Apples and Oranges: A Comparison of RDF Benchmarks and Real RDF Datasets

Songyun Duan Anastasios Kementsietsidis Kavitha Srinivas Octavian Udrea
IBM Research - Thomas J. Watson Research Ctr.
19 Skyline Dr
Hawthorne, New York
{sduan, akement, ksrinivs, oudrea}@us.ibm.com

ABSTRACT

The widespread adoption of the Resource Description Framework (RDF) for the representation of both open web and enterprise data is the driving force behind the increasing research interest in RDF data management. As RDF data management systems proliferate, so are benchmarks to test the scalability and performance of these systems under data and workloads with various characteristics.

In this paper, we compare data generated with existing RDF benchmarks and data found in widely used real RDF datasets. The results of our comparison illustrate that existing benchmark data have little in common with real data. Therefore any conclusions drawn from existing benchmark tests might not actually translate to expected behaviours in real settings. In terms of the comparison itself, we show that simple primitive data metrics are inadequate to flesh out the fundamental differences between real and benchmark data. We make two contributions in this paper: (1) To address the limitations of the primitive metrics, we introduce intuitive and novel metrics that can indeed highlight the key differences between distinct datasets; (2) To address the limitations of existing benchmarks, we introduce a new benchmark generator with the following novel characteristics: (a) the generator can use any (real or synthetic) dataset and convert it into a benchmark dataset; (b) the generator can generate data that mimic the characteristics of real datasets with user-specified data properties. On the technical side, we formulate the benchmark generation problem as an integer programming problem whose solution provides us with the desired benchmark datasets. To our knowledge, this is the first methodological study of RDF benchmarks, as well as the first attempt on generating RDF benchmarks in a principled way.

Categories and Subject Descriptors

H.2 [**Systems**]; K.6.2 [**Management Of Computing And Information Systems**]: Installation Management—*Benchmarks*

General Terms

Measurement

SIGMOD'11, June 12–16, 2011, Athens, Greece.

1. INTRODUCTION

The RDF (Resource Description Framework) [24] is quickly becoming the *de-facto* standard for the representation and exchange of information. This is nowhere more evident than in the recent Linked Open Data (LOD) [7] initiative where data from varying domains like geographic locations, people, companies, books, films, scientific data (genes, proteins, drugs), statistical data, and the like, are interlinked to provide one large data cloud. As of October 2010, this cloud consists of around 200 data sources contributing a total of 25 billion RDF triples. The acceptance of RDF is not limited, however, to open data that are available on the web. Governments (most notably from US [13] and UK [14]) are adopting RDF. Many large companies and organizations are using RDF as the *business* data representation format, either for semantic data integration (*e.g.,* Pfizer [19]), search engine optimization and better product search (*e.g.,* Best Buy [6]), or for representation of data from information extraction (*e.g.,* BBC [5]). Indeed, with Google and Yahoo promoting the use of RDF for search engine optimization, there is clearly incentive for its growth on the web.

One of the main reasons for the widespread acceptance of RDF is its inherent flexibility: A *diverse* set of data, ranging from structured data (*e.g.,* DBLP [18]) to unstructured data (*e.g.,* Wikipedia/DBpedia [8]), can all be represented in RDF. Traditionally, the *structuredness* of a dataset is one of the key considerations while deciding an appropriate data representation format (*e.g.,* relational for structured and XML for semi-structured data). The choice, in turn, largely determines how we organize data (*e.g.,* dependency theory and normal forms for the relational model [2], and XML [4]). It is of central importance when deciding how to index it (*e.g.,* B+-tree indexes for relational and numbering scheme-based indexes for XML [25]). Structuredness also influences how we query the data (*e.g.,* using SQL for the relational and XPath/XQuery for XML). In other words, data structuredness permeates every aspect of data management, and accordingly the performance of data management systems is commonly measured against data with the expected level of structuredness (*e.g.,* the TPC-H [29] benchmark for relational and the XMark [31] benchmark for XML data). The main strength of RDF is precisely that it can be used to represent data across the full spectrum of structuredness, from unstructured to structured. This flexibility of RDF, however, comes at a cost. By blurring the structuredness lines, RDF data management becomes a challenge since no assumptions can be made *a-priori* by an RDF DBMS as to what type(s) of data it is going to manage. Unlike the relational and XML case, an RDF DBMS has the onerous requirement that its performance should be tested against very diverse data sets (in terms of structuredness).

A number of RDF data management systems (a.k.a. RDF stores) are currently available, with Sesame [11], Virtuoso [15],

3Store [17], and Jena [20], being some of the most notable ones. There are also research prototypes, with RDF-3X [22] and SW-Store [1] being the most notable representatives. To test the performance of these RDF stores, a number of RDF benchmarks have also been developed, namely, BSBM [10], LUBM [16], and SP^{2}Bench [26]. For the same purposes of testing RDF stores, the use of certain real datasets has been popularized (*e.g.,* the MIT Barton Library dataset [28]). While the focus of existing benchmarks is mainly on the performance of the RDF stores in terms of scalability (*i.e.,* the number of triples in the tested RDF data), a natural question to ask is *which types of RDF data these RDF stores are actually tested against*. That is, we want to investigate: (a) whether existing performance tests are limited to certain areas of the structuredness spectrum; and (b) what are these tested areas in the spectrum. The results of this investigation constitute one of our key contributions. Specifically, we show that (i) the structuredness of each benchmark dataset is practically *fixed*; (ii) even if a store is tested against the full set of available benchmark data, these tests cover only a small portion of the structuredness spectrum. However, we show that many real RDF datasets lie in currently untested parts of the spectrum.

These two points provide the motivation for our next contribution. To expand benchmarks to cover the structuredness spectrum, we introduce a novel benchmark data generator with the following unique characteristics: Our generator accepts as input any dataset (*e.g.,* a dataset generated from any of the existing benchmarks, or any real data set) along with a desired level of structuredness and size, and uses the input dataset as a *seed* to produce a dataset with the indicated size and structuredness. Our data generator has several advantages over existing ones. The first obvious advantage is that our generator offers complete control over both the structuredness and the size of the generated data. Unlike existing benchmark generators whose data domain and accompanying queries are *fixed* (*e.g.,* LUBM considers a schema which includes Professors, Students and Courses, and the like, along with 14 fixed queries over the generated data), our generator allows users to pick their dataset and queries of choice and methodically create a benchmark out of them. By fixing an input dataset and output size, and by changing the value of structuredness, a user can test the performance of a system across any desired level of structuredness. At the same time, by considering alternative dataset sizes, the user can perform scalability tests similar to the ones performed by the current benchmarks. By offering the ability to perform all the above using a variety of input datasets (and therefore a variety of data and value distributions, as well as query workloads), our benchmark generator can be used for extensive system testing of a system's performance along multiple independent dimensions.

Aside from the practical contributions in the domain of RDF benchmarking, there is a clear technical side to our work. In more detail, the notion of structuredness has been presented up to this point in a rather intuitive manner. In Section 3, we offer a formal definition of structuredness and we show how the structuredness of a particular set can be measured. The generation of datasets with varying sizes and levels of structuredness poses its own challenges. As we show, one of the main challenges is due to the fact that *there is an interaction between data size and structuredness: altering the size of a dataset can affect its structuredness, and correspondingly altering the structuredness of a dataset can affect its size*. So, given an input dataset, a desired size and structuredness for an output dataset, we cannot just randomly add/remove triples in the input dataset until we reach the desired output size. Such an approach provides no guarantees as to the structuredness of the output dataset and is almost guaranteed to result in an output dataset with structuredness which is different from the one desired. Similarly, we cannot just adjust the structuredness of the input dataset until we reach the desired level, since this process again is almost guaranteed to result in a dataset with incorrect size. In Section 4, we show that the solution to our benchmark generation problem comes in the form of two objective functions, one for structuredness and one for size, and in a formulation of our problem as an integer programming problem.

To summarize, our main contributions are as follows:

• To our knowledge, this is the *first* study of the characteristics of real and benchmark RDF datasets. Our study provides a first glimpse on the wide spectrum of RDF data, and can be used as the basis for the design, development and evaluation of RDF data management systems.

• We introduce the formal definition of structuredness and propose its use as one of the main metrics for the characterization of RDF data. Through an extensive set of experiments, we show that more primitive metrics, although useful, are inadequate for differentiating datasets in terms of their structuredness. Using our structuredness metrics, we show that existing benchmarks cover only a small range of the structuredness spectrum, which has little overlap with the spectrum covered by real RDF data.

• We develop a principled, generic technique to generate an RDF benchmark dataset that varies independently along the dimensions of structuredness and size. We show that unlike existing benchmarks, our benchmark generator can output datasets that resemble real datasets not only in terms of structuredness, but also in terms of content. This is feasible since our generator can use any dataset as input (real or synthetic) and generate a benchmark out of it.

The rest of the paper is organized as follows: In the next section, we review the datasets used in our study, and present an extensive list of primitive metrics over them. Then, in Section 3 we introduce our own structuredness metrics, and we compute the structuredness for all our datasets and contrasts our metrics with the primitive ones. In Section 4, we present our own benchmark generator, and the paper concludes in Section 5 with a summary of our results and a discussion of the future directions.

