SP²Bench: A SPARQL Performance Benchmark

Michael Schmidt* ♯, Thomas Hornung ♯, Georg Lausen ♯, Christoph Pinkel ♮

♯*Freiburg University*
Georges-Koehler-Allee 51, 79110 Freiburg, Germany
{mschmidt|hornungt|lausen}@informatik.uni-freiburg.de

♮*MTC Infomedia OHG*
Kaiserstrasse 26, 66121 Saarbrücken, Germany
c.pinkel@mtc-infomedia.de

***Abstract*— Recently, the SPARQL query language for RDF has reached the W3C recommendation status. In response to this emerging standard, the database community is currently exploring efficient storage techniques for RDF data and evaluation strategies for SPARQL queries. A meaningful analysis and comparison of these approaches necessitates a comprehensive and universal benchmark platform. To this end, we have developed SP²Bench, a publicly available, language-specific SPARQL performance benchmark. SP²Bench is settled in the DBLP scenario and comprises both a data generator for creating arbitrarily large DBLP-like documents and a set of carefully designed benchmark queries. The generated documents mirror key characteristics and social-world distributions encountered in the original DBLP data set, while the queries implement meaningful requests on top of this data, covering a variety of SPARQL operator constellations and RDF access patterns. As a proof of concept, we apply SP²Bench to existing engines and discuss their strengths and weaknesses that follow immediately from the benchmark results.**

I. Introduction

The Resource Description Framework [1] (RDF) has become the standard format for encoding machine-readable information in the Semantic Web [2]. RDF databases can be represented by labeled directed graphs, where each edge connects a so-called *subject* node to an *object* node under label *predicate*. The intended semantics is that the *object* denotes the value of the *subject*'s property *predicate*. Supplementary to RDF, the W3C has recommended the declarative SPARQL [3] query language, which can be used to extract information from RDF graphs. SPARQL bases upon a powerful graph matching facility, allowing to bind variables to components in the input RDF graph. In addition, operators akin to relational joins, unions, left outer joins, selections, and projections can be combined to build more expressive queries.

By now, several proposals for the efficient evaluation of SPARQL have been made. These approaches comprise a wide range of optimization techniques, including normal forms [4], graph pattern reordering based on selectivity estimations [5] (similar to relational join reordering), syntactic rewriting [6], specialized indices [7], [8] and storage schemes [9], [10], [11], [12], [13] for RDF, and Semantic Query Optimization [14]. Another viable option is the translation of SPARQL into SQL [15], [16] or Datalog [17], which facilitates the evaluation with traditional engines, thus falling back on established optimization techniques implemented in conventional engines.

As a proof of concept, most of these approaches have been evaluated experimentally either in user-defined scenarios, on top of the LUBM benchmark [18], or using the Barton Library benchmark [19]. We claim that none of these scenarios is adequate for testing SPARQL implementations in a general and comprehensive way: On the one hand, user-defined scenarios are typically designed to demonstrate very specific properties and, for this reason, lack generality. On the other hand, the Barton Library Benchmark is application-oriented, while LUBM was primarily designed to test the reasoning and inference mechanisms of Knowledge Base Systems. As a trade-off, in both benchmarks central SPARQL operators like OPTIONAL and UNION, or solution modifiers are not covered.

With the **SP**ARQL **P**erformance **Bench**mark (SP²Bench) we propose a language-specific benchmark framework specifically designed to test the most common SPARQL constructs, operator constellations, and a broad range of RDF data access patterns. The SP²Bench data generator and benchmark queries are available for download in a ready-to-use format.[1]

In contrast to application-specific benchmarks, SP²Bench aims at a comprehensive performance evaluation, rather than assessing the behavior of engines in an application-driven scenario. Consequently, it is not motivated by a single use case, but instead covers a broad range of challenges that SPARQL engines might face in different contexts. In this line, it allows to assess the generality of optimization approaches and to compare them in a universal, application-independent setting. We argue that, for these reasons, our benchmark provides excellent support for testing the performance of engines in a comprising way, which might help to improve the quality of future research in this area. We emphasize that such language-specific benchmarks (e.g., XMark [20]) have found broad acceptance, in particular in the research community.

