each *MapReduce* program (or job) is carried out in two phases: a map followed by a reduce phase. Map and Reduce are two different functions that are user-defined that can become tasks that are executed in parallel. The Map function takes as input key-value pairs, performs a user-defined operation over an arbitrary part of the input, partitions the data, and produces a set of intermediate key-value pairs. Subsequently, all key-value pairs produced during the map phase are grouped by their key and are passed (shuffled to the appropriate tasks and sorted) to the reduce phase. During the reduce phase, a reduce function is called for each unique key and it processes the corresponding list of values and, finally, produces a set of output key-value pairs.

6.2 Overview of the Parallelization

Let m be the set of available machines. Our objective is to split the indexes in partitions so that each machine m_i to create a "slice" of each index. The first step of our process is to download the datasets and find the distinct URIs of each dataset. These tasks could be parallelized; however, we do not focus on these tasks, since the datasets are available in different formats and in different places (e.g., in dumps or in SPARQL endpoints), and their collection required manual work that cannot be automated for the time being. Concerning the distinct URIs of dataset D_i, it is straightforward to find them for a Dataset D_i by using a simple *MapReduce* program, i.e., we put as key each URI, and since each key is distinct, we just store in the reducer the distinct keys (URIs).

The parallelization of our approach starts from the construction of PrefixIndex, which needs one *MapReduce* round for being constructed. Moreover, we construct also a new index called SameAsPrefixIndex that needs also a single round. Next, we construct the SameAsCat by using the Hash-to-Min algorithm [42], which needs $O(logn)$ rounds. The fourth step is the creation of ElementIndex, which needs two rounds, while the last step, i.e., the computation of lattice measurements, needs a single round.

Let $UR = \{UR_1, \ldots, UR_m\}$ be a partition of the URIs where $UR_1 \cup \ldots \cup UR_m = U$ and let $SMP = \{SM_1, \ldots, SM_m\}$ be a partition of the owl:sameAs relationships where $SM_1, \ldots, SM_m = SM(\mathcal{D})$. For load balancing issues, we decided each machine m_i reads a subset of URIs UR_i; e.g., in the running example of Figure 11, we can observe that each machine reads a subset of URIs from all the datasets. We did not choose each machine to read the URIs of a whole dataset, since the size of the URIs of datasets varies: There are many datasets with low numbers of URIs and a few datasets

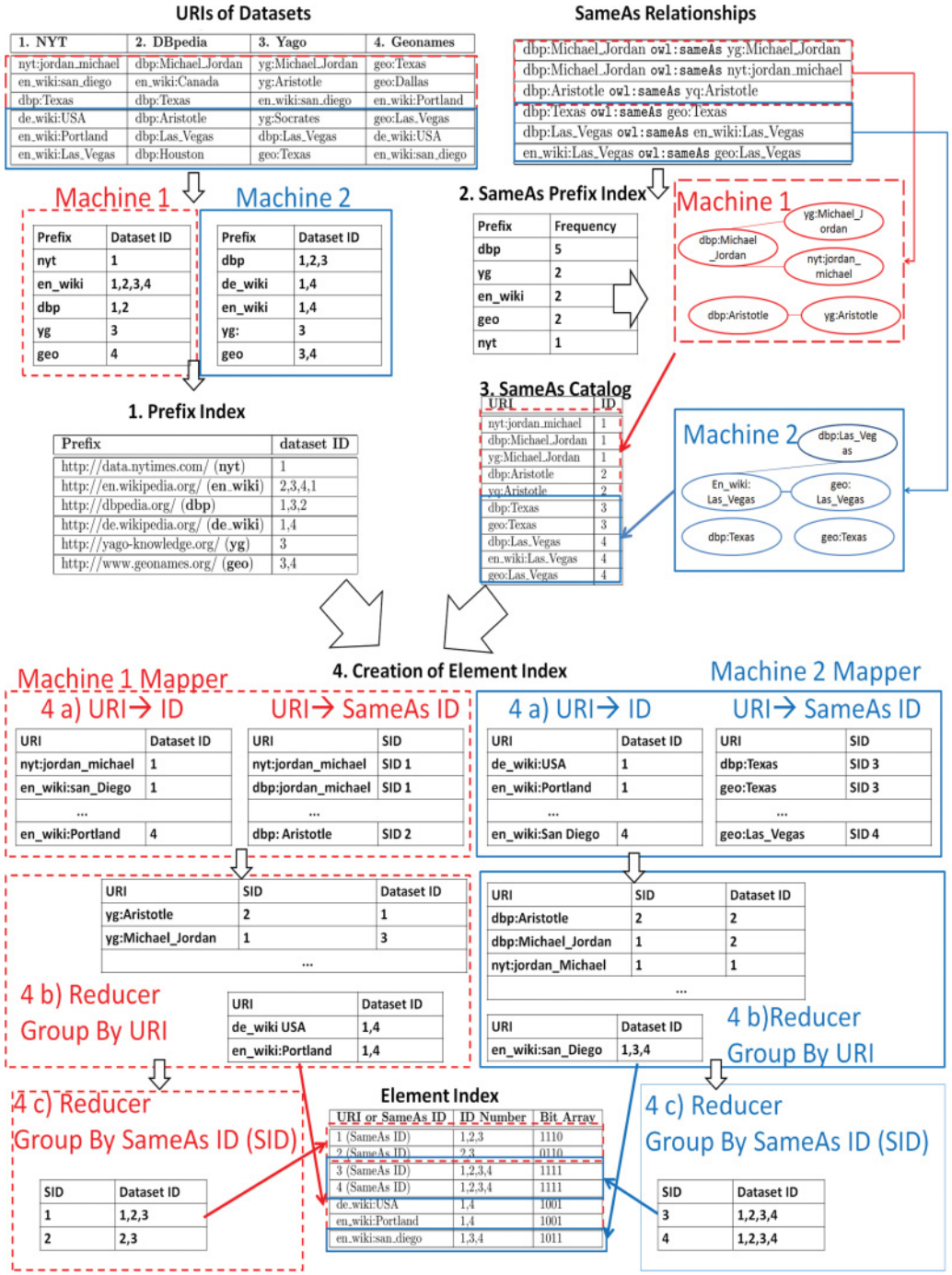

Fig. 11. *MapReduce* running example.

containing a large number of URIs, as we see in Figure 14. Moreover, for each different index, we define the following partitions:

PrefixIndex: Each machine m_i is assigned the responsibility of computing a function pi_i, where $pi_i: pre(\mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D})$. In the reducer, the original function pi can be derived by taking the union of partial functions, i.e., $pi = pi_1 \cup \ldots \cup pi_m$.

SameAsCat: Each machine m_i is assigned the responsibility to read a subset of owl:sameAs pairs and to compute a partial function $sm_i: U \cup SID \rightarrow Z$. In the reducer, SameAsCat can be derived as SameAsCat $= sm_1 \cup \ldots \cup sm_m$. From a wide range of proposed parallel algorithms [42], we selected to use the Hash-to-Min algorithm, since it computes the connected component in logarithmic rounds of *MapReduce* Jobs and has linear communication cost per round. However, we propose some optimizations that we apply to our context for reducing the number of *MapReduce* jobs and the size of intermediate data.

ElementIndex: Each machine m_i reads a set of URIs UR_i and a set of SameAsCat entries, and in the reducer it produces a function $ei_i: UR_i \cup SID \rightarrow \mathcal{P}(\mathcal{D})$. In the end of the algorithm, the original function ei can be derived as $ei = ei_1 \cup \ldots \cup ei_m$. Concerning the construction of ElementIndex, we propose a classical parallel inverted index approach that uses SameAsCat and PrefixIndex and needs two jobs to be constructed.

Lattice-Based Measurements: Concerning the lattice-based measurements, ideally we would like each machine to compute the measurements for the same (or approximately the same) number of lattice nodes, and this is quite challenging. Let LM_i denote the subset of lattice nodes for machine m_i. Ideally we would like $\forall i \in [1, \ldots, m]$, $|LM_i| = \frac{2^{|\mathcal{D}|}}{m}$. The challenge here is to split the lattice in "slices," where for each "slice" we want to be able to exploit the set-theory properties described in Section 4.1 and to perform the lattice-based measurements of its "slice" incrementally. For this reason, we create a parallelized version of the bottom-up incremental algorithm where we split the lattice in such "slices," where each "slice" of the lattice corresponds to the upper set or a part of the upper set of a specific node.

In particular, let $LP = \{L_1, \ldots, L_z\}$ denote a partition of the power set of $2^{|\mathcal{D}|}$ nodes, i.e., $L_1 \cup \ldots \cup L_z = \mathcal{P}(\mathcal{D})$, and if $i \neq j$, then $L_i \cap L_j = \emptyset$. Moreover, let θ be a threshold where $\theta = \frac{2^{|\mathcal{D}|}}{t}$, $t \geq 1$ (t can be given by the user). The size of the nodes that each "slice" contains is less or equal θ, i.e., $|L_i| \leq \theta$ and each machine m_i computes the measurements for a subset of lattice "slices" LP, i.e., $LM_i = \{L_j, \ldots, L_n\}$. Therefore, we split the power set of the lattice in z "slices" where z depends on θ; e.g., by having a big value for θ, fewer "slices" will be created, each of them containing a big number of nodes, whereas by choosing a small value for θ, a larger number of "slices" will be created, each of them having fewer number of nodes. As we shall describe later, each "slice" of the power set corresponds to a unique key-value pair and the "slices" are distributed randomly to the available machines.

6.3 Parallel Prefix Index

Rationale: For the parallel version of ElementIndex, a PrefixIndex can greatly reduce the communication cost to find common URIs, since there is no need to transfer to reducers URIs having prefixes existing in one dataset.

Construction Method: Initially, each machine reads a subset of URIs, i.e., UR_i, and creates a prefix sub-index pi_i for the prefixes occurring in UR_i. Then, for each $pre \in pi_i$, it emits a key value pair having as key the prefix pre and as value the $pi_i(pre)$, i.e., the dataset IDs where prefix pre occurs. Afterwards, in the reducer for a specific prefix pre (i.e., key) there exists a list of values $list(pi_j(pre))$, where $1 \leq j \leq m$. In the reducer, for each prefix pre, we just find $pi(pre) = \cup_{p \in list(pi_j(pre))} p$, i.e., the union of all dataset IDs where prefix pre occurs. For an

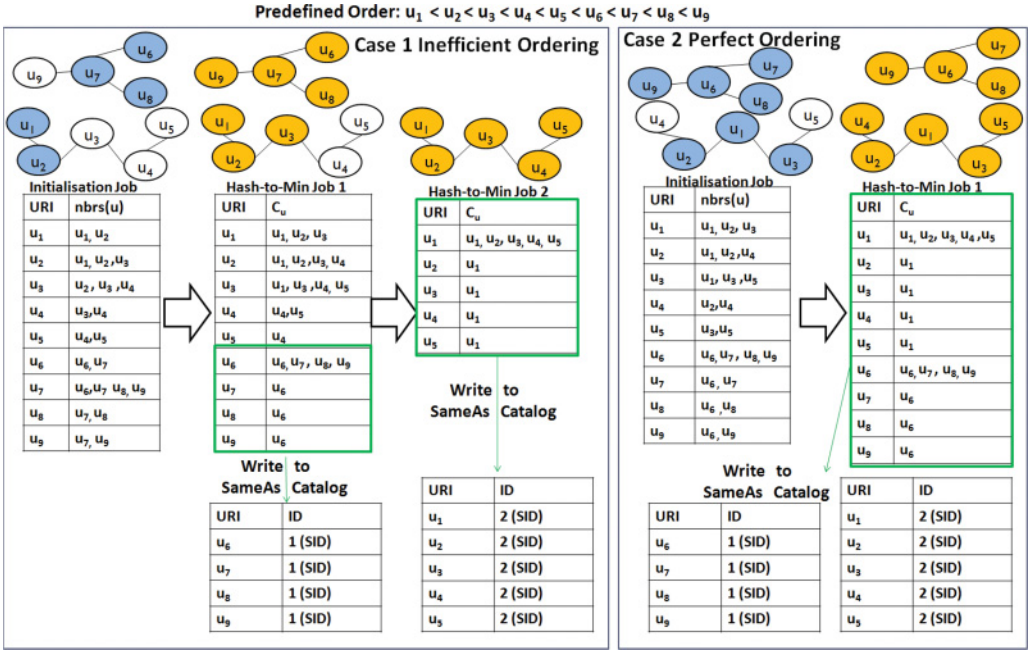

Fig. 12. Hash-to-Min algorithm example—impact of ordering.

example of this algorithm, see step 1 of Figure 11. Generally, a single *MapReduce* job is needed while its communication cost is $O(n)$, where n is the number of distinct prefixes.