2. DATASETS

In this section, we present briefly each dataset used in our benchmark study. For each dataset, we only considered their RDF representation, without any RDFS inferencing, both for real and benchmark datasets. At the end of the section, we also offer an initial set of primitive metrics that we computed for the selected datasets. Since the RDF data model is closer to the object oriented data model and not the relational data model (by nature), the list of primitive metrics considered is sufficient and comprehensive for our needs. These metrics will illustrate the need for a more systematic set of structuredness metrics.

2.1 Real datasets

• **DBpedia:** DBpedia [8] extracts structured information from Wikipedia. The dataset consists of approximately 150 million triples (22 GB). The entities stored in the triples come from a wide range of data types, including Person, Film, (Music) Album, Place, Organization, etc. In terms of properties, the generic wikilink property is the one with the most instantiations. Given the variety of entities stored in the dataset, a wide range of queries can be evaluated over it, but there is no clear set of *representative* queries.

DBpedia comes with many different type systems. One commonly used type system is a subset of the YAGO [32] ontology, and uses approximately 5000 types from YAGO. We report this dataset as DBpedia-Y. The other is a broader type system that includes

types from the DBpedia ontology, with approximately 150,000 types. We refer to this as DBpedia-B (for Base). Later on, while reporting metrics, some will depend on the type system (*e.g.,* instances per type), whereas others will be type system-independent (*e.g.,* the number of triples in the dataset). For the latter set of metrics, we will just refer to DBpedia (without a type system qualifier).

• **UniProt:** The UniProt dataset [3] contains information about proteins and the representation of the dataset in RDF consists of approximately 1.6 billion triples (280 GB). The dataset consists of instances mainly from type Resource (for a life science resource) and Sequence (for amino acid sequences), as well as instances of type Cluster (for clusters of proteins with similar sequences) and of course type Protein. The most instantiated properties, namely properties created and modified, record the creation and modification dates of resources. The UniProt datasets contained a number of reified RDF statements which can give us an inaccurate picture of the actual statistics on the data. We therefore consider only the non-reified statements in our analysis (as with all our datasets).

• **YAGO:** The YAGO ontology [27, 32] brings together knowledge from both Wikipedia and Wordnet, and currently the dataset consists of 19 million triples (2.4GB). Types like WordNet_Person and WordNet_Village are some of the most instantiated, as are properties like describes. In terms of a query load, Neumann and Weikum [21] provide 8 queries over the YAGO dataset that they use to benchmark the RDF-3X system.

• **Barton Library Dataset:** The Barton library dataset [28] consists of approximately 45 million RDF triples (5.5 GB) that are generated by converting the Machine Readable Catalog (MARC) data of the MIT Libraries Barton catalog to RDF. Some of the most instantiated data types in the dataset are Record and Item, the latter being associated with instances of type Person and with instances of Description. The more *primitive* Title and Date type are the most instantiated, while in terms of properties, the generic value property is the one appearing in most entities. In terms of queries, the work of Abadi et al. [1] considers 7 queries as a representative load for the dataset, all of which involve grouping and aggregation.

• **WordNet:** The RDF representation [30] of the well-known WordNet dataset was also used in our study, which currently consists of 1.9 million triples (300MB). In the dataset, the NounWordSense, NounSynset and Word types are among the ones with the most instances, while the tagCount and word properties are some of the most commonly used.

• **Linked Sensor Dataset:** The Linked Sensor dataset [23] contains expressive descriptions of approximately 20,000 weather stations in the United States. The dataset is divided up into multiple subsets, which reflect weather data for specific hurricanes or blizzards from the past. We report our analyses on the dataset about hurricane Ike, under the assumption that the other subsets of data will contain the same characteristics. The Ike hurricane dataset contains approximately 500 million triples (101 GB), but only about 16 types. Most of the instances are from the MeasureData type, which is natural since most instances provided various weather measurements.

2.2 Benchmark datasets

• **TPC-H:** The TPC Benchmark H (TPC-H) [29] is a well-known decision support benchmark in relational databases. We use the TPC-H benchmark in this study as our *baseline*. In more detail, notice that the structuredness of the TPC-H dataset should be almost *perfect*, since these are relational data that are converted to RDF. Therefore, the are two obvious reasons to use TPC-H as our baseline: First, the dataset can be used to check the correctness of our introduced structuredness metrics since any metrics that we devise must indeed confirm the high structuredness of TPC-H. Second, the dataset can be used as a baseline to which we can compare all the other datasets in the study. If a dataset has close, or similar, structuredness to TPC-H, then we expect that this dataset can also be classified as being a relational dataset with an RDF representation.

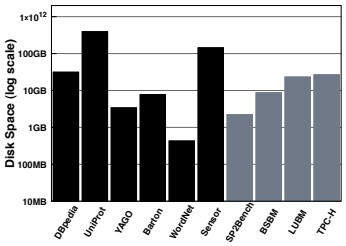

Figure 1: Disk space

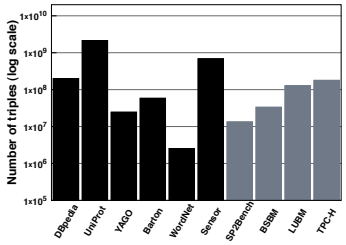

Figure 2: Number of triples

To represent TPC-H in RDF, we use the DBGEN TPC-H generator to generate a TPC-H relational dataset of approximately 10 million tuples with 6 million LINEITEM, 1.5 million ORDER, 800,000 PARTSUP, 200,000 PART and 150,000 CUSTOMER. Then, we use the widely used D2R tool [12] to convert the relational dataset to the equivalent RDF one. This process results in an RDF dataset with 130 million triples (19 GB).

• **BSBM Dataset:** The Berlin SPARQL Benchmark (BSBM) [10] considers an e-commerce domain where types Product, Offer and Vendor are used to model the relationships between products and the vendors offering them, while types Person and Review are used to model the relationship between users and product reviews these users write. We use the BSBM data generator and create a dataset with 25M triples (6.1 GB). In BSBM, type Offer is the most instantiated one, as is the case for the properties of this type. The benchmark also comes with 12 queries and 2 query mixes (sequences of the 12 queries) for the purposes of testing RDF store performance.

• **LUBM Dataset:** The Lehigh University Benchmark (LUBM) [16] considers a University domain, with types like UndergraduateStudent, Publication, GraduateCourse, AssistantProfessor, to name a few. Using the LUBM generator, we create a dataset of 100 million triples (17 GB). In the LUBM dataset, Publication is the most instantiated type, while properties like name and takesCourse are some of the most instantiated. The LUBM benchmark also provides 14 test queries for the purposes of testing the performance of RDF stores.

• **SP^{2}Bench Dataset:** The SP^{2}Bench benchmark [26] uses the DBLP as a domain for the dataset. Therefore, the types encountered include Person, Inproceedings, Article and the like. Using the SP^{2}Bench generator we create a dataset with approximately 10 million triples (1.6 GB). The Person type is the most instantiated in the dataset, as is the case for the name and homepage properties. The SP^{2}Bench benchmark is accompanied by 12 queries.

2.3 Basic metrics

In this section, we present an initial set of metrics collected from the datasets presented in the previous section.

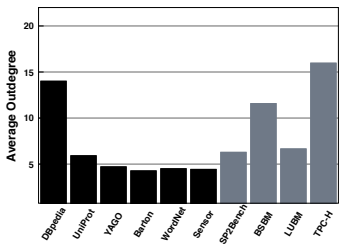

Figure 3: Average Outdegree

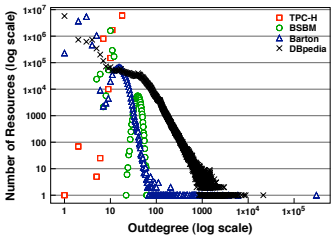

Figure 4: Outdegree distribution

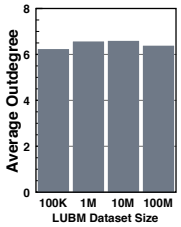

Figure 5: LUBM

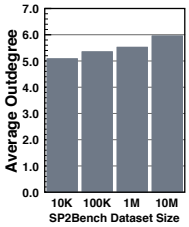

Figure 6: SP^{2}Bench

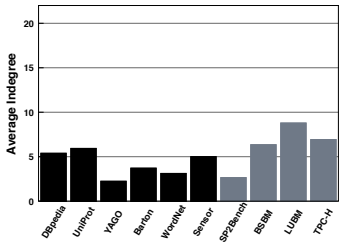

Figure 7: Average Indegree

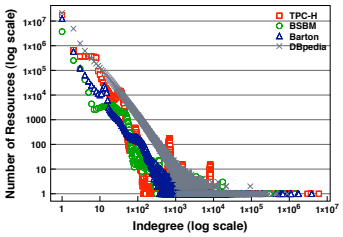

Figure 8: Indegree distribution

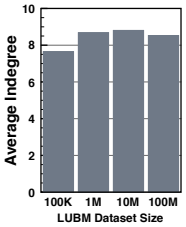

Figure 9: LUBM

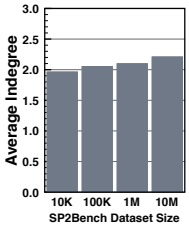

Figure 10: SP^{2}Bench

2.3.1 *Collecting the metrics*

To collect these metrics, the following procedure was followed:

■ **Step 1:** For some of the datasets (*e.g.,* LUBM), the dataset triples were distributed over a (large) number of files. Therefore, the first step in our procedure is to assemble all the triples into a single file. Hereafter, we use the dataset-independent file name SDF.rdf (Single Dataset File) to refer to this file.