It is quite evident that the domain of a language-specific benchmark should not only constitute a representative scenario that captures the philosophy behind the data format, but also leave room for challenging queries. With the choice of the DBLP [21] library we satisfy both desiderata. First, RDF has been particularly designed to encode metadata, which makes

*The work of this author was funded by DFG grant GRK 806/2.

[1] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.28

DBLP an excellent candidate. Furthermore, DBLP reflects interesting social-world distributions (cf. [22]), and hence captures the social network character of the Semantic Web, whose idea is to integrate a great many of small databases into a global semantic network. In this line, it facilitates the design of interesting queries on top of these distributions.

Our data generator supports the creation of arbitrarily large DBLP-like models in RDF format, which mirror vital key characteristics and distributions of DBLP. Consequently, our framework combines the benefits of a data generator for creating arbitrarily large documents with interesting data that contains many real-world characteristics, i.e. mimics natural correlations between entities, such as power law distributions (found in the citation system or the distribution of papers among authors) and limited growth curves (e.g., the increasing number of venues and publications over time). For this reason our generator relies on an in-depth study of DBLP, which comprises the analysis of entities (e.g. articles and authors), their properties, frequency, and also their interaction.

Complementary to the data generator, we have designed 17 meaningful queries that operate on top of the generated documents. They cover not only the most important SPARQL constructs and operator constellations, but also vary in their characteristics, such as complexity and result size. The detailed knowledge of data characteristics plays a crucial role in query design and makes it possible to predict the challenges that the queries impose on SPARQL engines. This, in turn, facilitates the interpretation of benchmark results.

The key contributions of this paper are the following.

- We present SP^{2}Bench, a comprehensive benchmark for the SPARQL query language, comprising a data generator and a collection of 17 benchmark queries.
- Our generator supports the creation of arbitrarily large DBLP documents in RDF format, reflecting key characteristics and social-world relations found in the original DBLP database. The generated documents cover various RDF constructs, such as blank nodes and containers.
- The benchmark queries have been carefully designed to test a variety of operator constellations, data access patterns, and optimization strategies. In the exhaustive discussion of these queries we also highlight the specific challenges they impose on SPARQL engines.
- As a proof of concept, we apply SP^{2}Bench to selected SPARQL engines and discuss their strengths and weaknesses that follow from the benchmark results. This analysis confirms that our benchmark is well-suited to identify deficiencies in SPARQL implementations.
- We finally propose performance metrics that capture different aspects of the evaluation process.

Outline. We next discuss related work and design decisions in Section II. The analysis of DBLP in Section III forms the basis for our data generator in Section IV. Section V gives an introduction to SPARQL and describes the benchmark queries. The experiments in Section VI comprise a short evaluation of our generator and benchmark results for existing SPARQL engines. We conclude with some final remarks in Section VII.

II. Benchmark Design Decisions

Benchmarking. The Benchmark Handbook [23] provides a summary of important database benchmarks. Probably the most "complete" benchmark suite for relational systems is TPC[2], which defines performance and correctness benchmarks for a large variety of scenarios. There also exists a broad range of benchmarks for other data models, such as object-oriented databases (e.g., OO7 [24]) and XML (e.g., XMark [20]).

Coming along with its growing importance, different benchmarks for RDF have been developed. The Lehigh University Benchmark [18] (LUBM) was designed with focus on inference and reasoning capabilities of RDF engines. However, the SPARQL specification [3] disregards the semantics of RDF and RDFS [25], [26], i.e. does not involve automated reasoning on top of RDFS constructs such as subclass and subproperty relations. With this regard, LUBM does not constitute an adequate scenario for SPARQL performance evaluation. This is underlined by the fact that central SPARQL operators, such as UNION and OPTIONAL, are not addressed in LUBM.

The Barton Library benchmark [19] queries implement a user browsing session through the RDF Barton online catalog. By design, the benchmark is application-oriented. All queries are encoded in SQL, assuming that the RDF data is stored in a relational DB. Due to missing language support for aggregation, most queries cannot be translated into SPARQL. On the other hand, central SPARQL features like left outer joins (the relational equivalent of SPARQL operator OPTIONAL) and solution modifiers are missing. In summary, the benchmark offers only limited support for testing native SPARQL engines.