6.4 Parallel SameAs Catalog

Rationale: We exploit the parallel connected components algorithm Hash-to-Min [42] for constructing `SameAsCat`, since it is essential for the construction of `ElementIndex`.

Construction method: Let G be a graph $G = \{V, E\}$, where V is the set of vertices, in our case $V = \{u \mid \{(u, \texttt{owl:sameAs}, u') \cup (u', \texttt{owl:sameAs}, u)\} \in \mathcal{T}\}$, and E is the set of edges, in our case $E = SM(\mathcal{D})$. The key notion of Hash-to-Min is that it creates one cluster C_u, where $C_u = Equiv(u, \mathcal{D}) \cup \{u\}$, for all the URIs belonging in the same class of equivalence (or connected component), and for each $u \in C_u$ it assigns to them the same signature (see an example in step 3 of Figure 11). The Hash-to-Min [42] algorithm takes as initial input a set of key-value pairs having key a URI u and value a set $nbrs(u) = \{u' \mid \{(u, \texttt{owl:sameAs}, u') \cup (u', \texttt{owl:sameAs}, u)\} \cup \{u\}\}$. Consequently, we need a single job that reads a "slice" of `owl:sameAs` pairs in the mapper and produces the set $C_u = nbrs(u), \forall\, u \in V$ in the reducer, e.g., see the initialization job of the example in Figure 12 (the vertices with the blue color represent the neighbors of u_{min}). We can observe in the aforementioned example that we have nine URIs and suppose that there exists the following predefined order $u_1 < u_2 < u_3 < u_4 < u_5 < u_6 < u_7 < u_8 < u_9$, thereby, for the cluster of vertices u_1 to u_5, u_1 is the u_{min} while for the cluster u_6 to u_9, u_6 is the u_{min}. It is worth noting that this example contains two cases having connected components with exactly the same structure. However, due to the impact of ordering (which is described in Section 6.4.1), the connected components of case 1 need one additional job to be computed comparing to the connected components of case 2.

Concerning the Hash-to-Min algorithm, it sends in its first job the entire cluster C_u to a reducer u_{min}, where u_{min} is the smallest node in the cluster C_u according to a global ranking (e.g.,

lexicographical order or any other prespecified order). Moreover, it sends a key-value pair (u', u_{min}) for each $u' \in (C_u \setminus u)$ to "inform" the other nodes of the cluster C_u about the smallest node inside the aforementioned cluster, i.e., u_{min}. In the reducer, the algorithm computes for each URI u_{min} the union of the cluster $C_{u_{min}}$ for a specific u_{min}, e.g., see job 1 in Figure 12 (the vertices having orange color represent the cluster of u_{min} in each job). When the cluster of the vertex with the minimum id contains the entire connected component while the cluster of other vertices in the component is a singleton having the minimum vertex, the connected component has been computed. Afterwards, for all the URIs $u \in C_{u_{min}}$, we assign them a new signature $[u]$ (i.e., they belong to the same class of equivalence), and we store $\forall u \in C_{u_{min}}$ in `SameAsCat` an entry comprising of URI u and [u]. For instance, in the first Hash-to-Min job of case 1 in Figure 12, the computation of connected component containing the URIs u_6 to u_9 has finished, since u_6 which is the u_{min} in this connected component contains the entire connected component while at the same time each of the other vertices of the connected component (i.e., u_7, u_8, and u_9) is a singleton having the minimum vertex (i.e., u_6). Afterwards, an entry in `SameAsCat` is created for each of these URIs, i.e., each of these URIs is assigned the same signature, since they belong to the same class of equivalence.

On the contrary, for the connected components that have not finished after a specific job, a new *MapReduce* job is performed (e.g., see the second Hash-to-Min job of case 1 in Figure 12 for the URIs u_1 to u_5), where we send the cluster again to the u_{min} while we "inform" the other nodes about the u_{min} (which is possibly different). In the reducer, the same process is performed again (i.e., we merge the clusters for u_{min}). The complexity, regarding the *MapReduce* rounds needed by this algorithm, is $O(logV)$, where V is the number of nodes in the largest component of a path graph (i.e., a tree with only nodes of degree 2 or 1 like in the case of URIs u_1 to u_5 in Figure 12) while its communication cost is $O(logn|V| + |E|)$ [42].

Combination with Signature-Based Algorithm: In many cases, the size of the classes of equivalence is small for most of real-world objects (as we shall see in Figure 22) while there exists only a few number of connected components needing more *MapReduce* jobs to be finished. To avoid performing *MapReduce* jobs for few number of large connected components, we can use a threshold t and when the number of remaining URIs is lower than t, we can send the remaining data to one machine and use the signature-based algorithm described in Section 3.4. In Section 7.1.1, we introduce comparative results showing the gain by combining the Hash-to-Min algorithm with the signature-based one.

6.4.1 Deciding the Global Ranking of Nodes for Connected Components—SameAs Prefix Index. **Rationale:** For many connected components parallel algorithms [42], it is important to decide a global ranking to "foresee" the centre of the connected component. For instance, in Figure 12, we can observe that for two graphs with the same structure, except for the initialisation job, in the first case we need two Hash-to-Min jobs for computing the connected components of the graph while in the second case we need one Hash-to-Min job. The difference is that in the second case, we always select as u_{min} the centre of the connected components while in the first one we select as u_{min} the URI having the lowest degree. Generally, if there is available information about the URI occurring as the center of a connected component (URI with the minimum radius), we can select as u_{min} this URI, however, such a pre-processing step can be expensive [28]. As it is stated in Reference [28], nodes with high degree usually have small radii and is efficient to be selected as the minimum node, i.e., u_{min}, of a connected component. In our case, instead of computing the degree of each URI (that requires an additional *MapReduce* job and to store and read the degree of each URI in each step), we can use the `SameAsPrefixIndex` (step 2 of Figure 11 shows the `SameAsPrefixIndex` for our running example), which is defined as follows.

SameAsPrefixIndex: It is essentially a function $sp : SAP(\mathcal{D}) \rightarrow \mathbb{Z}_{>0}$, where $SAP(\mathcal{D})$ is the set of prefixes of the owl:sameAs relationships in $\mathcal{D}$, i.e., $SAP(\mathcal{D}) = \{pre(u) \mid u \in SM(D), D_i \in \mathcal{D}\}$, and $\mathbb{Z}_{>0}$ is the set of positive integers. Therefore, this index stores the frequency of each prefix in all the owl:sameAs relationships. For instance, in the example of Figure 11 (see step 2), we can observe that the prefix *dbp* exists in five owl:sameAs relationships; thereby, we store this information in SameAsPrefixIndex.

By exploiting this index, we use the following heuristic rules to foresee the "center" of the connected component: (a) Always select the URI whose prefix occurs in the most owl:sameAs relationships (this information can be taken from SameAsPrefixIndex), and (b) if more than two URIs contain a prefix that occur in the same number of owl:sameAs relationships, then compare the URIs lexicographically. Therefore, by using this heuristic, we suppose that a URI with a prefix that occur in many owl:sameAs relationships will have a high degree.

Construction Method: For getting the prefixes that occur in owl:sameAs relationships and their frequency, we read each owl:sameAs pair once to get the prefixes and then we count in how many owl:sameAs pairs this prefix exists. Its construction is similar to PrefixIndex, and it needs a single job while its communication cost is $O(n)$, where n is the number of distinct prefixes occurring in all the owl:sameAs relationships.

6.5 Parallel 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 the datasets to which a URI appears quickly. To reduce (a) its space and (b) the communication cost in the case of the parallel implementation, we can avoid storing URIs that occur in only one dataset (this information can be obtained by the PrefixIndex).

Construction Method: This index needs two different phases to be constructed. In the first phase, we group the real-world objects by URI, to find the datasets where an exact URI occurs. In the second phase, we group the real-world objects occurring in SameAsCat by their signature, to find the datasets where a real-world object occurs.

Map phase. Initially, the algorithm, which is described in Algorithm 2, distributes PrefixIndex, i.e., *pi*, among the mappers while it reads a subset of URIs of a D_i and a subset of the SameAsCat. It checks whether the input is an entry of SameAsCat and in that cases it emits a key-value pair consisting of the URI as key and its signature as value (lines 3 and 4). On the contrary, if the input is a URI, it checks if the prefix of the URI belongs to $\geq$2 datasets, i.e., by accessing the PrefixIndex, and only in that case it emits a key-value pair having as key the URI and as value the ID of the dataset (lines 5-6). An example is shown in step 4 a) of Figure 11.

Reduce phase. The *MapReduce* framework then invokes the reduce function, which is described in Algorithm 2 (lines 7–18), which in turn takes as input a key-value pair $\langle URI, list(IDs) \cup SID \rangle$. It checks if there exists a signature for that URI (lines 11 and 12), meaning that the URI belongs to the SameAsCat. Moreover, it takes the union of $list(IDs)$ to find the distinct dataset IDs in which each URI occurs (lines 13 and 14). For the URIs being part of SameAsCat (lines 15 and 16), it emits a key-value pair having as key the signature of the URI and as value the datasets where URI u occurs (see step 4 b) of Figure 11). On the contrary, if we have a URI occurring in two or more datasets but this URI does not belong to the SameAsCat, then we store in ElementIndex an entry comprising of the URI and the datasets where it occurs (lines 17 and 18). However, since a signature corresponds to two or more URIs (that belong to the same class of equivalence) and in this job we grouped the real-world objects by URI (and not by signature), we need an additional simple *MapReduce* job to concatenate the dataset IDs where a real-world objects belongs. Indeed, we just emit a key-value pair consisting of $([u], list(dsets(u))$ and in another reducer function (see SIDReducer, i.e.,