■ **Step 2:** We also perform some *data cleaning* and *normalization*. In more detail, some of the real datasets contain a small percentage of triples that are syntactically incorrect. In this stage, we identify such triples, and we either correct the syntax, if the fix is obvious (*e.g.,* missing quote or angle bracket symbols), or we drop the triple from consideration, when the information in the triple is incomplete. We also drop triples in a reified form (*e.g.,* as in UniProt) and normalize all the datasets by converting all of them in the N-Triples format, which is a plain text RDF format, where each line in the text corresponds to a triple, and each triple is represented by the subject, property and object separated by space and the line terminates with a full stop symbol. We refer to SDF.nt as the file with the N-Triples representation of file SDF.rdf.

■ **Step 3:** We generate three new files, namely SDF_subj.nt, SDF_prop.nt, and SDF_obj.nt, by independently sorting the file SDF.nt by the subjects, properties and objects of the triples in SDF.nt. Each sorted output file is useful for different types of collected metrics, and the advantage of sorting is that the corresponding metrics can be collected by making a *single pass* of the sorted file. Although the sorting simplifies the computation cost of metrics, there is an initial considerable overhead since sorting files with billions of triples that occupy many GBs on disk requires large amounts of memory and processing power (for some datasets, each individual sort took more than two days in a dual processor server with 24GB of memory and 6TB of disk space). However, the advantage of this approach is that the sorting need only be done *once*. After sorting is done, metrics can be collected efficiently and new metrics can be developed that take advantage of the sort order. Another important advantage of sorting the SDF.nt file is that this way duplicate triples are eliminated during the sorting process. Such duplicate triples occur especially when the input dataset is originally split into multiple files.

■ **Step 4:** We select the SDF_subj.nt file generated in the previous step, and use it to extract the type system of the current dataset. The reason for extracting the type system will become clear in the next section where we introduce the structuredness metrics.

■ **Step 5:** We use file SDF_subj.nt to collect metrics such as counting the *number of subjects* and *triples* in the input dataset, as well as detailed statistics about the *outdegree* of the subjects (*i.e.,* the number of properties associated with the subject); we use file SDF_prop.nt to collect metrics such as the *number of properties* in the dataset as well as detailed statistics about the occurrences of each property; and we use file SDF_obj.nt to collect metrics such as the *number of objects* in the dataset as well as detailed statistics about the *indegree* of the objects (*i.e.,* the number of properties associated with the object).

Overall, to collect metrics we had to process over 2.5 billion triples from our input datasets, and we generated approximately 2TB of intermediate and output files. In the next section, we present an analysis of the collected metrics.

2.3.2 *Analysis of metrics*

Figures 1 to 18 show the collected primitive metrics for all our datasets. In the figures, we cluster the real and benchmark data sets together, and we use black bars for the former and gray bars for the latter datasets.

Figure 1 shows (in logarithmic scale) the space required for each dataset on disk, while Figure 2 shows in logarithmic scale the number of triples per dataset. Notice that there is a clear correlation between the two numbers, and the majority of datasets have between 10^7 to 10^8 triples, with the exception of UniProt which has in the order of 10^9 triples.

In terms of the indegree and outdegree, we note that some of the datasets have a power law-like distribution of indegree/outdegree, as shown in Figures 4 and 8, for only four of our input datasets (two benchmark and two real). In both figures, we see that there is a large concentration of entities around the 1 to 10 range (*i.e.,* the average number of properties of a subject or to an object is between 1 and 10) while a small number of entities have very large numbers of indegree and outdegree. In Figures 3 and 7 we plot the average outdegree and indegree for all our datasets. In terms of the standard deviation of the average outdegree, for most datasets this is

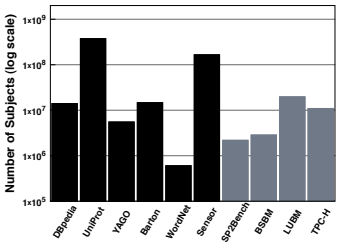

Figure 11: Dataset subjects

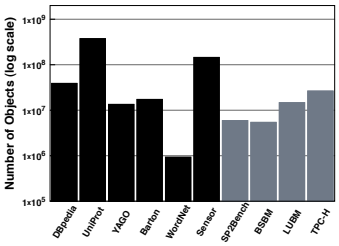

Figure 12: Dataset objects

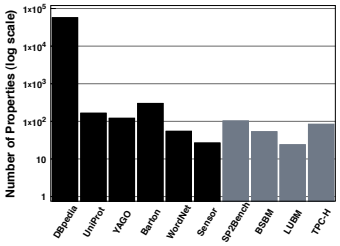

Figure 13: Dataset properties

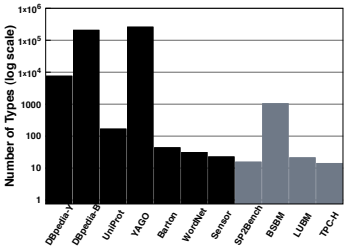

Figure 14: Number of Types

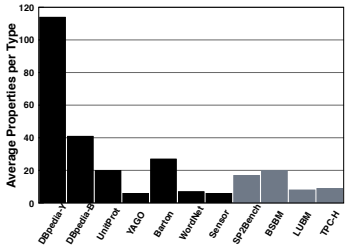

Figure 15: Average type properties

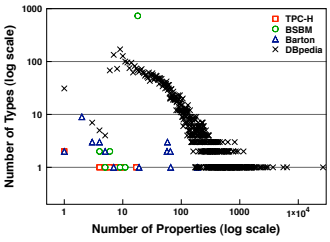

Figure 16: Type properties distribution for 4 datasets

relatively small (between 3 and 6) with the exception of DBpedia, Sensor, and Barton datasets. For these three datasets the deviation is in the order of 40, 60 and 80, respectively. In terms of the standard deviation of the average indegree, for almost all datasets this is relatively large (in the order of many hundreds). Finally, in Figures 5 and 6 we plot the average outdegrees for two of our benchmark datasets, namely, for LUBM and SP^{2}Bench, for various dataset sizes. The figures clearly illustrate that outdegrees are not a function of dataset sizes, since the average outdegree remains almost constant across datasets whose sizes span across four orders of magnitude. Figures 9 and 10 show that the same is true for indegrees.

For the next set of figures, we consider the number of subjects, objects and properties per dataset, shown (in log scale) in Figures 11, 12 and 13, respectively. The first two figures show that each dataset has subjects and objects in the same order of magnitude. However, looking at the raw data reveals that the TPC-H, BSBM, SP^{2}Bench, WordNet, Barton, YAGO and DBpedia datasets have more objects than subjects, whereas the opposite holds for the LUBM, UniProt and Sensor datasets. In terms of properties, most datasets have around 100 distinct properties, with the exception of DBpedia which has almost 3 orders of magnitude more properties.

Figure 14 shows the number of types in the type system of each dataset. Most datasets have only a few types (less than 100), with the exception of DBpedia-Y, DBpedia-B and the YAGO datasets. DBpedia-Y has approximately 5,000 types, while both DBpedia-B and YAGO have more than 140,000 types. Figure 15 shows the average number of properties per type. Notice that most datasets have less than 20 properties per type (on average) with the exception of DBpedia-Y which has approximately 111 properties per type, and DBpedia-B which has 38 properties per type. We must note that to compute the average number of properties per type we cannot just divide the distinct number of properties (shown in Figure 13) by the number of types (shown in Figure 14) since the former number counts properties irrespectively of the types these properties belong to. Such a division would incorrectly state that DBpedia has only 8 properties per type on average. This is also verified by Figure 16 which shows the actual distribution of properties per type for 4 datasets. In the x-axis we plot the number of properties and in the y-axis we plot how many types in the dataset have this number of properties.

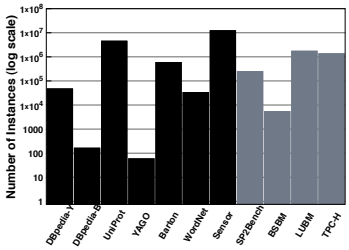

Figure 17: Average number of instances per type

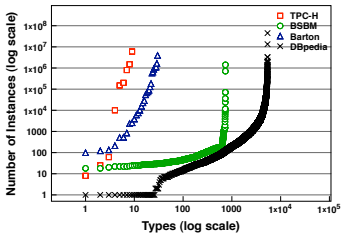

Figure 18: Instance distribution for 4 datasets

In terms of the number of instances for these types, Figure 17 shows that on average in most datasets a type has approximately 10^5 instances. A notable exception to this is the YAGO dataset. Since YAGO is essentially an ontology, there are only a few instances in the dataset (and a lot of types, as we already mentioned). It is interesting to notice the difference between DBpedia-B and DBpedia-Y in this metric. Since the type system in DBpedia-B is more *fine-grained*, there are only a few instances for each type (approximately, on average, 100 instances for each of the 150,000 DBpedia-B types). On the other hand, DBpedia-Y has a more *coarse-grained* type system, with only 5,000 types and these types are bound to aggregate a larger number of instances. Indeed, in DBpedia-Y there are approximately, on average, 30,000 instances per type. In terms of more concrete distributions of instances per type, Figure 18 shows how instances are distributed in various types, for 4 of our datasets. In the x-axis of the figure, we plot the types by assigning a unique id to each type of each dataset. In the y-axis, we plot the exact number of instances for each type.

2.3.3 Discussion

What can we say about the structure and nature of these datasets? Not much, apparently, if all we have are these metrics. UniProt is the dataset with the most triples, TPC-H is the dataset with the biggest average outdegree, and LUBM the dataset with the largest average indegree. DBpedia has the most properties, and YAGO has the most types. Different datasets have different characteristics for different primitive metrics. Can we say that YAGO is more structured than UniProt? Or can the metrics capture the intuition that TPC-H is more structured than DBpedia? *It is really hard from the above primitive metrics to say anything concrete about the structure of the datasets under consideration, and how they compare to each other*. Even more importantly, it is impossible to compare a new dataset with the analyzed datasets.

Each primitive metric quantifies some specific aspect of each dataset. Each metric in isolation offers no clues about the dataset as a whole. Clearly, what is required here is some intuitive way to combine all these metrics into a single composite metric that can characterize each dataset by itself, and by comparison to other datasets. In the next section, we introduce such a composite metric.

3. COVERAGE AND COHERENCE

In what follows, we formally define the notion of structuredness (through the coverage and coherence metrics) and show the values of these metrics for the datasets introduced in the previous section.

3.1 The metrics

Intuitively, the level of structuredness of a dataset D with respect to a type T is determined by how well the instance data in D *conform* to type T. Consider for example the dataset D of RDF triples in Figure 19(a). For simplicity, assume that the type T of these triples has properties name, office and ext. If each entity (subject) in D sets values for most (if not all) of the properties of T, then all the entities in D have a fairly similar structure that conforms to T. In this case, we can say that D has *high structuredness* with respect to T. This is indeed the case for the dataset D in Figure 19(a). Consider now a dataset D_M that consists of the union D ∪ D' of triples in Figures 19(a) and (b). For illustration purposes, consider a type T_M with properties major and GPA, in addition to the properties of T. Dataset D_M has low structuredness with respect to T_M. To see why this is the case, notice that type T_M *bundles* together entities with overlapping properties. So, while all entities in D_M have a value for the name property, the first three entities (those belonging to dataset D) set values only for the office and ext properties, while the last three entities (those belonging to dataset D') only set values for the major and GPA properties. The objective of our work is to measure the level of structuredness of a dataset (whatever it is), and to generate datasets with a desired (high or low) structuredness level for the purpose of benchmarking. In what follows, we formally define structuredness and show how it can be measured.

The first step in the computation of structuredness is clearly the determination of the type system $\mathcal{T}$. Conceivably, one might infer the type system $\mathcal{T}$ of a dataset D *intensionally*, through an RDFS or OWL specification associated with D. In practice, however, many datasets do not come with such specifications. Morever, when OWL or RDFS specifications exist, they often do not specify the schema or the type system; as an example in DBpedia the instances of a given type have many properties that are not defined in any of its multiple type systems [9]. This is because RDFS and OWL are mechanisms to infer new facts in the dataset rather than schema specifications. RDFS domain and range for example, are not constraints that the data needs to adhere to, but rather they are inference rules for a reasoner to apply to add new facts into the dataset D using open world semantics.

To address the fact that there is no schema available in RDF datasets to apply an intensional approach, we determine the type system $\mathcal{T}$ of D *extensionally* through the dataset itself, and thus do not need any *schema*-level information. Specifically, we scan D looking for triples whose property is <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> (hereafter abbreviated to w3type), and for these triples we extract the type T that appears as the object of the triple. That is,

DEFINITION 1. *The type system* $\mathcal{T}$ *of* D *is*

$$\mathcal{T} = \{ \mathsf{T} \mid \exists(\mathsf{s}, \mathsf{w3type}, \mathsf{T}) \in \mathsf{D} \} \quad (1)$$

Now, consider a particular type $\mathsf{T} \in \mathcal{T}$. Then,

DEFINITION 2. *The set* $\mathsf{I}(\mathsf{T}, \mathsf{D})$ *of instances of* T *in* D *is*

$$\mathsf{I}(\mathsf{T}, \mathsf{D}) = \{ \mathsf{s} \mid \exists(\mathsf{s}, \mathsf{w3type}, \mathsf{T}) \in \mathsf{D}\} \quad (2)$$

We then determine the properties of a type T through the union of all the properties that the instances of type T have. In more detail,

DEFINITION 3. *The set* $\mathsf{P}(\mathsf{T})$ *of properties of* T *is*

$$\mathsf{P}(\mathsf{T}) = \{ \mathsf{p} \mid \mathsf{s} \in \mathsf{I}(\mathsf{T}, \mathsf{D}) \text{ and } \exists(\mathsf{s}, \mathsf{p}, \mathsf{o}) \in \mathsf{D} \} \quad (3)$$

Notice that we are not interested in the number of times a property p is set for an instance of T. Intuitively, this is because the structuredness of T is affected by the sparsity (presence) of its properties across instances, rather than the number of occurrences of the properties. Since an entity might belong to more than one types, the properties of this entity contribute to $\mathsf{P}(\mathsf{T})$, for each such type T.

DEFINITION 4. *The number of occurrences* $\mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T}, \mathsf{D}))$ *of a property* p *in* $\mathsf{I}(\mathsf{T}, \mathsf{D})$ *is*

$$\mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T}, \mathsf{D})) = | \{ \mathsf{s} \mid (\mathsf{s} \in \mathsf{I}(\mathsf{T}, \mathsf{D}) \text{ and } \exists(\mathsf{s}, \mathsf{p}, \mathsf{o}) \in \mathsf{D}\} | \quad (4)$$

The last definition counts the number of entities for which property p has its value set in the instances $\mathsf{I}(\mathsf{T}, \mathsf{D})$ of T. In this manner, we count essentially the number of times p is set in $\mathsf{I}(\mathsf{T}, \mathsf{D})$. Similar to the definition of $\mathsf{P}(\mathsf{T})$, we are only interested in whether a property is present or not for a particular entity instead of how many times the property is set for this entity. Going back to Figure 19(a), for the type and dataset defined there, $\mathsf{P}(\mathsf{T}) = \{\mathsf{name}, \mathsf{office}, \mathsf{ext}\}$, $\mathsf{I}(\mathsf{T}, \mathsf{D})$ is equal to $\{\mathsf{person0}, \mathsf{person1}, \mathsf{person2}\}$, while $\mathsf{OC}(\mathsf{office}, \mathsf{T})$ is equal to 2 (since property office is set for the first two entities, not the third entity in dataset D).

DEFINITION 5. *The coverage* $\mathsf{CV}(\mathsf{T}, \mathsf{D})$ *of* T *on a dataset* D *is*

$$\mathsf{CV}(\mathsf{T}, \mathsf{D}) = \frac{\sum_{\forall \mathsf{p} \in \mathsf{P}(\mathsf{T})} \mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T}, \mathsf{D}))}{|\mathsf{P}(\mathsf{T})| \times |\mathsf{I}(\mathsf{T}, \mathsf{D})|} \quad (5)$$

To understand the intuition behind coverage, consider Figure 20. The figure considers the type T_M and dataset D_M, as defined by Figure 19 earlier. Note that $|\mathsf{P}(\mathsf{T})_M| = 5$, since there are five properties in the combined type system, and $|\mathsf{I}(\mathsf{T}_M, \mathsf{D}_M)| = 6$ for the six person instances. For each property p, the figure plots $\mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T}_M, \mathsf{D}_M))$. So, for example, for property name $\mathsf{OC}(\mathsf{name}, \mathsf{I}(\mathsf{T}_M, \mathsf{D}_M))$ is equal to 6, while for property major $\mathsf{OC}(\mathsf{major}, \mathsf{I}(\mathsf{T}_M, \mathsf{D}_M))$ is equal to 1. Now, as mentioned above, the structuredness of a type depends on whether the instances of the type set a value for all its properties. So, for a dataset D_M of T_M with *perfect* structuredness, every instance of the type in D_M sets all its properties, that

(person0, name, Eric)
(person0, office, BA7430)
(person0, ext, x4402)
(person1, name, Kenny)
(person1, office, BA7349)
(person1, office, BA7350)
(person1, ext, x5304)
(person2, name, Kyle)
(person2, ext, x6282)

(a) Dataset D

(person3, name, Timmy)
(person3, major, C.S.)
(person3, GPA, 3.4)
(person4, name, Stan)
(person4, GPA, 3.8)
(person5, name, Jimmy)
(person5, GPA, 3.7)

(b) Dataset D'