ALGORITHM 2: Element Index Creation In Parallel

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  function Mapper (input = U_i ∪ SM(D_i))
2      forall the u ∈ U_i do
3          if u ∈ SameAsCat then
4              emit (u, SameAsCat(u))
5          else if u ∈ U_i and |pi(u)| ≥ 2 then
6              emit (u, i)
7  function URIReducer (URI u, values = list(IDs) ∪ [u])
8      dsets(u) ← ∅, u ← key
9      equivURI ← false
10     forall the v ∈ values do
11         if v = [u] then
12             equivURI ← true
13         else if v ∈ list(IDs) then
14             dsets(u) ← dsets(u) ∪ {v}
15     if equivURI = true then
16         emit ([u], dsets(u))
17     else if dsets(u) > 1 then
18         Store ei(u, dsets(u))
19 function SIDReducer ([u],list(dsets(u)))
20     dsets_~(u) ← ∅
21     forall the v ∈ list(dsets(u)) do
22         dsets_~(u) ← dsets_~(u) ∪ {v}
23     Store ei([u], dsets_~(u))
```

lines 19–23 of Algorithm 2) we find for a specific $[u]$ the union of the dataset IDs where each real-world object occurs, i.e., $dsets_{\sim}(u)$ (see step 4 c) of Figure 11). Generally, we ignore again (like in the case of single-node process) all the URIs that neither belong to `SameAsCat` nor occur in two or more datasets. As we can observe in Figure 11, from the steps 4 b) and 4 c) we construct `ElementIndex`; i.e., in step 4b we find the datasets where a URI that does not belong to `SameAsCat` occurs, while in step 4c we find the dataset IDs of a real-world object (having more than one URIs referring to it). Generally, two *MapReduce* jobs are needed while its communication cost for the first job is $O(n)$, where n is the number of URIs occurring in two or more datasets, and for the second job is $O(y)$, where y is the URIs that occur in `SameAsCat`.

6.6 Parallel Lattice Measurements

We decided to generalize the bottom-up approach, since (a) it was faster comparing to the top-down as the number of dataset grows; (b) it uses depth-first traversal, where there is no need to create all the edges of the lattice; and (c) its computation can be divided into the computation of the upper sets of different pairs. The proposed approach can be used for constructing either the whole lattice or a part of it (e.g., threshold-based nodes, lattice of a subset of datasets, etc.). As described in Section 4.1, the main notion of the *bottom-up* algorithm is that we always start from a subset B containing a pair of sources, and we compute the measurements of all the nodes of its upper set (including B) before continuing with the upper set of the next pair and so forth. Therefore, one possible way to traverse in parallel the lattice is to split the lattice into smaller "slices," where each

"slice" corresponds to the upper set of each pair. However, the size of the upper set of each pair (or triad, quad, and so on) is not the same. Let $Ups(B)$ be the upper set of a subset B, i.e., a subset of the supersets of B, where for each $B' \in Ups(B)$ there does not exist a dataset D_j in B' whose ID (i.e., j) according to the numerical order is lower than the dataset ID i for any $D_i \in B$. For instance, $\langle D_1, D_2, D_3\rangle$ is a superset of $\langle D_2, D_3\rangle$; however, it does not belong to $Ups(D_2, D_3)$, because the ID of dataset D_1, i.e., 1 is lower than the dataset ID for any dataset in $\langle D_2, D_3\rangle$ (i.e., 1 is lower than 2 and 3). Therefore, $Ups(B)$ is defined as follows: $Ups(B) = \{B' \mid B \subseteq B' \, s.t \nexists j > i, D_j \in B, D_i \in B' \setminus B\}$. The key notion is that the size of each "slice" (i.e., $Ups(B)$) is a power of two (as it is stated in Proposition 4), and for this reason we define (later in this section) a threshold θ that is always a power-of-two number.

Prop. 4. *Let $\mathcal{D} = \{D_1, \ldots, D_n\}$ and $B = \langle D_j, \ldots, D_k\rangle$, where $j < k \leq n$ and $n = |\mathcal{D}|$. The upper set of subset B, i.e., $Ups(B)$, contains 2^{n-k} subsets.*

Proof. The upper set of B contains the nodes $Ups(B) = \{B' \mid B \subseteq B' \, s.t \nexists j > i, D_j \in B, D_i \in B' \setminus B\}$. It means that each dataset that belongs to B also belongs also to B', i.e., for each $D_i \in B, D_i \in B'$ and each of the datasets that belongs to B' and not to B; i.e., for each $D_j \in B'$ where $D_j \notin B$, it holds that $j > k$. From the n datasets, $j > k$ holds for $|D'| = n - k$ in number datasets: $D' = \{D_{k+1}, \ldots, D_n\}$, where $(k + 1 < k + 2 < \cdots < n)$. Each $D_j \in D'$ can either belong to a particular subset $B' \in Ups(B)$, i.e., $D_j \in B'$ or not belong to B'; i.e., $D_j \notin B'$. It means that two different options exist for each dataset. Therefore, for $n - k$ datasets, the number of all possible combinations are $2 \times 2 \times \cdots \times 2 \, (n - k \, times) = 2^{n-k}$, i.e., $|Ups(B)| = 2^{n-k}$. □

In our case, we have $|\mathcal{D}|$ datasets where $\mathcal{D} = \{D_1, \ldots, D_n\}$, and for the pair $\langle D_1, D_2\rangle$, its upper set contains the power set of the set $\mathcal{D}' = \{D_3, \ldots, D_n\}$, i.e., $2^{|\mathcal{D}|-2}$ nodes that correspond to 1/4 of all the nodes ($\frac{2^{|\mathcal{D}|-2}}{2^{|\mathcal{D}|}}$). Indeed, in such a case one machine will compute at least 1/4 of the nodes regardless of the number of the available machines. For this reason, we propose a distributed version of the bottom-up algorithm where we split the power set into smaller "slices," where the size of each "slice" L_i is less than or equal to a threshold $|L_i| \leq \theta$, where $\theta = 2^{|\mathcal{D}|}/t$, $t \geq 1$. For instance, if t equals 16, each "slice" of the power set will be equal to or lower than $2^{|\mathcal{D}|}/16$ of the nodes of the power set. In Figure 13, we can see an example of six datasets and (2^6 nodes). We assume that there are 16 machines; thereby, we can choose $t = 16$, which implies that $\theta = 64/16 = 4$. It means that the size of each slice of the lattice that will be sent to the reducer will contain four or fewer nodes. It is worth mentioning that the size of each "slice" is always a power of two (as it follows Proposition 4), while for each "slice" we just emit a single key-value pair, as we shall see below.

Let $UpsR(L)$ be all the nodes belonging in the upper set of a subset L that the algorithm has not yet explored (obviously, $UpsR(L) \subseteq Ups(L)$). The *Bottom-up* parallel algorithm (shown in Algorithms 3 and 4) starts in the mapper by computing θ, traverses a subset of pairs, i.e., the set B_{pairs}, and calls the recursive function *BupMapper* for each subset L (a pair of datasets) that belongs in B_{pairs} (lines 3 and 4). The recursive function *BupMapper* computes the size of $Ups(L)$ (lines 6 and 7) and checks whether its upper set size is less than or equal to θ (line 8). In such a case (lines 9 and 10), it sends to the reducer a key-value pair having as a key the subset L and as a value the following two different parts: (a) $Up(L)$ and (b) a value zero that indicates that the reducer will compute the measurements for the whole upper set of subset L. In particular, when the algorithm reaches line 9, the reducer function (see the first function of Algorithm 4) will be triggered for computing the measurements for the upper set of L. Otherwise it assigns $|UpsR(L)| = |Ups(L)|$ (line 11) and then it starts to explore its supersets of the next level (lines 12 and 13).

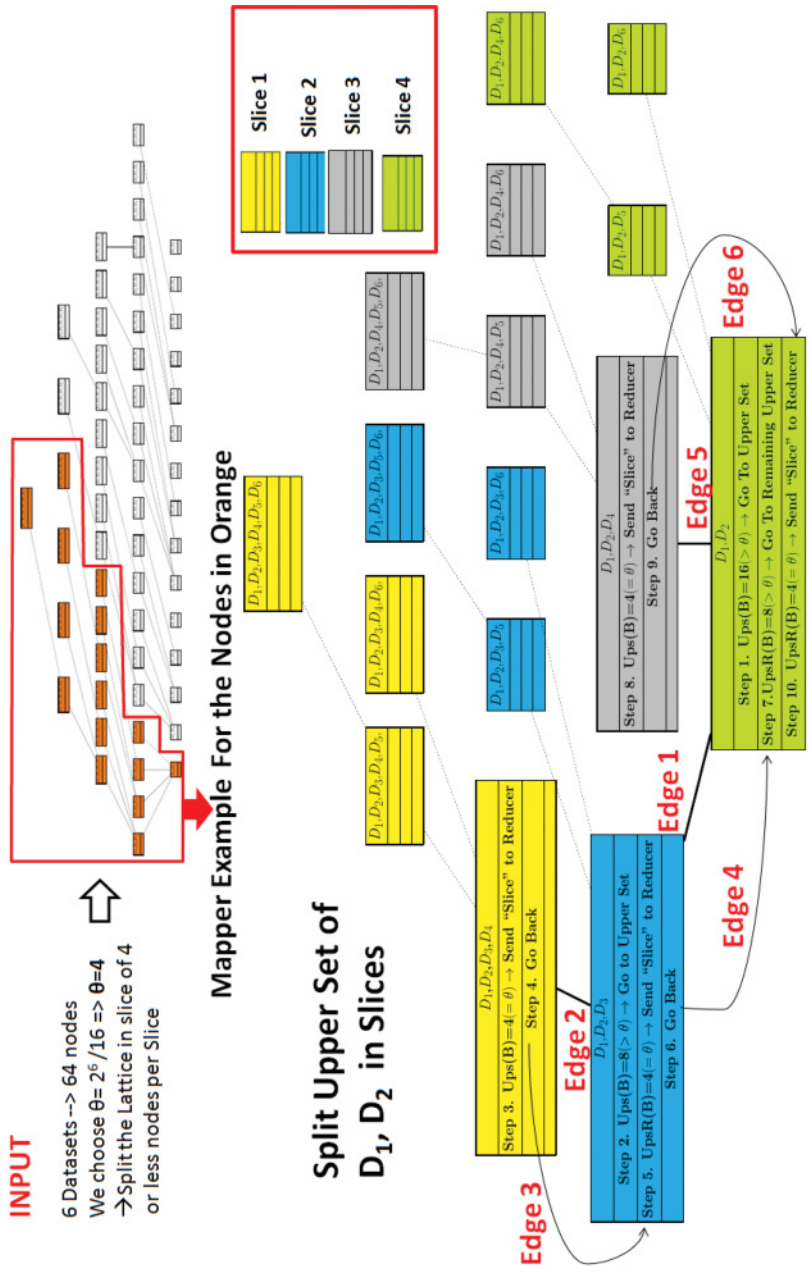

Fig. 13. Example of splitting the Upper Set of a set of nodes into four "Slices," each having four nodes.

Each time it checks whether $|UpsR(L)| \leq \theta$ (line 14) and, if yes (i.e., the number of the nodes of this upper set is equal to or lower than θ), it sends to the reducer the subset L, $Up(L)$ and a node B indicating the node from which we will start the computation of the measurements of the nodes belonging in $Ups(L)$ (lines 15 and 16). Specifically, when line 15 is triggered, the aforementioned key-value pair will be given as an input to the Reducer function (see the first

function of Algorithm 4). Otherwise, it continues upwards (since we follow a Depth-First Search traversal) with a subset B, where $B \subset L, B \in UpsR(L)$ and calls the recursive function again (lines 17–20). The sequence that is followed for traversing the supersets of the next level of a subset (e.g., the triads of a quad) is the numerical ascending order. For instance, for the subset $\langle D_1, D_2\rangle$, it first visits $\langle D_1, D_2, D_3\rangle$ and next $\langle D_1, D_2, D_4\rangle$, since D_1 and D_2 occur in both supersets, while for the datasets D_3 and D_4 it holds $3 < 4$ according to the numerical ascending order.