Figure 19: RDF sample triples

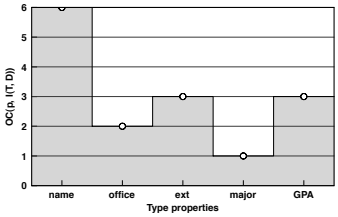

Figure 20: $\mathsf{OC}(\mathsf{p}, \mathsf{T})$ for each property p of T

is, $\mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T_M}, \mathsf{D_M}))$ is equal to $|\mathsf{I}(\mathsf{T_M}, \mathsf{D_M})\ |$, for every property $\mathsf{p} \in \mathsf{P}(\mathsf{T_M})$. Then, in the figure perfect structuredness translates to the area covered by $|\mathsf{P}(\mathsf{T_M})| \times |\mathsf{I}(\mathsf{T_M}, \mathsf{D_M})|$, *i.e.,* the area corresponding to the rectangle of the whole Figure 20 (this is also the denominator in the computation of $\mathsf{CV}(\mathsf{T_M}, \mathsf{D_M})$). In general, however, not all properties are set for every instance. Then, the shaded area in Figure 20 (computed by the numerator in the computation of $\mathsf{CV}(\mathsf{T_M}, \mathsf{D_M})$) corresponds to the current level of structuredness of $\mathsf{D_M}$ with respect to $\mathsf{T_M}$. Given the above, the formula of the coverage $\mathsf{CV}(\mathsf{T_M}, \mathsf{D_M})$ above is essentially an indication of the structuredness of $\mathsf{D_M}$ with respect to $\mathsf{T_M}$ normalized in the [0, 1] interval (with values close to 0 corresponding to low structuredness, and 1 corresponding to perfect structuredness). In our specific example, the value of the computed coverage for $\mathsf{CV}(\mathsf{T_M}, \mathsf{D_M})$ is equal to $\frac{6+2+3+1+3}{30} = 0.5$ which intuitively states that each instance of type $\mathsf{T_M}$ in dataset $\mathsf{D_M}$ only sets half of its properties.

Formula 5 considers the structuredness of a dataset with respect to a *single* type. Obviously, a dataset D has entities from multiple types, with each entity belonging to at least one of these types (if multiple instantiation is supported). It is possible that dataset D might have a high structuredness for a type T, say $\mathsf{CV}(\mathsf{T}, \mathsf{D})$ = 0.8, and a low structuredness for another type T', say $\mathsf{CV}(\mathsf{T'}, \mathsf{D})$ = 0.15. But then, what is the structuredness of the whole dataset with respect to our type system (set of all types) $\mathcal{T}$? We propose a mechanism to compute this, by considering the weighted sum of the coverage $\mathsf{CV}(\mathsf{T}, \mathsf{D})$ of individual types. In more detail, for each type T, we weight its coverage using the following formula:

$$\mathsf{WT}(\mathsf{CV}(\mathsf{T}, \mathsf{D})) = \frac{|\mathsf{P}(\mathsf{T})| + |\mathsf{I}(\mathsf{T}, \mathsf{D})|}{\sum_{\forall \mathsf{T'} \in \mathcal{T}} (|\mathsf{P}(\mathsf{T'})| + |\mathsf{I}(\mathsf{T'}, \mathsf{D})|)} \quad (6)$$

where $|\mathsf{P}(\mathsf{T})|$ is the number of properties for a type T, $|\mathsf{I}(\mathsf{T}, \mathsf{D})|$ is the number of entities in D of type T, and the denominator sums up these numbers for all the types in the type system $\mathcal{T}$. The weight formula has a number of desirable properties: It is easy to see that if the coverage $\mathsf{CV}(\mathsf{T}, \mathsf{D})$ is equal to 1, for every type T in $\mathcal{T}$, then the weighted sum of the coverage for all types T in $\mathcal{T}$ is equal to 1. The formula also gives higher weight to types with more instances. So, the coverage of a type with, say a single instance, has a lower influence in the computation of structuredness of the whole dataset, than the coverage of a type with hundreds of instances. This also matches our intuition that types with a small number of instances are usually more structured than types with larger number of instances. Finally, the formula gives higher weight to types with a larger number of properties. Again, this matches our intuition that one expects to find less variance in the instances of a type with, say only two properties, than the variance that one encounters in the instances of a type with hundreds of properties. The latter type is expected to have a larger number of *optional* properties, and therefore if the type has high coverage, this should carry more weight than a type with high coverage which only has two properties.

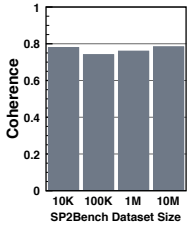

Figure 21: SP^{2}Bench coherence

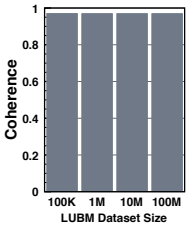

Figure 22: LUBM coherence

We are now ready to compute the structuredness, hereafter termed as *coherence*, of a whole dataset D with respect to a type system $\mathcal{T}$ (to avoid confusion with the term coverage which is used to describe the structuredness of a single type).

DEFINITION 6. *We define the coherence* $\mathsf{CH}(\mathcal{T}, \mathsf{D})$ *of a dataset* D *with respect to a type system* $\mathcal{T}$ *as*

$$\mathsf{CH}(\mathcal{T}, \mathsf{D}) = \sum_{\forall \mathsf{T}\ in\ \mathcal{T}} \mathsf{WT}(\mathsf{CV}(\mathsf{T}, \mathsf{D})) \times \mathsf{CV}(\mathsf{T}, \mathsf{D}) \quad (7)$$

3.2 Computing coherence

To compute the coherence of an input dataset, we consider file SDF_subj.nt (see Section 2.3.1). Remember that the file contains all the triples in the dataset (after cleaning, normalization and duplicate elimination) expressed in the N-Triples format. We proceed by *annotating* each triple in SDF_subj.nt with the type of triple's subject and object. This process converts each triple to a quintuple. We call the resulted file SDF_WT.nt (for Single Dataset File With Types). One more pass of the SDF_WT.nt file suffices to collect for each type T of the dataset the value of $\mathsf{OC}(\mathsf{p}, \mathsf{I}(\mathsf{T}, \mathsf{D}))$, for each property p of T. At the same time, we compute the values for $\mathsf{P}(\mathsf{T})$ and $\mathsf{I}(\mathsf{T}, \mathsf{D})$, and at the end of processing the file we are in a position to compute $\mathsf{CV}(\mathsf{T}, \mathsf{D})$, $\mathsf{WT}(\mathsf{CV}(\mathsf{T}, \mathsf{D}))$ and finally $\mathsf{CH}(\mathcal{T}, \mathsf{D})$.

In the introduction, we claim that coherence (structuredness) is relatively fixed in existing benchmarks. Figures 21 and 22 verify our claim. In more detail, Figure 21 shows the coherence of SP^{2}Bench datasets, as we scale the size of the generated data in the benchmark. The figure shows that coherence does not change for the different sizes, and more importantly that even the minor fluctuations are outside the user's control. Figure 22 shows the trend for the LUBM benchmark where coherence is practically constant (the situation is similar for the other benchmarks as well).

Figure 23 shows the computed coherence values for all our datasets. A few important observations are necessary here. Remember our choice of the TPC-H dataset as our baseline over which to compare all the other datasets. Since TPC-H is a relational dataset, we expect that it will have high structuredness. Indeed, *our coherence metric verifies this since TPC-H has a computed coherence equal to 1.* Contrast this result with Figures 1 to 18 which show the collected primitive metrics. None of the primitives metrics could have been used to identify and highlight the structuredness of TPC-H.

The second important observation we make is *the striking distinction between benchmark datasets and real datasets*. Indeed,

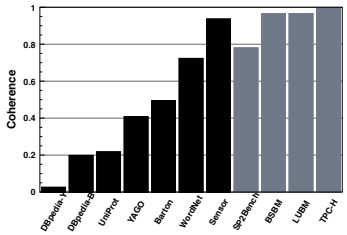

Figure 23: Coherence metric for all input datasets

there is a clear divide between the two sets of datasets, with the former datasets being fairly structured and relational-like (given the proximity of their coherence value to the one of TPC-H), while the latter datasets cover the whole spectrum of structuredness. DBpedia, a fairly popular and well-known dataset, is extremely unstructured with a coherence value of 0.002 for DBpedia-Y and 0.175 for DBpedia-B. Intuitively, this low coherence compared to a dataset like TPC-H can be interpreted as follows: If one considers two instances of the same type in DBpedia, with high probability, the properties set by the two instances are expected to be non-overlapping. Given that the instances belong to the same type, this is a rather surprising and interesting characteristic of the dataset (the situation is actually worse in DBpedia-Y since it *aggregates* more diverse instances under the same type, when compared to DBpedia-B). Conversely, for TPC-H one expects that for any two instances of the same type, the instances have exactly the same properties. Such a difference in the characteristics of the data is bound to have consequences on the storage of the data (TPC-H RDF data fit comfortably in a relational database, while DBpedia data will probably result in relational tables with lots of NULL values, if a relational representation of RDF data is chosen), and it also has effects on how the data are indexed, and queried (a simple SPARQL query returning all resources to a specified property always has the same selectivity for TPC-H, but the selectivity is different for each property in DBpedia). But these are some key dimensions existing RDF stores are tested and compared against each other. Remember in the introduction that we posed a simple question, namely, which types of RDF data existing RDF stores are tested against. *Figure 23 is the answer to this question: Existing RDF stores are tested and compared against each other with respect to datasets that are not representative of most real RDF data.* Again, we refer the reader to Figures 1 to 18 which show the collected primitive metrics. Nowhere in these figures is this distinction as clear cut and visible as in Figure 23. Yet, in a sense our coverage and coherence formulas incorporate the underlying metrics shown in these figures, since the formulas use the number of types, properties, properties per type, instance entities per type (distinct subjects for a type), and instance properties for these entities for the computation of our introduced metrics.

Since RDF benchmarks are distinct from real RDF datasets, the key question is whether we can design a benchmark that generates data which more accurately represent real RDF data, in terms of structuredness. We investigate this next, and present a benchmark generator that can convert any existing dataset (even datasets generated with existing benchmarks) into a series of datasets with varying characteristics in terms of size and structuredness.

4. BENCHMARK GENERATION

There are two overall methods to be considered in generating benchmarks with structuredness that better represent real datasets. The first method, similar to the approach taken by the developers of LUBM, SP^{2}Bench and BSBM is to generate a dataset with a given coherence and size from scratch. The main issue with this approach is that the generated benchmark is domain specific. In all the aforementioned benchmark datasets, the relationships and relative cardinalities between different types come from knowledge of the target domain, and are hard-coded into the generation algorithms and thus not controllable by the user. For instance, the relative cardinalities of professors and students or students and courses in LUBM are a feature of the generation algorithm and not available to the user generating the benchmark.

The second method, which applies to any domain for which there already is a benchmark or real dataset, involves taking a given dataset, *and producing a dataset with a specified smaller size and coherence*. Ideally, we would like to take an existing benchmark or real dataset D and produce a dataset D' with a specified size $|D'| < |D|$ and a specified coherence $CH(\mathcal{T}, D') < CH(\mathcal{T}, D)$. We believe the latter method has a larger impact in practice, since it can be used on top of any already existing benchmark or real-world dataset.

The central idea behind our approach is that under certain circumstances we can estimate the impact that removing a set of triples with the same subject and property can have on coherence. Let *(s, p, o)* be a triple from D and let $\mathcal{T}$ (s) = $\{T_s^1, \ldots, T_s^n\}$ be the set of types[1] of instance s. We are going to compute the impact on coherence of removing all triples with subject *s* and property *p* from D, under the following two assumptions:

- (A1) We do not completely remove property *p* from any of the types $\{T_s^1, \ldots, T_s^n\}$. That is, after the removal, for each type there will exist instances that have property *p*.
- (A2) We do not completely remove instance *s* from the dataset. This can be very easily enforced by keeping the triples $\{s, \text{rdf:type}, T_s^i\}$ in the dataset.

Under these two assumptions, the weights WT(CV(T, D)) for the coverage of any type $T \in \mathcal{T}(s)$ do not change since we keep the same number of properties and instances for each such type. For each type $T \in \mathcal{T}(s)$, we can compute the new coverage as

$$CV(T, D)' = \frac{\sum_{\forall q \in P(T) - \{p\}} OC(q, I(T, D)) + OC(p, I(T, D) - 1)}{|P(T)| \times |I(T, D)|} \quad (8)$$

Note that there is one less instance (specifically, *s*) that has property p for type T. It is evident from this formula that removing all triples with subject *s* and property *p* will decrease the coverage of all types $T \in \mathcal{T}(s)$ by CV(T, D) - CV(T, D)'. Consequently, we can also compute the coherence $CH(\mathcal{T}, D)$' of D after removing these triples by simply replacing CV(T, D) with CV(T, D)' for all types T in $\mathcal{T}$ (s). Finally, we compute the impact on the coherence of D of the removal as:

$$coin(\mathcal{T}(s), p) = CH(\mathcal{T}, D) - CH(\mathcal{T}, D)'$$

Let us illustrate this process with an example. Consider the dataset D_M introduced in Figure 19, and assume we would like to remove the triple (person1, ext, x5304) from D_M. Then the new coverage for the type person in this dataset becomes $\frac{6+2+2+1+3}{30} \approx 0.467$, hence the impact on the coverage of person is approximately $0.5 - 0.467 = 0.033$. In this example, D_M contains a single type, therefore the coherence of the dataset is the same as the coverage for person, which brings us to coin({person },ext) ≈ 0.033.

[1]We remind the reader that a single instance *s* can have multiple types, for example a GraduateStudent can also be a ResearchAssistant.

4.1 Benchmark generation algorithm

In this section we outline our algorithm to generate benchmark datasets of desired coherence and size by taking a dataset D and producing a dataset D' $\subset$ D such that $\mathsf{CH}(\mathcal{T}, \mathsf{D}') = \gamma$ and $|\mathsf{D}'| = \sigma$, where γ and σ are specified by the user. To do this, we need to determine which triples need to be removed from D to obtain D'. We will formulate this as an integer programming problem and solve it using an existing integer programming solver.

Previously in this section, for a set of types $\mathcal{S} \subseteq \mathcal{T}$ and a property p, we have shown how to compute $\mathsf{coin}(\mathcal{S},\mathsf{p})$, which represents the impact on coherence of removing all triples with subjects that are instances of the types in $\mathcal{S}$ and properties equal to p. For simplification, we will overload notation and use $|\mathsf{coin}(\mathcal{S},\mathsf{p})|$ to denote the number of subjects that are instances of all the types in $\mathcal{S}$ and have at least one triple with property p, *i.e.,*

$$|\mathsf{coin}(\mathcal{S},\mathsf{p})| = |\{s \in \bigcap_{T \in \mathcal{S}} \mathsf{I}(\mathsf{T},\mathsf{D}) | \exists (s,p,v) \in \mathsf{D}\}|$$

Our objective is to formulate an integer programming problem whose solutions will tell us how many "coins" (triples with subjects that are instances of certain types and with a given property) to remove to achieve the desired coherence γ and size σ. Although we could use the same integer programming approach (with a different objective function and different constraints) for adding triples to the dataset, we did not explore this option because adding triples means we would have to "invent" new values/URIs/blank nodes, which is effectively what most benchmark generators do already. The challenge here is to have a systematic way to invent meaningful nodes that can have an impact on query evaluation (for instance, the invented nodes are returned as part of query answers) over various datasets; otherwise, the invented nodes have little value for the benchmarking purpose. We therefore took the approach of removing triples in existing datasets to create versions that reflect varying degrees of coherence and data size. This approach has an additional advantage that it can be applied to multiple different benchmark datasets with different primitive metrics (such as number of types, number of properties, and average indegree) and is still be able to generate a dataset with desired size and coherence.

In our benchmark generation algorithm, we will use $\mathsf{X}(\mathcal{S},\mathsf{p})$ to denote the integer programming variable representing the number of coins to remove for each type of coin. In the worst case, the number of such variables (and the corresponding coin types) for D can be $2^{\tau}\pi$, where τ is the number of types and π is the number of properties in the dataset. However, in practice, many type combinations will not have any instances. For instance, in LUBM, we will not find instances of UndergraduateStudent that are also instance of Course or Department. For LUBM, we found that although there are 15 types and 18 properties, we only have 73 valid combinations (sets of types and property with at least one coin available).

To achieve the desired coherence, we formulate the following constraint and maximization criteria for the integer programming problem.

$$\sum_{\mathcal{S} \subseteq \mathcal{T}, p} \mathsf{coin}(\mathcal{S},\mathsf{p}) \times \mathsf{X}(\mathcal{S},\mathsf{p}) \leq \mathsf{CH}(\mathcal{T},\mathsf{D}) - \gamma \qquad \text{(C1)}$$

$$MAXIMIZE \sum_{\mathcal{S} \subseteq \mathcal{D}, p} \mathsf{coin}(\mathcal{S},\mathsf{p}) \times \mathsf{X}(\mathcal{S},\mathsf{p}) \qquad \text{(M)}$$

Inequality C1 states that the amount by which we decrease coherence (by removing coins) should be less than or equal than the amount we need to remove to get from $\mathsf{CH}(\mathcal{T},\mathsf{D})$ (the coherence of the original dataset) to γ (the desired coherence). The objective function M states that the amount by which we decrease coherence should be maximized. The two elements together ensure that we decrease the coherence of D by as much as possible, while not going below γ.

We will also put lower and upper bounds on the number of coins that can be removed. Remember that assumption (A1) required us not to completely remove any properties from any types, so we will ensure that at least one coin of each type remains. Furthermore, we will enforce assumption (A2) about not removing instances from the dataset by always keeping triples with the rdf:type property.

$$\forall \mathcal{S} \subseteq \mathcal{T}, p \;\; 0 \leq \mathsf{X}(\mathcal{S},\mathsf{p}) \leq |\mathsf{coin}(\mathcal{S},\mathsf{p})| - 1 \qquad \text{(C2)}$$

Achieving the desired size σ is similar, but requires an approximation. Under the simplifying assumption that all properties are single-valued (*i.e.,* there is only one triple with a given subject and a given property in D), we could write the following constraint:

$$\sum_{\mathcal{S} \subseteq \mathcal{T}, p} |\mathsf{X}(\mathcal{S},\mathsf{p})| = |\mathsf{D}| - \sigma$$

This equation would ensure that we remove exactly the right number of coins to obtain size σ assuming that all properties are single-valued (meaning one coin represents exactly one triple). However, this assumption does not hold for any of the datasets we have seen. In particular, for LUBM, many properties are multi-valued. As an example, a student can be enrolled in multiple courses, a paper has many authors, etc. We will address this by computing an average number of triples per coin type, which we denote by $\mathsf{ct}(\mathcal{S},\mathsf{p})$, and relaxing the size constraint as follows:

$$(1-\rho) \times (|\mathsf{D}| - \sigma) \leq \sum_{\mathcal{S} \subseteq \mathcal{T}, p} \mathsf{X}(\mathcal{S},\mathsf{p}) \times \mathsf{ct}(\mathcal{S},\mathsf{p}) \qquad \text{(C3)}$$

$$\sum_{\mathcal{S} \subseteq \mathcal{T}, p} \mathsf{X}(\mathcal{S},\mathsf{p}) \times \mathsf{ct}(\mathcal{S},\mathsf{p}) \leq (1+\rho) \times (|\mathsf{D}| - \sigma) \qquad \text{(C4)}$$

In these two constraints, ρ is a relaxation parameter. The presence of ρ is required because of the approximation we introduced by using the average number of triples per coin. In practice, setting ρ helped us tune the result of our algorithm closer to the target coherence and size.