Figure 13 shows an example where we have selected $\theta = 4$. At first (see step 1 of node $\langle D_1, D_2\rangle$), $|Ups(D_1, D_2)| > \theta$; therefore, we continue with the triad $\langle D_1, D_2, D_3\rangle$ (i.e., edge 1 is created). In the case of $\langle D_1, D_2, D_3\rangle$ (see step 2), the size of its upper set is greater than θ, and therefore we continue upwards with the quad $\langle D_1, D_2, D_3, D_4\rangle$, i.e., edge 2 is created. Then, the size of the upper set of the quad $\langle D_1, D_2, D_3, D_4\rangle$ is equal to θ (see step 3), so we send this "slice" (i.e., nodes in yellow) to a reducer. Therefore, for this "slice," Algorithm 3 sends as a key the subset $\langle D_1, D_2, D_3, D_4\rangle$ and a value with two parts: (a) $Up(D_1, D_2, D_3, D_4)$ and (b) a value "0" indicating that for this "slice" we should perform the measurements for all the nodes belonging in the upper set of $\langle D_1, D_2, D_3, D_4\rangle$. Afterwards, we return to the previous node (see step 4), i.e., $\langle D_1, D_2, D_3\rangle$ (see the edge 3 in Figure 13), and we check if the remaining nodes of its upper set, i.e., $|UpsR(D_1, D_2, D_3)| = |Ups(D_1, D_2, D_3)| - |Ups(D_1, D_2, D_3, D_4)|$, is less than or equal to θ (see step 5). In particular, $|UpsR(D_1, D_2, D_3)|$ equals θ, and therefore we send to the reducer the remaining upper set of the node $\langle D_1, D_2, D_3\rangle$ in a reducer (nodes in cyan color). Concerning Algorithm 3, in this case it emits a key-value pair consisting of subset $\langle D_1, D_2, D_3\rangle$ as a key and a value with two parts: (a) $Up(D_1, D_2, D_3)$ and (b) $\langle D_1, D_2, D_3, D_5\rangle$ (starting node). In this case, we would like to compute the measurements for the upper set of $\langle D_1, D_2, D_3\rangle$, but we should exclude from the computation the upper set of $\langle D_1, D_2, D_3, D_4\rangle$, which will be computed separately, i.e., see "slice" 1 of Figure 13. For this reason, we put as a starting node the subset $\langle D_1, D_2, D_3, D_5\rangle$ for starting exploring in the reducer the upper set of $\langle D_1, D_2, D_3\rangle$ from this node and not from $\langle D_1, D_2, D_3, D_4\rangle$. Afterwards, we return to node $\langle D_1, D_2\rangle$ (see step 6 and edge 4), and we check in step 7 whether $|UpsR(D_1, D_2)| < \theta$. Since it is greater than θ, edge 5 is created, and we reach the node $\langle D_1, D_2, D_4\rangle$. The size of the upper set of this node equals θ (see step 8), and we send its upper set (nodes in gray) to a reducer. Finally, we return again to node $\langle D_1, D_2\rangle$ (see step 9 and edge 6), and we send its remaining upper set (nodes in green) to a reducer (see step 10), since $|UpsR(D_1, D_2)|$ equals θ. Therefore, each time we reach a node B whose upper set is equal to or lower than θ, we send a key-value pair to the reducer for computing the measurements for the subsets of $Ups(B)$. Then, we return back to its subset of the previous node and we recheck the size of the "remaining" upper set of the previous node B. Then, we send to the reducer to compute the "remaining" upper set nodes of B when $|UpsR(B)| \leq \theta$.

We can observe in Figure 13 that we finally split the upper set of $\langle D_1, D_2\rangle$ in four different "slices" (and, consequently, four key-value pairs), where the number of the nodes of all of these parts equals 4. Each of these "slices" corresponds to a unique key-value pair, and the "slices" are distributed randomly to the available machines. Therefore, the communication cost (from the mappers to the reducers) of the proposed algorithm is $O(z)$, where z is the number of "slices."

In the reducer (see Algorithm 4), the bottom-up algorithm is used for computing the measurements for the nodes of each "slice." In particular, it computes $co_{\sim}(L)$ (line 2 in the right column) and then it starts exploring the upper set of L by calling the recursive function *BupReducer* (line 3). This function computes the measurements for a specific part of $Ups(L)$, i.e., $UpsR(L)$, starting from a specific subset belonging in $UpsR(L)$, i.e., the "starting" node (line 6). Then, for a superset B, it assigns $Up(B)$, where $Up(B) \subseteq Up(L)$, it finds $co_{\sim}(B)$ for a specific superset and continues upwards (lines 6–8). For instance, for the subset $\langle D_1, D_2\rangle$ of Figure 13, Algorithm 4 computes its intersection value (line 2), i.e., $co_{\sim}(D_1, D_2)$. Afterwards, *BupReducer* function is

called (line 3) to explore its "remaining" upper set starting from $\langle D_1, D_2, D_5 \rangle$. In particular, we ignore $\langle D_1, D_2, D_3 \rangle$, $\langle D_1, D_2, D_4 \rangle$, since the aforementioned nodes and their upper sets belong to other "slices," and the measurements for them will be computed in other reducers. As we shall see in the experiments, with the parallel version of bottom-up algorithm, we are able to compute the measurements for the whole lattice for billions of nodes in less than 1min and the measurements of a lattice of trillions of nodes in approximately 6h.

ALGORITHM 3: *Bottom-up* Parallel Incremental Algorithm Mapper

Input: Pairs B_{pairs}, Up of pairs and threshold t
Output: Splitting the Lattice in "Slices"

1 **function** *Mapper* (B_{pairs},*Up of pairs,t*)
2 $\theta \leftarrow \frac{2^{|\mathcal{D}|}}{t}$
3 **forall the** $L \in B_{pairs}$ **do**
4 BupMapper($L, Up(L), \theta$)
5 **function** *BupMapper (subset L,Up(L),*θ*)*
6 $D' = \{D_i \in \mathcal{D} | \forall D_k \in L, D_i > D_k\}$
7 $|Ups(L)| \leftarrow 2^{|D'|}$
8 **if** $|UpsR(L)| \leq \theta$ **then**
9 $emit(L, Up(L)|sNode \rightarrow 0)$
10 $return|Ups(L)|$
11 $|UpsR(L)| \leftarrow |Ups(L)|$
12 $L' \leftarrow \{B_i | B_i \supset L, |B_i| = |L| + 1, B_i \in Ups(L)\}$
13 **forall the** $B \in L'$ **do**
14 **if** $|UpsR(L)| \leq \theta$ **then**
15 $emit(L, Up(L)|sNode \rightarrow B)$
16 **return** $|Ups(L)|$
17 **else**
18 $Up(B) \subseteq Up(L)$
19 $|Ups(B)| \leftarrow$ $BupMapper(B, Up(B), \theta)$
20 $|UpsR(L)| \leftarrow |UpsR(L)| - |Ups(B)|$
21 **return** 0

ALGORITHM 4: *Bottom-up* Parallel Incremental Algorithm Reducer

Input: Subset L, $Up(L)$, starting node sn
Output: Computation of $co_{\sim}(B)$

1 **function** *Reducer* (*L,Up(L)* | *sNode sn)*
2 $co_{\sim}(L) = \sum_{B' \in Up(L)} directCount(B')$
3 BupReducer($L, Up(L) \mid sn$)
4 **function** *BupReducer* ($L, Up(L)|sn$)
5 $L' \leftarrow \{B_i \mid B_i \supset L, |B_i| = |L| + 1, B_i \in Ups(L)\}$
6 **forall the** $B \in L', B \geq sn$ **do**
7 $Up(B) \subseteq Up(L)$
8 $co_{\sim}(B) =$ $\sum_{B' \in Up(B)} directCount(B')$
9 BupReducer($B, Up(B), 0$)

7 EXPERIMENTAL EVALUATION—CLUSTER OF MACHINES

Here, we report the results of two kinds of experiments: (a) measurements that quantify the speedup obtained by the introduced *MapReduce* techniques and (b) interesting measurements over the entire LOD. We used a cluster in the okeanos cloud computing service[15] containing 64 cores, 64GB main memory and 480GB disk space. In total, we created 64 different virtual machines, each one having 1 core and 1GB of memory. Moreover, we used the *Apache Hadoop 2.5.2*[16] release. On the *LODsyndesis* website,[17] one can find the data that were used for performing the experiments and the code for reproducing the results.

[15] http://okeanos.grnet.gr.
[16] https://hadoop.apache.org/docs/r2.5.2/.
[17] http://www.ics.forth.gr/isl/LODsyndesis.

Table 8. Datasets Statistics

Domain	$\|\mathcal{D}\|$	|Triples|	|URIs|	|Literals|
Cross Domain **(CD)**	19	925,672,216	191,242,185	413,256,091
Publications **(PUB)**	78	640,752,256	123,479,239	358,748,594
Geographical **(GEO)**	14	151,485,048	41,746,235	58,917,978
Life Sciences **(LF)**	17	73,087,983	9,913,973	33,117,116
Government **(GOV)**	45	58,548,183	6,934,070	24,753,151
Media **(MED)**	9	15,775,101	4,450,664	7,056,373
Linguistics **(LIN)**	8	9,146,099	2,059,465	4,097,874
Social Networking **(SN)**	96	2,551,826	561,686	1,399,799
User Content **(UC)**	16	1,139,155	308,193	414,662
All	302	1,878,157,867	380,695,710	901,761,638

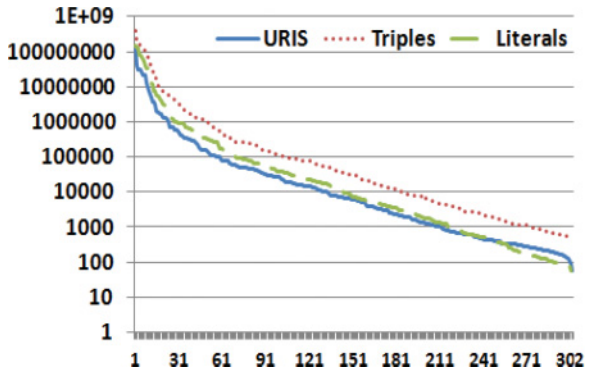

Fig. 14. Number of triples, URIs, and literals per dataset in descending order.

Datasets. The set of datasets that was used in the experiments contains 302 datasets that were collected from the following resources: (a) the dump of the data that were used in Reference [44], (b) online datasets from `datahub.io` website, (c) subsets of (i) *DBpedia* version 3.9, (ii) *Wikidata*, (iii) *Yago* and (iv) *Freebase*, and (d) a large number of `owl:sameAs` relationships from *LinkLion* website [35]. Table 8 shows the number of datasets, triples, URIs, and literals for each domain (in descending order with respect to their size in triples). This set of 302 datasets is adequately large, since it covers a big part of the current LOD cloud. Other approaches, like that in Reference [43], contains thousands of documents; however, in these approaches, even thousands of documents represents a unique dataset. It is mentioned in Reference [14] that the set of approximately 650,000 documents of Reference [43] corresponds to 319 unique datasets. Concerning our set of 302 datasets, most of them belong to the social networking domain; however, they contain a small number of triples and URIs compared to the datasets of the other domains. On the contrary, 83.4% of triples, 82.6% of URIs, and 85.6% of literals are part of cross-domain and publications datasets, although their union contains 97 (of 302) datasets. In Figure 14, we can observe that only 50 datasets have more than 1 million triples, while only 59 datasets have 100,000 or more URIs and 71 datasets over 100,000 literals. Moreover, most datasets (198 in number) have fewer than 100,000 triples, 168 datasets contain fewer than 10,000 URIs, and fewer than 10,000 literals are included in 158 datasets. Therefore, we can observe a power-law distribution for triples, URIs, and literals in these 302 datasets.

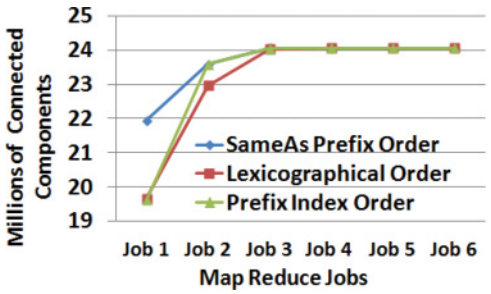

Fig. 15. Number of connected components having computed after the execution of each *MapReduce* Job per different order.

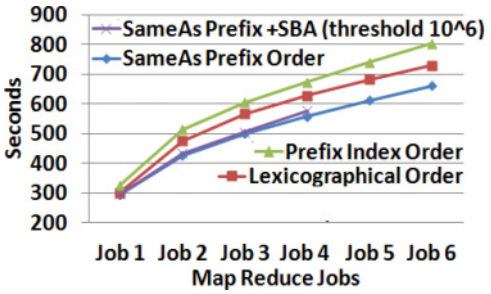

Fig. 16. Total execution time (in seconds) after the execution of each *MapReduce* job per algorithm variation.

7.1 Comparative Results

Here, we report measurements that quantify the speedup obtained by the parallelization methods and techniques.

7.1.1 SameAs Catalog. Here, we compare the execution time of the creation `SameAsCat` and the number of connected components computed after each job by selecting different global rankings of the URIs: (a) lexicographical order, (b) order by the frequency of each prefix in `SameAsPrefixIndex`, and (c) order by the frequency of each prefix in `PrefixIndex` (where we take into account all the URIs and not only the URIs being part of `owl:sameAs` relationships as in the case of `SameAsPrefixIndex`). In Figure 15, we show the number of connected components that were computed after each *MapReduce* job by using different order for "foreseeing" the center of the connected component. By using `SameAsPrefixIndex`, the Hash-to-Min algorithm was able to compute in the first job correctly approximately 22 million connected components, while, with the two other orders, it managed to compute approximately 19.5 million connected components. It is worth noting that the size of approximately 19 million connected components is two, i.e., there exist exactly two URIs for each of these 19 million real-world objects. For instance, in the example of Figure 11, for the entity *Aristotle* there exists two URIs, and therefore the size of the connected component of this entity is two, while for the entity *Michael Jordan* there exists three URIs, and, thereby, its connected component size its three. The computation of connected components having size two requires only one job (except for the initialisation job) regardless of the order that we use. However, the proposed order seems very effective compared to the other two approaches, especially for the connected components containing three or more URIs.