We can now outline the algorithm that generates a benchmark dataset of desired coherence γ and size σ from an input dataset D:

(Step 1) Compute the coherence $\mathsf{CH}(\mathcal{T},\mathsf{D})$ and the coin values $\mathsf{coin}(\mathcal{S},\mathsf{p})$ and average triples per coin $\mathsf{ct}(\mathcal{S},\mathsf{p})$ for all sets of types $\mathcal{S} \subseteq \mathcal{T}$ and all properties p.

(Step 2) Formulate the integer programming problem by writing constraints C1, C2, C3, C4 and objective function M. Solve the integer programming problem.

(Step 3) If the problem did not have a solution, then try to make the dataset smaller by removing a percentage of instances and continue from Step 1.

(Step 4) If the problem had a solution, then for each coin given by $\mathcal{S}$ and p, remove triples with $\mathsf{X}(\mathcal{S},\mathsf{p})$ subjects that are instances of types in $\mathcal{S}$ and have property p.

(Step 5) If the resulting dataset size is larger than σ, perform post-processing by attempting to remove from triples with the same subject and property.

We have previously explained in detail how steps (1) and (2) can be executed. Step (3) is an adjustment in case there is no solution to the linear programming problem. Remember that assumption (A2) required us not to remove entire instances from the dataset if the integer programming formulation is to produce the correct number of coins to remove. In practice we found that for certain combinations

of γ and σ the integer programming problem does not have solutions – in particular for cases where the desired coherence γ is high, but the desired size σ is low (*i.e.,* we have to remove many coins, but we should not decrease coherence much). For these cases, we found that we can remove entire instances from D first to bring down its size, then reformulate the integer programming problem and find a solution. The intuition behind this approach is that when starting with original datasets of very high coherence (e.g., LUBM, TPC-H, etc.), removing instances uniformly at random will not decrease coherence much (if at all), since the coverage for all types is high, but it can decrease dataset size to a point where our integer programming approach finds a solution.

To perform this removal of instances effectively, we needed to understand how many instances to remove from the original dataset to have a high probability of finding a solution on the new dataset. In our experiments, the integer programming problem always had a solution for $\frac{\sigma}{|\mathsf{D}|} \approx \frac{\gamma}{\mathsf{CH}(\mathcal{T},\mathsf{D})}$. Therefore, we want to remove enough instances as to have the size of our new dataset approximately $\frac{\mathsf{CH}(\mathcal{T},\mathsf{D})}{\gamma} \times \sigma$. Assuming that the dataset size is proportional to the number of instances[2], then we should remove uniformly at random a proportion of $1 - \frac{\mathsf{CH}(\mathcal{T},\mathsf{D})}{\gamma} \times \frac{\sigma}{|\mathsf{D}|}$ instances to arrive at a dataset for which we have a good chance of solving the integer programming problem. After this process, we must restart the algorithm since the coherence and the numbers of coins for the dataset after the instance removal may be different than those of the original dataset.

In Step (4), we perform the actual removal of triples according to the solutions to the integer programming problem. Step (5) is a post-processing step that attempts to compensate for the approximation introduced by constraints C3 and C4 of the integer programming problem. Specifically, if the solution we obtain after Step (4) has a size higher than σ, then we can compensate by looking at triples with the same subject and property. Note that based on the way we have defined coverage for types, the formula measures whether instances have *at least one value* for each property of that type. Therefore, if a property is multi-valued, we can safely remove the triples containing extra values (ensuring that we keep at least one value), and therefore reduce the size of the dataset. While this step is optional, it can improve the match between σ and the actual size of the resulting dataset.

Note that the algorithm presented in this section performs at least two passes through the original dataset D. The first pass is performed in Step (1) to compute coherence and coin values and the average number of triples per coin. The second pass is performed in Step (4), where coins are removed from D to generate the desired dataset. If the integer programming problem does not have a solution, then at least four passes are required: one pass in Step (1), one pass in Step (3) to remove instances, a third pass in Step (1) to compute coherence and coin values after instance removal and finally a fourth pass in Step (4) to remove coins from the dataset. In addition, in either case there may be an additional pass over the resulting dataset to adjust for size (Step 5).

Our discussion of future work in Section 5 includes a query evaluation study that looks at how coherence of a dataset affects query performance. To perform such a study, we will need datasets with the same size but varying coherence and a set of queries that return the same results for each dataset. In our implementation, we added an additional option of specifying a list of triples that should not be removed from a dataset. This option impacts Step (2) where we need impose different upper bounds on the $\mathsf{X}(\mathcal{S},\mathsf{p})$ (to account for coins that cannot be removed) and in Step (4), to avoid removing the "forbidden" triples. The following process helps obtain the necessary datasets for a query evaluation study:

1. Start with a dataset D of high coherence and size.
2. Generate a first dataset D_0 with the desired size σ and the same coherence as D by removing entire instances uniformly at random.
3. Issue queries over D_0 and record *all triples required to produce answers* to these queries.
4. Generate datasets $\mathsf{D}_1, \ldots, \mathsf{D}_n$ of different coherence and the same size σ using the benchmark generation algorithm presented here. Use the set of triples computed in the previous step as "forbidden" triples during the generation to ensure that all datasets yield the same answers to all queries.

By experimenting with the option of avoiding triples, we found that it works well for queries of medium to high selectivity. Queries of low selectivity require many triples to answer, thus greatly diminishing the set of useful coins that can reduce coherence of a dataset.

4.2 Experimental results and discussion

We implemented our benchmark generation algorithm in Java and performed experiments on multiple original datasets, both benchmark and real-world. We used lpsolve 5.5[3] as our integer programming solver with a timeout of 100 seconds per problem. We report results from running the benchmark generation on four servers with identical configuration: 2 dual-core 3GHz processors, 24GB of memory and 12 TB of shared disk space. We used these machines to run experiments in parallel, but we also performed similar experiments (one at a time) on a standard T7700 2.4 Ghz 3GB of RAM laptop machine.

We started with the LUBM dataset in different sizes (100K triples, 1, 10 and 100 million triples). For each, we executed benchmark generation with nine sets of parameters, $\gamma \in \{0.25, 0.5, 0.75\}$ and $\sigma \in \{0.25, 0.5, 0.75\} \times |\mathsf{D}|$. The results are shown in Figures 24–27. In each figure, the x axis represents the percentage of the original dataset size desired and the y axis represents the coherence desired (as an absolute value, not as a percentage of original coherence). The square markers in each figure represent the ideal output and the *X* markers represent the achieved size/coherence combination. We can easily see that with the exception of the 75% size, 0.25 coherence combination the algorithm achieves almost perfect matches. The relaxation parameter ρ was set to a default of 0.1 for all LUBM runs.

The 75% size, 0.25 coherence combination essentially requires that we lower coherence by a large margin (remember that the original coherence of LUBM is above 0.9), while maintaining most of the triples in the dataset. In this case, the integer programming solver maximizes the amount of coherence that can be removed (as instructed by the objective function), but cannot achieve perfect coherence. This is caused by the fact that we have an upper bound on the amount by which coherence should decrease (constraint C1), but no lower bound like we do for size. However, we noticed that by introducing a lower bound for the amount by which coherence can decrease has two downsides: (i) it introduces a new control parameter (in addition to ρ) and (ii) without adjustment of this parameter, the integer programming solver either times out or cannot solve the problem in many cases. As a result, we decided that the algorithm is more useful without the lower bound on coherence reduction (which means an upperbound on resulting coherence). As

[2]We found this to be true for all datasets we examined.