After the second job, both algorithm variations that use the order by the frequency of each prefix in either `SameAsPrefixIndex` or `PrefixIndex` had computed approximately 23.5 million connected components while the variation with the lexicographical order less than 23 million. In the remaining four jobs, there were a few number of connected components left, since most real-world objects belong to a connected component having either size two or three as can be observed in Figure 22. Concerning the execution time, as we can see in Figure 16, the variations of the algorithm using the `SameAsPrefixIndex` are faster comparing to the other ones. This is justified, since more connected components had been finished after the execution of each job. The best execution time achieved by combining the algorithm with the *Signature-Based* one, where the computation of connected components finished in fewer than 10min in four *MapReduce* jobs. More specifically, when fewer than 1 million URIs left, we used the *Signature-Based* for constructing the

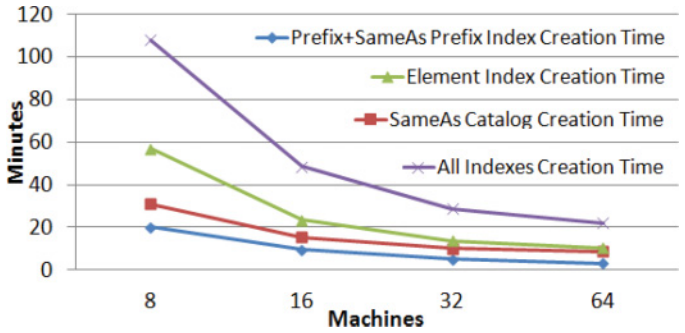

Fig. 17. Creation time of indexes for different number of machines.

Table 9. Lattice Measurements for 35 Datasets and 2^{35} Nodes (34.35 Billion Nodes)

Size of each "slice"	Number of "slices"	Maximum Nodes/ all Nodes from one m_i	Distance from Ideal (Ideal is 1.56%)	Execution Time (Minutes)
≤ 1/4 of all Nodes	595	25.10%	23.54%	185.00
≤ 1/8 of all Nodes	596	18.80%	17.24%	147.00
≤ 1/16 of all Nodes	600	12.60%	11.04%	95.00
≤ 1/32 of all Nodes	611	7.90%	25.1%	59.00
≤ 1/64 of all Nodes	637	6.00%	6.34%	45.00
≤ 1/128 of all Nodes	694	4.10%	2.54%	31.50
≤ 1/256 of all Nodes	814	3.10%	1.54%	25.00
≤ 1/512 of all Nodes	1,061	2.85%	1.29%	22.50
≤ 1/1024 of all Nodes	1,563	2.79%	1.23%	20.20
≤ 1/2048 of all Nodes	2,576	1.64%	0.08%	12.10
≤ 1/4096 of all Nodes	4,612	1.62%	0.06%	12.30
≤ 1/8192 of all Nodes	8,695	1.60%	0.04%	12.50

classes of equivalence for these URIs. Finally, the remaining variations finished in six jobs and needed 10–13min.

7.1.2 Indexing Execution Time. Here, we compare the execution time of the creation of indexes by selecting different number of machines, and we notice the gain observed by using `PrefixIndex`. As we can see in Figure 17, the whole indexing time needed is approximately 22min, while as the number of machines decreases, the indexing time increases even exponentially in some cases. Furthermore, we can see that by using 64 machines, the creation of each of the `SameAsCat` and `ElementIndex` needs approximately 10min while the creation of `SameAsPrefixIndex` and `PrefixIndex` needs only 3min. Concerning the gain from `PrefixIndex` we ignored 37,252,333 URIs whose prefix occurs only in one dataset, which corresponds to 500MB of data, i.e., there was no need to transfer the aforementioned URIs to the reducer.

7.1.3 Lattice Measurements in Parallel. Here, we compare the execution time of the creation of a lattice by selecting different numbers of machines for (a) the same number of datasets, different thresholds, and a specific number of machines (see Table 9); (b) a specific threshold for a different number of machines; and (c) the execution time for different sizes of datasets and a specific threshold and number of machines. Regarding (a), we can observe in Table 9 measurements concerning

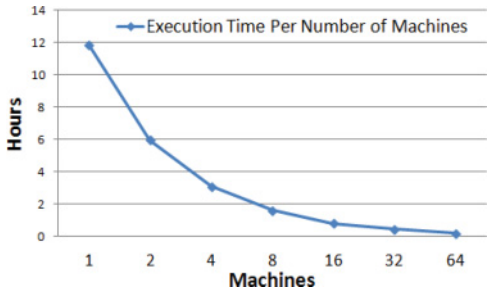

Fig. 18. Lattice-based measurements for different numbers of machines and 35 datasets.

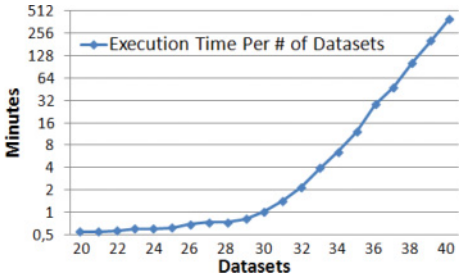

Fig. 19. Lattice-based measurements for different numbers of datasets and 64 machines.

the lattice of 35 nodes by using 64 machines. In this experiment, we change the threshold each time. In particular, we split the power set into smaller "slices," where the size of each "slice" is less than or equal to a specific number of nodes. Since we use 64 machines, the ideal case is for each machine to compute the measurements for 1/64 of all the nodes, i.e., 1.56%. We can observe that when we split the lattice into 595 "slices," i.e., a "slice" for each pair of datasets (number of pairs=$\frac{|\mathcal{D}|*|\mathcal{D}-1|}{2}$), where the number of the nodes of each "slice" is less or equal than 1/4 of all the nodes, the execution time of the measurements was over 3h, and one machine computed the measurements for 25.1% of all the nodes. On the contrary, by choosing the number of each "slice's" nodes to be less or equal than 1/2048 of all the nodes, 2,576 "slices" are created. In that case, the algorithm needed just 12min to perform the measurements, whereas there was no single machine that computed more than 1.64% of all the nodes (which is close to the ideal case). Moreover, by creating more "slices" where the size of each "slice" is smaller (e.g., each "slice" is $\leq$1/4096 or $\leq$1/8192 of all the nodes of the power set), we can go "closer" to the ideal case; however, this can lead to an increase in execution time (but not so much as shown in the last column of Table 9), since the number of key-value pairs, i.e, "slices," are larger, and the communication cost increases. The "slices" are distributed randomly in the machines; therefore, it seems that by choosing to create more "slices," where each "slice" does not contain a large number of nodes, it is not as likely that a single machine will compute a large proportion of nodes compared to the number of nodes that each of the other machines will compute.

In Figure 18, we can observe the scalability that is achieved by using different numbers of machines. More specifically, we can see that each time we double the number of machines, the execution time is reduced in half. One machine needs over 11h to compute the measurements of 2^{35} subsets, while 64 machines need 12min. Concerning the measurements for different numbers of datasets (by using 64 machines), in Figure 19, we can see that for 1 billion or fewer nodes (e.g., 30 or fewer datasets), less than 1min is needed to compute the lattice of all the nodes. As the number of datasets increases, the execution time is increased exponentially. However, this algorithm was able to compute the measurements for over 1 trillion nodes, i.e., a lattice of 40 datasets, in approximately 6h.

7.1.4 Executive Summary. We saw that with the proposed parallelization methods and algorithms we were able to partition efficiently the data and to achieve a scalability close to the ideal. Regarding the construction of the indexes, in the construction of `SameAsCat` due to the existence of some large connected components that delayed the execution time of some reducers, the difference in the execution time between 32 and 64 machines was only 1.5min (10min in the first case and 8.6 in the second one). Concerning the construction of `ElementIndex`, which is a classical way

to create an inverted index in parallel, we achieved a good scalability (by reducing the number of machines, execution time increases by double in some cases). Moreover, by using the same settings, i.e., creation of `PrefixIndex`, computation of transitive and symmetric closure of `owl:sameAs` relationships, and creation of `ElementIndex`, we needed 22min with the parallelized version, while the single machine needed 7h. Therefore, we can observe a 19× speedup for the whole process by using a cluster of 64 machines. As far as lattice measurements are concerned (whose partition is challenging), by choosing a small value for θ ($\theta = 2^{|\mathcal{D}|}/2048$), i.e., by splitting the lattice into many small "slices," the number of nodes that each machine computed was very close to the ideal case, while as we reduced the number of machines, the execution time increased by double. Furthermore, we were able to compute a lattice of 1 billion lattice nodes in 1min with the parallelized version (by using 64 machines), while with the single machine the execution time for the same number of nodes was 35min, i.e., we observe a 35× speedup. Finally, regarding approaches that we used for optimizing the proposed algorithms, by combining the Hash-to-Min algorithm with `SameAsPrefixIndex` and the *Signature-Based* algorithm, we achieved the lowest execution time, while, by using `PrefixIndex`, we were able to ignore millions of URIs in our computation.

7.2 Measurements over the Datasets

The measurements that are described in this section allow someone not only to get an overview of the connectivity (integration quality) of the LOD cloud but also to quantify the quality and value of the integration of a subset of datasets. For instance, suppose one wants to compare two different semantic warehouses (or two or more versions of a semantic warehouses) for a particular domain, e.g., for the marine domain. For quantifying the value of each warehouse, and thus for judging which of them is better, one prominent aspect is how connected the constituent sources of each warehouse are. In addition, one can exploit metrics, like those described in Reference [32], which also quantify the value of each underlying source. For instance, with these metrics, one can find how many unique triples each source contributes, to what extent the average degree (i.e., the average number of triples for the URIs of a source in the warehouse) of a source's URIs are increased, and so on. Moreover, such measurements are important for deciding which datasets are worth considering for building a warehouse for a domain. For example, consider a user/scientist who wants to create a warehouse for a domain, but has no prior knowledge of the existing datasets, or who knows some datasets but would like to find more datasets that have commonalities with the datasets that he or she already knows. Since, it is not worthwhile to spend resources for integrating sources that do not have commonalities, the measurements that we introduce can give an overview of the value of the integration of a subset of sources and therefore can help scientists (or users) to decide which datasets to use.

Statistics derived by the Indexes. Table 10 synopsizes some interesting statistics for the datasets and the creation of the element index. First, according to the prefix index, each dataset's URIs contains on average 655 different prefixes. Moreover, 67.6% of prefixes exist only in one dataset. However, this percentage corresponds only to the 9.8% of distinct URIs, while 33.4% of the prefixes correspond to 90.2% of the URIs. The element index contains 27 million real-world objects (*rwo*). As one can see, there are 6.8 million *rwo* (2% of all the *rwo*), which are part of three or more datasets. This percentage corresponds to 30 million URIs (8% of URIs). The number of unique B having $directCount(B) > 0$ are 11,892, i.e., 0.04% of the ei size.

We used the aforementioned *directCount*s for computing the $co_{\sim}(B)$ of over 3 billion lattice nodes (all pairs, triads, quads, quintets, and each subset B, where $|B| \geq 6$ and $co_{\sim}(B) \geq 30$) and the $co_{\sim}(B)$ of over 34 billion nodes (having $co_{\sim}(B) \geq 20$) by using the bottom-up parallel traversal algorithm. Finally, we excluded from this computation URIs belonging to *rdf*, *rdfs*, *owl*, and popular ontologies such as *foaf*.

Table 10. Index Creation Statistics

Category	Value
Prefix Index Size	197,826
SameAs Prefix Index Size	3,895
Unique Real World Objects	339,456,595
Element Index Size (*rwo*)	26,232,857
Element Index Size (URIs)	76,111,907
rwo in 3 or more D_i	6,806,819
URIs corresponding to *rwo* in 3 or more D_i	30,336,586
Num. of Lattice Nodes (threshold $\geq$ 30)	3,178,929,282
Num. of Lattice Nodes (threshold $\geq$ 20)	34,166,090,688

Table 11. SameAs Catalog Statistics

Category	Value
SameAs Triples	44,028,829
SameAs Catalog Size	65,315,931
SameAs Triples Inferred	97,457,139
Pairs sharing at least 1 real-world object	7,119
New Pairs discovered due to SameAs Alg.	2,725
Triads sharing at least 1 real-world object	86,540
New Triads discovered due to SameAs Alg.	37,445
SameAs Unique IDs	24,076,816

Gain from transitive and symmetric closure computation. Regarding the `owl:sameAs` catalog, Table 11 shows some statistics derived by the computation of transitive and symmetric closure. The computation of closure had as a result 2,725 new pairs (38.2% of the number of all connected pairs) and 37,445 new triads (43.2%) that have at least one common real-world object (without taking into account URIs belonging in popular ontologies). Moreover, the algorithm found more than 97 million new `owl:sameAs` relationships. Indeed, the increased percentage of `owl:sameAs` triples was 221%. The unique URIs of the `owl:sameAs` catalog are over 65 million, while the different *rwo* are over 24 million. This means that on average there are approximately 3 URIs for a specific real-world object.

Figure 20 shows how many pairs exist for a value of the *rwo* threshold (e.g., threshold 10 means that two datasets shares at least 10 *rwo*) and the analogous measurement for the triads. It is worth noting that over 1,000 pairs of datasets share at least 100 common *rwo*, while 61 pairs of datasets share more than 100,000 common *rwo* (14 of them also share more than 1,000,000 common *rwo*). Concerning the triads, it is worth mentioning that 123 triads share more than 100,000 *rwo*, while 6 of them share over 1 million *rwo*.

Power-Law Distribution of Real-World Objects. In Figure 22, we can observe that there exists a power-law distribution concerning the size of the connected components that occur with each *rwo*. In particular, there exist approximately 17 million connected components having size 2, i.e., there exist exactly two URIs for the same *rwo*. Only a few *rwo* belong to a connected component having size greater than or equal to 10 (fewer than 1 million *rwo*), while for less than 3,000 of *rwo* their connected component size is greater than or equal to 50 (i.e., there exists 50 or more URIs for a unique *rwo*). Concerning the exact number of datasets where a unique *rwo* occurs, there

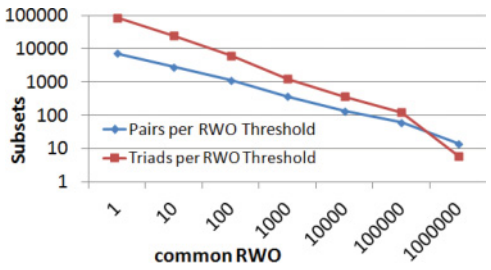

Fig. 20. Number of pairs and triads per threshold.

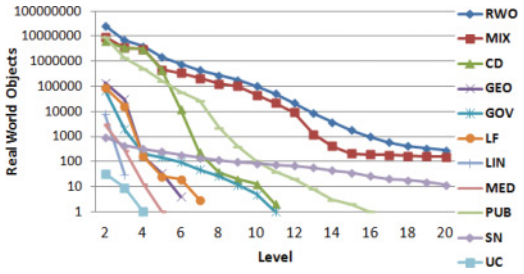

Fig. 21. Unique(RWO)–Max subset per level.

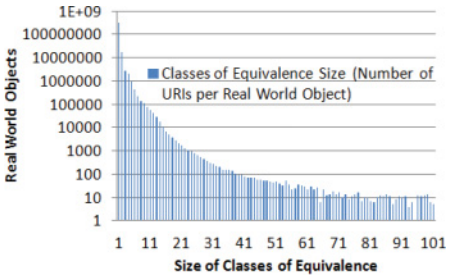

Fig. 22. Number of connected components per real-world object.

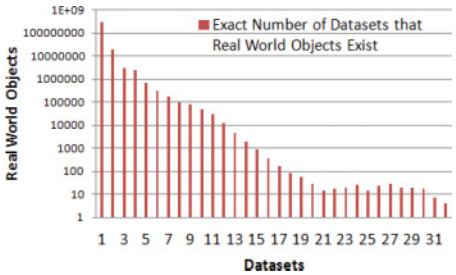

Fig. 23. Number of datasets where real-world objects exactly occur.

Table 12. Top 10 Subsets ≥3 with the Most Common *rwo*

Datasets of subset B	$co_{\sim}(B)$
1: {DBpedia,Freebase,Wikidata}	3,528,389
2: {DBpedia,Yago,Wikidata}	3,515,102
3: {DBpedia,Yago,Freebase}	3,040,046
4: {Yago,Freebase,Wikidata}	2,990,648
5: {DBpedia,Yago,Freebase,Wikidata}	2,990,176
6: {DNB,id.loc.gov,VIAF}	1,344,217
7: {bl.uk,id.loc.gov,VIAF}	1,039,372
8: {BNF,id.loc.gov,VIAF}	995,839
9: {Wikidata,id.loc.gov,VIAF}	605,043
10: {DNB,BNF,VIAF}	588,891

exists again a power-law distribution, i.e., see Figure 23. Indeed, most *rwo* occur only in 1 dataset, while 19 million *rwo* occur in exactly 2 datasets. Moreover, it is worth mentioning that millions of URIs exist either in three or four datasets, whereas over 100,000 *rwo* occur in 10 or more datasets. However, only a few *rwo* (fewer than 2,000) occur in 15 or more datasets.

Common real-world objects among three or more datasets. Table 12 shows the 10 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

Table 13. Top 10 Datasets with the Most *rwo* Existing in at Least 3 Datasets

Dataset D_i	***rwo* in** ≥ 3 D_i	**(% of** D_i ***rwo*)**
Wikidata	4,447,325	15.00%
DBpedia	4,169,588	19.60%
Freebase	3,646,926	3.00%
Yago	3,616,916	17.00%
VIAF	3,052,331	9.80%
id.loc.gov	2,679,436	0.40%
d-nb	1,727,677	26.30%
bnf	1,149,920	3.00%
bl.uk	1,051,576	5.00%
GeoNames	534,139	1.70%

datasets. In particular, the subset comprising the datasets *DBpedia*,[18] *Freebase*,[19] and *Wikidata*[20] shares over 3.5 million *rwo*, while the quad that contains also *Yago*[21] (apart from these datasets) contains approximately 3 million common *rwo*. Afterwards, triads of datasets that belong in the publication domain follow, such as *VIAF*,[22] *DNB*,[23] *bl.uk*,[24] *id.loc.gov*,[25] and *BNF*[26]. Finally, *Wikidata* shares a lot of common *rwo* with datasets of the publication domain.

Figure 21 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 levels 3 to 5 is the cross domain, whereas from levels 6 to 10 the publications domain is the most connected one. In the remaining levels (from 11 to 20), the domain with the most common *rwo* is the social networking domain. Moreover, regarding combinations with datasets from different domains, there are 13 datasets that share thousands of *rwo* and 20 datasets sharing over 100 *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, *Wikidata* is the biggest dataset, as we can observe in Table 13 (in ascending order with respect to the number of *rwo* in three or more datasets). It is logical that *Wikidata* is first in this category, since it has a lot of connections with both cross-domain and publications domain datasets, while the three other popular cross-domain datasets follow. The other datasets are predominantly from the publication domain, while the only dataset outside the cross-domain and publications domain is the *GeoNames*[27] dataset from the geographical domain. Additionally, 126 of 302 datasets (41.7%) contain at least hundreds of *rwo* that can be found in three or more datasets.

[18] http://dbpedia.org/.
[19] http://developers.google.com/freebase/.
[20] http://wikidata.org.
[21] http://yago-knowledge.org.
[22] http://viaf.org/.
[23] http://www.dnb.de/.
[24] http://www.bl.uk/.
[25] http://id.loc.gov/.
[26] http://www.bnf.fr.
[27] http://www.geonames.org/.

Table 14. Number of Connections between the Datasets

Category	Number of Datasets	(% of Datasets)
|Datasets| being isolated	19	6.20%
|Datasets| having 1 to 5 connections	26	8.60%
|Datasets| having 6 to 10 connections	16	5.20%
|Datasets| having 11 to 50 connections	130	43.00%
|Datasets| having 51 to 100 connections	73	24.1%
|Datasets| having 101 to 150 connections	28	9.2%
|Datasets| having 151 to 200 connections	9	2.9%
|Datasets| having over 200 connections	1	0.3%

Conclusions about the Connectivity among the Datasets. Here, we describe the conclusions that can be derived concerning the connectivity of LOD datasets. Initially, as we have seen, only 2% of real-world objects are part of three or more datasets and 7.7% of real-world objects are part of two or more datasets. Moreover, Table 14 shows how many connections the RDF datasets have, i.e., with how many datasets they share real-world objects (we have again excluded from the measurements URIs referring to schema elements that belong in popular ontologies). In particular, there exist 19 datasets that are "isolated," i.e., they have no common real-world objects with other ones and 26 datasets having five or fewer connections. Most of these datasets belong to the publications and government domains, and either they cover aspects that cannot be found in other datasets or their publishers have not created mappings to other datasets. A large percentage of datasets, i.e., 130 datasets, have connections with more (11–50) datasets, while a small percentage of datasets (belonging predominantly to the cross domain and publications domain), i.e., 10 datasets, share real-world objects with over 150 other datasets.