[3]http://lpsolve.sourceforge.net/5.5/

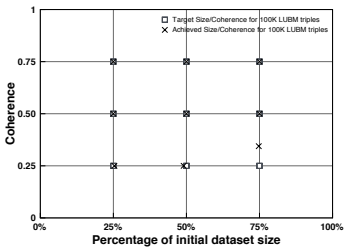

Figure 24: LUBM 100K

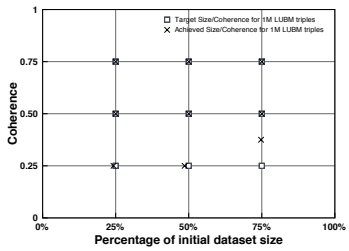

Figure 25: LUBM 1M

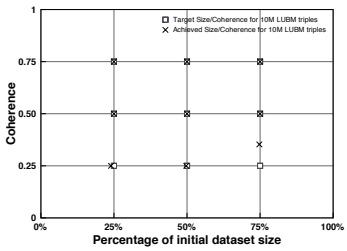

Figure 26: LUBM 10M

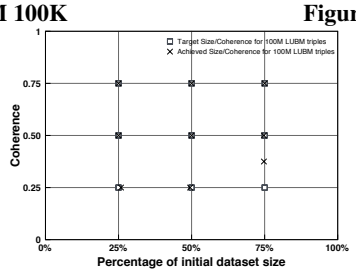

Figure 27: LUBM 100M

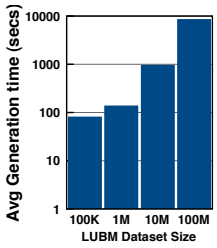

Figure 28: Running time: LUBM

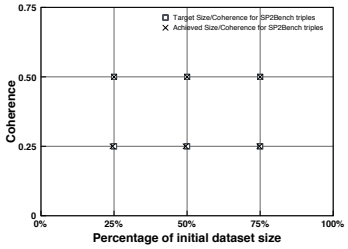

Figure 29: SP^{2}Bench

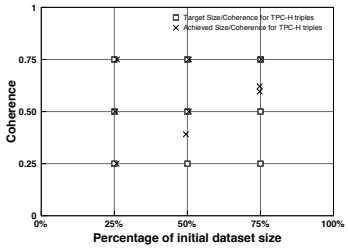

Figure 30: TPC-H

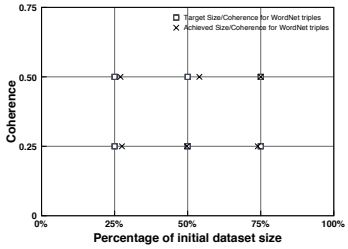

Figure 31: Wordnet

an alternative, in cases where such combinations are desired, the user can start with a larger dataset size – for instance, instead of starting with a 1 million triple dataset requiring 75% size and 0.25 coherence, the user can start with a 3 million triple dataset and request 25% size and 0.25 coherence.

The average running time for the benchmark generation over LUBM is shown in Figure 28 and is linear in the size of the dataset (in the figure, both axes are on a logarithmic scale), since the algorithm requires between 2 and 4 passes over the original dataset.

Next, we ran our algorithm with SP^{2}Bench, TPC-H and Wordnet as the original dataset. The results are shown in Figures 29 – 31. Note that for Wordnet and SP^{2}Bench, the maximal achievable coherence is 0.5 (given the original coherence of these datasets). SP^{2}Bench contains a number of RDF container objects, which contain elements in no particular order. In its n-triples format, each container had *numbered* predicate edges to each of the contained elements (e.g., the container was connected to value $v1$ by $p1$, value $v2$ by $p2$, and so on). Because this numbering of containment relationships is arbitrary, and it can skew any measurement of type coherence for the container type, we replaced all numbered predicate labels with a single containment relationship before generation. For TPC-H, the relaxation parameter ρ is 0.01 for all 9 points. Notice that as in the case of LUBM, the integer programming solver finds a sub-optimal solution for some points. So the solver finds a solution that has the right size but misses coherence. For SP^{2}Bench, the relaxation is 0.01 for the points with size 25% and 0.05 for the points with size 50%. For Wordnet, ρ varies between 0.01 and 0.05. We observed the following in practice about setting the relaxation parameter ρ. Low values of ρ mean that the bounds on the size of the generated dataset are tighter, therefore we are more likely to achieve the desired size. However, remember that there is a lower and an upper bound on size reduction (depending on ρ), whereas there is a single upper bound on coherence reduction. This means that if size bounds are tight, the integer programming solver will attempt to find the optimal coherence reduction within the given size bounds, which may not be very close to the desired coherence γ. On the other hand, if the size bounds are loose (higher ρ), that means that the integer programming solver may find a better optimal value for the coherence reduction, and our algorithm can still approach the desired size using the compensation in Step (5). Generally, choosing ρ depends on the original dataset size and the desired dataset size. For small initial datasets, having "loose" bounds (*i.e.,* $\rho = 0.1$) is a good option because it gives the integer programming solver a larger solution space to work for. For larger datasets, we do not want too much flexibility in the size, hence we can choose low relaxation values (*i.e.,* 0.01 in TPC-H). However, if the target dataset size is also large (75% of the original), we can again have higher values of the relaxation parameter because we still achieve a good match.

5. CONCLUSIONS

We presented an extensive study of the characteristics of real and benchmark RDF data. Through this study, we illustrated that primitive metrics, although useful, offer little insight about the inherent characteristics of datasets and how these compare to each other. We introduced a *macroscopic* metric, called coherence, that essentially combines many of these primitive metrics and is used to measure the structuredness of different datasets. We showed that while real datasets cover the whole structuredness spectrum, benchmark datasets are very limited in their structuredness and are mostly relational-like. Since structuredness plays an important role on how we store, index and query data, the above result indicates

that existing benchmarks do not accurately predict the behaviour of RDF stores in realistic scenarios. In response to this limitation, we also introduced a benchmark generator which can be used to generate benchmark datasets that actually resemble real datasets in terms of structuredness, size, and content. This is feasible since our generator can use any dataset as input (real or synthetic) and generate a benchmark out of it. On the technical side, we formally introduced structuredness through the coherence metric, and we showed how the challenge of benchmark generation can be solved by formulating it as an integer programming problem.

Using this work as a starting point, there are several avenues we are investigating next. Specifically, one important topic is the effects of structuredness on RDF storage strategies. Our preliminary investigation shows that while relational column stores are considered a popular and effective storage strategy for RDF data, this assumption holds true mainly for data with high structuredness. Such data often exhibit a small number of types in their type system, and each type has a small number of properties. Then, a column store approach to RDF requires only a few tens of tables in the RDBMS. However, data with low structuredness, like DBpedia or Yago, have large number of types and properties (in the thousands). For such datasets, a column store solution would require thousands of tables in RDBMS and is thus not expected to scale.

Another avenue we are investigating relates to the indexing and querying of RDF data. Structuredness influences both the type and the density of the indexes used. At the same time, it influences the selectivity of queries and therefore the overall query performance. We plan to investigate the concrete effects of structuredness on query performance and contrast the performance of equivalent queries over datasets with varying structuredness. Ideally, for this comparison to be meaningful, the queries should return identical results across all varying structuredness datasets. We have done some preliminary work, and included an exclusion list in our integer programming formulation so as to exclude certain triples (those that participate in the query result) from removal, as we compute a new dataset with smaller structuredness. It turns out however, that trying to maintain such triples for low-selectivity queries often leads to a linear programming problem that is impossible to solve. Therefore, we are considering other alternatives to meaningfully compare low-selectivity queries across the different datasets.

6. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] R. Apweiler, A. Bairoch, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L. S. Yeh. Uniprot: the universal protein knowledgebase. *Nucleic Acids Res.*, 32:D115–D119, 2004.

[4] M. Arenas and L. Libkin. A normal form for xml documents. In *PODS*, pages 85–96, 2002.

[5] BBC World Cup 2010 dynamic semantic publishing. `http://www.bbc.co.uk/blogs/bbcinternet/2010/07/bbc_world_cup_2010_dynamic_sem.html`.

[6] Best Buy jump starts data web marketing. `http://www.chiefmartec.com/2009/12/best-buy-jump-starts-data-web-marketing.html`.

[7] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[8] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia—a crystallization point for the web of data. *Web Semantics*, 7(3):154–165, 2009.

[9] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semant.*, 7:154–165, September 2009.

[10] C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[11] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68, 2002.

[12] D2R Server: Publishing Relational Databases on the Semantic Web. `http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/`.

[13] Data.gov. `http://www.data.gov/`.

[14] Data.gov.uk. `http://www.data.gov.uk/`.

[15] O. Erling and I. Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.

[16] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Semantics*, 3(2-3):158–182, 2005.

[17] S. Harris and N. Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, 2003.

[18] L3S Research Center D2R Server publishing the DBLP Bibliography Database. `http://dblp.l3s.de/d2r/`.

[19] Making a Semantic Web Business Case at Pfizer. `http://www.semanticweb.com/news/making_a_semantic_web_business_case_at_pfizer_161731.asp`.

[20] B. McBride. Jena: Implementing the rdf model and syntax specification. In *SemWeb*, 2001.

[21] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.

[22] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.

[23] H. Patni, C. Henson, and A. Sheth. Linked sensor data. 2010.

[24] Resource Description Framework (RDF). `http://www.w3.org/RDF/`.

[25] V. Sans and D. Laurent. Prefix based numbering schemes for xml: techniques, applications and performances. *Proc. VLDB Endow.*, 1(2):1564–1573, 2008.

[26] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. In *ICDE*, pages 222–233, 2009.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *J. Web Sem.*, 6(3):203–217, 2008.

[28] The MIT Barton Library dataset. `http://simile.mit.edu/rdf-test-data/`.

[29] The TPC Benchmark H. `http://www.tpc.org/tpch`.

[30] WordNet: A lexical database for English. `http://www.w3.org/2006/03/wn/wn20/`.

[31] XMark: An XML Benchmark Project. `http://www.xml-benchmark.org/`.

[32] YAGO: Yet Another Great Ontology. `http://www.mpi-inf.mpg.de/yago-naga/yago/`.