A main conclusion is that although most datasets have connections with others, they have in common only a small percentage of real-world objects. Therefore, publishers tend to create mappings for a small subset of their dataset's real-world entities, and mainly they create such mappings with the most popular cross-domain datasets, i.e., *DBpedia*, *Yago*, *Wikidata*, and *Freebase*. Except for datasets belonging in the cross domain, many datasets from the social networks domain are highly connected, i.e., there exist 20 datasets sharing common real-world objects, while there are also datasets from the publication domain sharing many entities. The aforementioned measurements reveal the "sparsity" of the current LOD cloud and make evident the need for better connectivity so the contents of each dataset can be easily found and reusable. A possible increase in the dimension of datasets' connectivity can offer more "relevant" information for the entities and can reveal the different values that a specific fact may have in different datasets (e.g., birth year of a person). As stated in Reference [11], for increasing the correctness of data (i.e., veracity), the detection of instances (from different sources) referring to the same entity is a requirement for detecting such conflicts and errors.

8 OTHER APPLICATIONS OF THE PROPOSED INDEXES AND MEASUREMENTS

In this section, we show how the proposed approach (a) can be implemented over different frameworks and (b) can be exploited for computing other kinds of commonalities.

8.1 Execution Time by Using Different Big Data Frameworks

In this section, we report the execution times for creating the indexes using the same parallel algorithms over the *Spark* [48], and we compare the results with the corresponding execution

Table 15. Speedup Achieved by Using Spark (Comparing to MapReduce) for Different Number of Machines

Index	Speedup (4 Machines)	Speedup (8 Machines)	Speedup (16 Machines)	Speedup (32 Machines)
Prefix Indexes	3.91×	4.21×	3.62×	2.50×
SameAs Catalog	1.54×	1.82×	2.00×	1.84×
Element Index	2.30×	2.18×	1.38×	1.45×
All Indexes	2.20×	2.25×	1.78×	1.69×

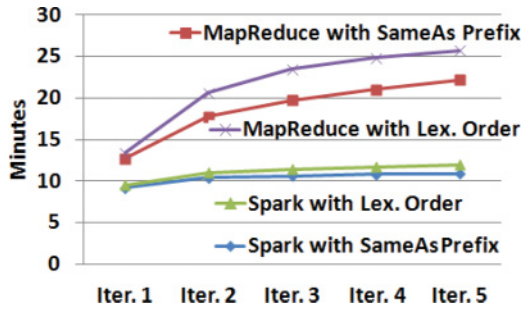

Fig. 24. Total execution time after the execution of each iteration of the Hash-To-Min algorithm for 16 machines by using Spark and MapReduce.

times achieved over *MapReduce* [8]. For the experiments of this section, we used cloudera-cdh-5.4.4 software and the following releases: *Apache Hadoop 2.5.2*[28] and *Apache Spark 1.3.0*.[29] In the experiments, we used 32 different virtual machines in the okeanos cloud computing service,[30] each having 1 core and 1GB of memory. For performing the experiments using *Spark*, we employed the same *Map Reduce* methods that were described in Section 6.

In Figure 25, we can see the execution time for constructing the `SameAsCat` by using Spark and MapReduce. In particular, we can observe that Spark is faster than MapReduce in all the experiments. The achieved speedup, as we can observe in Table 15, is from 1.54 to 2 times. The Hash-to-Min algorithm needs multiple iterations to be completed by using either Spark or MapReduce. However, Spark performs in-memory processing of data that is much faster compared to MapReduce. In particular, MapReduce reads data from the disk, and, after a particular iteration, it writes the results to the HDFS, and then it reads the data again from the HDFS for the next the iteration. This is a major reason why Spark is much faster than MapReduce, especially when multiple iterations should be performed. Such an experiment can be seen in Figure 24, where we ran the algorithm for 16 machines for both Spark and MapReduce. As we can see, Spark is much faster, especially in the second to fifth iterations. In Figure 26, we observe that MapReduce needs much more time compared to Spark for constructing the prefix indexes (i.e., `PrefixIndex` and `SameAsPrefixIndex`). In particular, it is always 2.5 to 4.21 times slower than Spark (see Table 15), while the gain of using more machines is also clearly observed for Spark. It is worth noting that the indexing time for Spark for the aforementioned indexes is just 1min and 50s by using 32 machines. Regarding the `ElementIndex` construction, the Spark framework is 1.45 to 2.3 times faster (see Figure 27). Concerning the execution time of the whole process, i.e., see Figure 28, by using

[28] https://hadoop.apache.org/docs/r2.5.2/.
[29] https://spark.apache.org/docs/1.3.0/.
[30] http://okeanos.grnet.gr.

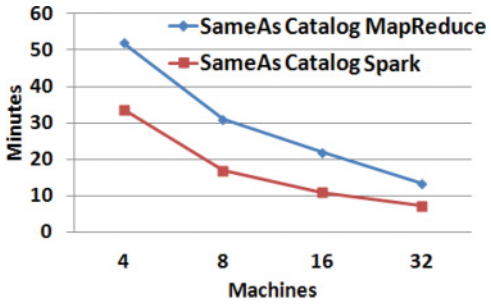

Fig. 25. Comparison of SameAs catalog construction time by using Spark and MapReduce.

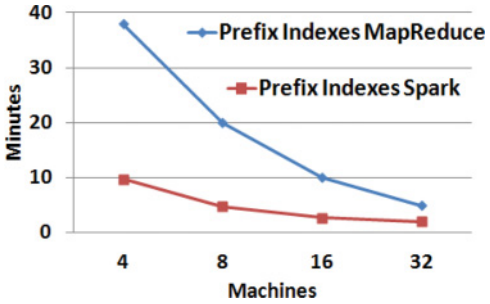

Fig. 26. Comparison of Prefix Indexes construction time by using Spark and MapReduce

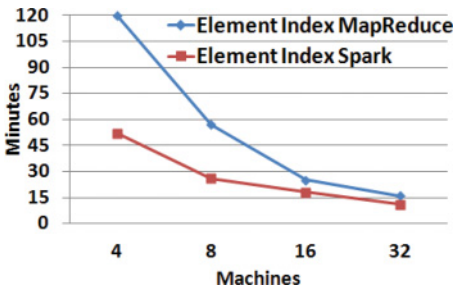

Fig. 27. Comparison of Element Index construction time by using Spark and MapReduce.

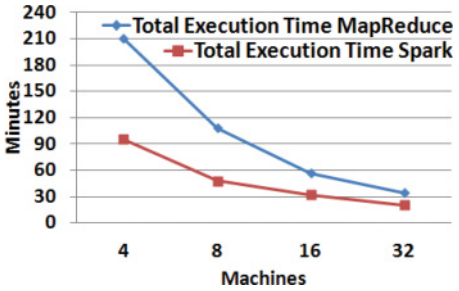

Fig. 28. Comparison of the total execution time for all indexes by using Spark and MapReduce.

32 machines, with Spark we were able to create all the indexes in 20min while MapReduce needed over 34min. Finally, Spark was 1.69 to 2.25 times faster in all four cases (see Table 15).

8.2 Supporting Other RDF Features

The indexes and algorithms presented in this article can be easily adjusted for other RDF features, and here we describe how we can compute the *common literals*. Let L be the set of literals of all the available datasets, where $L = Lit_1 \cup \ldots \cup Lit_n$ and Lit_i is the set of literals of a dataset D_i after having transformed them in lowercase. Suppose that we want to find the *common literals* 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 $cl(B)$, defined as $cl(B) = \cap_{D_i \in B} Lit_i$. Let $dsets(l) = \{D_i \in \mathcal{D} \mid l \in Lit_i\}$, i.e., all the datasets containing a particular literal l. For finding the common literals, we create a simple inverted index, `LiteralsIndex`, which is essentially a function $li : L \rightarrow \mathcal{P}(\mathcal{D})$, where $li(l) = dsets(l)$. Afterwards, we can use the following variation of the *directCount* formula: $directCount(B) = |\{ l \in Left(li) \mid li(l) = B \}|$ to find the *directCount* of each subset B. The final step is to run any lattice algorithm, i.e., either the bottom-up or the top-down, for computing the number of common literals among any subset of datasets.

We created such an inverted index for the literals in parallel by using MapReduce and 16 machines in 40min. Table 16 shows some statistics about the common literals index, where we can observe that over 21 million literals (i.e., 2.5% of literals) can be found in two or more datasets and over 7 million literals in three or more datasets (i.e., 0.8% of literals). Concerning the connections among the datasets, 25,597 pairs of datasets share at least one literal (i.e., 56.3% of all the

Table 16. Literals Index Creation Statistics

Category	Value
Unique Literals	867,922,173
Literals Index Size	21,732,701
Literals in three or more D_i	7,535,819
Pairs of Datasets sharing literals	25,597
Triads of Datasets sharing literals	950,414
Direct Counts	133,450

Table 17. Top 10 Subsets ≥3 with the Most Common Literals

Datasets of subset B	$cl(B)$
1: {Yago,Freebase,Wikidata}	3,644,482
2: {DBpedia,Freebase,Wikidata}	3,030,208
3: {DBpedia,Yago,Freebase}	1,492,503
4: {DBpedia,Yago,Wikidata}	1,469,452
5: {DBpedia,Yago,Freebase,Wikidata}	1,262,739
6: {DBpedia,Wikidata,VIAF}	649,371
7: {Freebase,Wikidata,VIAF}	645,966
8: {Freebase,DBpedia,VIAF}	643,855
9: {Freebase,DBpedia,VIAF,Wikidata}	545,735
10: {Freebase,Yago,VIAF}	438,330

possible pairs), while 950,414 triads of datasets share at least one literal (i.e., 20.9% of all possible triads). Therefore, compared to URIs, there exist more pairs and triads of datasets that are connected through common literals. Moreover, in Table 17, we can see the top 10 subsets containing ≥3 datasets that have the most common literals. In particular, the four popular datasets from the cross domain are again highly connected, i.e., they share millions of literals, whereas the aforementioned datasets share a large number of literals with the VIAF Dataset (which belongs to the publications domain). Finally, it is worth noting that only four datasets, i.e., 1.3% of datasets, do not have common literals with other datasets.

9 CONCLUDING REMARKS

Existing approaches for measurements over the LOD report measurements between pairs of datasets, focus on document-based features (not dataset based), or do not focus on indexes for speeding up such measurements. In this article, we showed real motivating scenarios for a number of tasks where measurements and indexes involving two or more datasets are important. Since it would be prohibitively expensive to compute measurements that involve more than two datasets without special indexes, we introduced various indexes (and their construction algorithms) that can speed up 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 45s to compute the reflexive and transitive closure for 13 million `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. Afterwards, we introduced scalable methods and techniques for achieving better speedup. In particular, we introduced parallel versions for the construction of the indexes and for measuring the lattice of measurements. With the parallel versions of the algorithms (and by using MapReduce), we were able to construct all the indexes in 22min (while for the single-node process we needed approximately 7h) and to compute the lattice of billions of nodes in less than 1min and the lattice of trillions of nodes in approximately 6h. Moreover, 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 655 different prefixes and that 33.4% of the prefixes are used in 9.8% of URIs and occur in only one dataset. Concerning `owl:sameAs` relationships, a real-world object exists on, on average, three URIs. The transitive and symmetric closure of the `owl:sameAs` relationships of all datasets yielded more than 97 million new `owl:sameAs` relationships, and this increases the connectivity of the datasets; 38.2% of the 7,119 connected pairs of datasets are due to these new relationships. The measurements also reveal the "sparsity" of the current LOD cloud and make evident the need for better connectivity. Only 2% of real-world objects (in 6,806,819 real-world objects) are part of 3 or more datasets. Most of these real-world objects (belonging in $\geq$3 datasets) are part of cross-domain datasets such as *Wikidata* and *DBpedia*, publications datasets such as *VIAF*, and geographical datasets like *GeoNames*. Additionally, many datasets from the social networking domain are highly connected. Finally, we compared the execution times of the algorithms by using two big data frameworks, i.e., MapReduce and Spark, while we showed how the proposed methods can be adjusted for computing common literals. By constructing the index for literals, we saw that only a small percentage of literals (i.e., 2.5%) can be found in two or more datasets.

There are several topics that are worth further research. One direction is to generalize and cover more RDF features, and this requires extending the semantics-aware indexes so that they describe the entire contents of datasets (i.e., also properties and triples). Another (complementary) direction is to extend the lattice-based measurements to enable the efficient computation of features for any Boolean combination (i.e., a set-theoretic operation, like $(D_1 \cup D_2) \cap D_3$) of datasets).

REFERENCES

[1] Maribel Acosta, Amrapali Zaveri, Elena Simperl, Dimitris Kontokostas, Sören Auer, and Jens Lehmann. 2013. Crowdsourcing linked data quality assessment. In *International Semantic Web Conference*. Springer, 260–276.

[2] Charu Aggarwal, Yan Xie, and Philip S. Yu. 2009. Gconnect: A connectivity index for massive disk-resident graphs. *Proc. VLDB Endow.* 2, 1 (2009), 862–873.

[3] Sören Auer, Jan Demter, Michael Martin, and Jens Lehmann. 2012. LODStats–An extensible framework for high-performance dataset analytics. In *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 353–362.

[4] Nikos Bikakis and Timos K. Sellis. 2016. Exploration and visualization in the web of big linked data: A survey of the state of the art. In *Proceedings of the International Conference on Extending Database Technology/International Conference on Display Technology Workshops (EDBT/ICDT'16)*, Vol. 1558.

[5] Christian Bizer, Peter Boncz, Michael L Brodie, and Orri Erling. 2012. The meaningful use of big data: Four perspectives–four challenges. *ACM SIGMOD Rec.* 40, 4 (2012), 56–60.

[6] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. 2012. LINDA: Distributed web-of-data-scale entity matching. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. ACM, 2104–2108.

[7] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. Entity resolution in the web of data. *Synth. Lect. Semant. Web* 5, 3 (2015), 1–122.

[8] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[9] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R. Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. 2004. Swoogle: A search and metadata engine for the semantic web. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*. ACM, 652–659.
[10] Xin Luna Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Kevin Murphy, Shaohua Sun, and Wei Zhang. 2014. From data fusion to knowledge fusion. *Proc. VLDB Endow.* 7, 10 (2014), 881–892.
[11] Xin Luna Dong and Felix Naumann. 2009. Data fusion: Resolving data conflicts for integration. *Proc. VLDB Endow.* 2, 2 (2009), 1654–1655.
[12] Ahmed El-Roby and Ashraf Aboulnaga. 2015. ALEX: Automatic link exploration in linked data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1839–1853.
[13] Mohamed Ben Ellefi, Zohra Bellahsene, Stefan Dietze, and Konstantin Todorov. 2016. Dataset recommendation for data linking: An intensional approach. In *International Semantic Web Conference*. Springer, 36–51.
[14] José M. Giménez-García, Harsh Thakkar, and Antoine Zimmermann. 2016. Assessing trust with pagerank in the web of data. In *International Semantic Web Conference*. Springer, 293–307.
[15] Hugh Glaser, Afraz Jaffri, and Ian Millard. 2009. Managing co-reference on the semantic web. In *Proceedings of the WWW2009 Workshop: Linked Data on the Web (LDOW'09)*.
[16] Christophe Guéret, Paul Groth, Claus Stadler, and Jens Lehmann. 2012. Assessing linked data mappings using network measures. In *The Semantic Web: Research and Applications*. Springer, 87–102.
[17] Harry Halpin and Patrick J. Hayes. 2010. When owl: Sameas isn't the same: An Analysis of identity links on the semantic web. In *Proceedings of the Conference on Linked Data on the Web (LDOW'10)*.
[18] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. 2007. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web: Proceedings of the 6th International Semantic Web Conference (ISWC'07)*. 211–224.
[19] James Hendler. 2014. Data integration for heterogenous datasets. *Big Data* 2, 4 (2014), 205–215.
[20] Aidan Hogan, Andreas Harth, and Stefan Decker. 2007. Performing object consolidation on the semantic web data graph. In *Proceedings of the WWW2007 Workshop* I^3.
[21] Aidan Hogan, Andreas Harth, Jürgen Umbrich, Sheila Kinsella, Axel Polleres, and Stefan Decker. 2011. Searching and browsing linked data with swse: The semantic web search engine. *Web Semant.* 9, 4 (2011), 365–401.
[22] Filip Ilievski, Wouter Beek, Marieke van Erp, Laurens Rietveld, and Stefan Schlobach. 2016. LOTUS: Adaptive text search for big linked data. In *International Semantic Web Conference*. Springer, 470–485.
[23] Antoine Isaac and Bernhard Haslhofer. 2013. Europeana linked open data–data. europeana. eu. *Semant. Web* 4, 3 (2013), 291–297.
[24] Thomas Jech. 2013. *Set Theory*. Springer Science & Business Media.
[25] Tobias Käfer, Ahmed Abdelrahman, Jürgen Umbrich, Patrick O'Byrne, and Aidan Hogan. 2013. Observing linked data dynamics. In *Extended Semantic Web Conference*. Springer, 213–227.
[26] Tobias Käfer, Jürgen Umbrich, Aidan Hogan, and Axel Polleres. 2012. Towards a dynamic linked data observatory. *Proceedings of the Linked Data on the Web at WWW (LDOW'12)*.
[27] Jan-Christoph Kalo, Silviu Homoceanu, Jewgeni Rose, and Wolf-Tilo Balke. 2015. Avoiding chinese whispers: Controlling end-to-end join quality in linked open data stores. In *Proceedings of the ACM Web Science Conference*. ACM.
[28] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2011. PEGASUS: Mining peta-scale graphs. *Knowl. Inf. Syst.* 27, 2 (2011), 303–325.
[29] Michael Hausenblas Keith Alexander, Richard Cyganiak and Jun Zhao. 2011. Describing Linked Datasets with the VoID Vocabulary, W3C Interest Group Note. Retrieved from http://www.w3.org/TR/2011/NOTE-void-20110303/.
[30] Graham Klyne and Jeremy J. Carroll. 2004. Resource description framework (RDF): Concepts and abstract syntax, W3C Recommendation. Retrieved from http://www.w3.org/TR/rdf-concepts/.
[31] M. Mountantonakis, C. Allocca, P. Fafalios, N. Minadakis, Y. Marketakis, C. Lantzaki, and Y. Tzitzikas. 2014. Extending VoID for expressing the connectivity metrics of a semantic warehouse. In *Proceedings of the 1st International Workshop on Dataset Profiling & Federated Search for Linked Data (PROFILES'14)*.
[32] M. Mountantonakis, N. Minadakis, Y. Marketakis, P. Fafalios, and Y. Tzitzikas. 2016. Quantifying the connectivity of a semantic warehouse and understanding its evolution over time. *Int. J. Semant. Web Inf. Syst.* 12, 3 (2016), 27–78.
[33] Michalis Mountantonakis and Yannis Tzitzikas. 2016. On measuring the lattice of commonalities among several linked datasets. *Proc. VLDB Endow.* 9, 12 (2016), 1101–1112.
[34] Michalis Mountantonakis and Yannis Tzitzikas. 2017. How linked data can aid machine learning-based tasks. In *International Conference on Theory and Practice of Digital Libraries*. Springer, 155–168.
[35] Markus Nentwig, Tommaso Soru, Axel-Cyrille Ngonga Ngomo, and Erhard Rahm. 2014. LinkLion: A link repository for the web of data. In *The Semantic Web: ESWC 2014 Satellite Events*. Springer, 439–443.
[36] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113.

[37] Damla Oguz, Belgin Ergenc, Shaoyi Yin, Oguz Dikenelli, and Abdelkader Hameurlain. 2015. Federated query processing on linked data: A qualitative survey and open challenges. *Knowl. Eng. Rev.* 30, 5 (2015), 545–563.
[38] Laura Papaleo, Nathalie Pernelle, Fatiha Saïs, and Cyril Dumont. 2014. Logical detection of invalid sameas statements in RDF data. In *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 373–384.
[39] Norman W. Paton, Klitos Christodoulou, Alvaro A. A. Fernandes, Bijan Parsia, and Cornelia Hedeler. 2012. Pay-as-you-go data integration for linked data: Opportunities, challenges and architectures. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*. ACM.
[40] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *VLDB J.* 25, 2 (2016), 243–268.
[41] Eric Prud' Hommeaux and Andy Seaborne. 2008. SPARQL query language for RDF. *W3C Recommend.* Retrieved from http://www.w3.org/TR/rdf-sparql-query/.
[42] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *Proceedings of the International Conference on Data Engineering (ICDE'13)*. IEEE, 50–61.
[43] Laurens Rietveld, Wouter Beek, and Stefan Schlobach. 2015. LOD lab: Experiments at LOD scale. In *International Semantic Web Conference*. Springer, 339–355.
[44] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the linked data best practices in different topical domains. In *Proceedings of the International Semantic Web Conference (ISWC'14)*. Springer, 245–260.
[45] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *Conference Record of the 1971 12th Annual Symposium on Switching and Automata Theory*. IEEE, 114–121.
[46] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. 2008. On graph features of semantic web schemas. *IEEE Trans. Knowl. Data Eng.* 20, 5 (2008), 692–702.
[47] Giovanni Tummarello, Eyal Oren, and Renaud Delbru. 2007. Sindice.com: Weaving the open linked data. In *Proceedings of the International Semantic Web Conference (ISWC'07)*, Vol. 4825. 547–560.
[48] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets.*HotCloud* 10, 10-10 (2010), 95.
[49] Amrapali Zaveri, Dimitris Kontokostas, Mohamed A Sherif, Lorenz Bühmann, Mohamed Morsey, Sören Auer, and Jens Lehmann. 2013. User-driven quality evaluation of dbpedia. In *Proceedings of the 9th International Conference on Semantic Systems*. ACM, 97–104.
[50] Amrapali Zaveri, Anisa Rula, Andrea Maurino, Ricardo Pietrobon, Jens Lehmann, and Sören Auer. 2016. Quality assessment for linked data: A survey. *Semant. Web J.* 7, 1 (2016), 63–93.
[51] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006), 6.

Received March 2017; revised November 2017; accepted November 2017