The application-oriented Berlin SPARQL Benchmark [27] (BSBM) tests the performance of SPARQL engines in a prototypical e-commerce scenario. BSBM is use-case driven and does not particularly address language-specific issues. With its focus, it is supplementary to the SP^{2}Bench framework.

The RDF(S) data model benchmark in [28] focuses on structural properties of RDF Schemas. In [29] graph features of RDF Schemas are studied, showing that they typically exhibit power law distributions which constitute a valuable basis for synthetic schema generation. With their focus on schemas, both [28] and [29] are complementary to our work.

A synthetic data generation approach for OWL based on test data is described in [30]. There, the focus is on rapidly generating large data sets from representative data of a fixed domain. Our data generation approach is more fine-grained, as we analyze the development of entities (e.g. articles) over time and reflect many characteristics found in social communities.

Design Principles. In the Benchmark Handbook [23], four key requirements for domain specific benchmarks are postulated, i.e. it should be (1) *relevant*, thus testing typical operations within the specific domain, (2) *portable*, i.e. should be executable on different platforms, (3) *scalable*, e.g. it should be possible to run the benchmark on both small and very large data sets, and last but not least (4) it must be *understandable*.

[2]See http://www.tpc.org.

For a language-specific benchmark, the relevance requirement (1) suggests that queries implement realistic requests on top of the data. Thereby, the benchmark should not focus on correctness verification, but on common operator constellations that impose particular challenges. For instance, two SP^{2}Bench queries test negation, which (under closed-world assumption) can be expressed in SPARQL through a combination of operators OPTIONAL, FILTER, and BOUND.

Requirements (2) portability and (3) scalability bring along technical challenges concerning the implementation of the data generator. In response, our data generator is deterministic, platform independent, and accurate w.r.t. the desired size of generated documents. Moreover, it is very efficient and gets by with a constant amount of main memory, and hence supports the generation of arbitrarily large RDF documents.

From the viewpoint of engine developers, a benchmark should give hints on deficiencies in design and implementation. This is where (4) understandability comes into play, i.e. it is important to keep queries simple and understandable. At the same time, they should leave room for diverse optimizations. In this regard, the queries are designed in such a way that they are amenable to a wide range of optimization strategies.

DBLP. We settled SP^{2}Bench in the DBLP [21] scenario. The DBLP database contains bibliographic information about the field of Computer Science and, particularly, databases.

In the context of semi-structured data one often distinguishes between data- and document-centric scenarios. Document-centric design typically involves large amounts of free-form text, while data-centric documents are more structured and usually processed by machines rather than humans. RDF has been specifically designed for encoding information in a machine-readable way, so it basically follows the data-centric approach. DBLP, which contains structured data and little free text, constitutes such a data-centric scenario.

As discussed in the Introduction, our generator mirrors vital real-world distributions found in the original DBLP data. This constitutes an improvement over existing generators that create purely synthetic data, in particular in the context of a language-specific benchmark. Ultimately, our generator might also be useful in other contexts, whenever large RDF test data is required. We point out that the DBLP-to-RDF translation of the original DBLP data in [31] provides only a fixed amount of data and, for this reason, is not sufficient for our purpose.

We finally mention that sampling down large, existing data sets such as U.S. Census[3] (about 1 billion triples) might be another reasonable option to obtain data with real-world characteristics. The disadvantage, however, is that sampling might destroy more complex distributions in the data, thus leading to unnatural and "corrupted" RDF graphs. In contrast, our decision to build a data generator from scratch allows us to customize the structure of the RDF data, which is in line with the idea of a comprehensive, language-specific benchmark. This way, we easily obtain documents that contain a rich set of RDF constructs, such as blank nodes or containers.

[3]http://www.rdfabout.com/demo/census/

III. THE DBLP DATA SET

The subsequent study of DBLP lays the foundations for our data generator. The analysis of frequency distributions in scientific production was introduced in [32], and characteristics of DBLP have been investigated in [22]. The latter work studies a subset of DBLP, restricted to publications in database venues. It is shown that this subset reflects vital social relations and forms a small world on its own. Although this analysis forms valuable groundwork, our approach is much more pragmatic, as we approximate distributions by concrete functions.

We use function families that naturally reflect the scenarios, e.g. logistics curves for modeling limited growth or power equations for power law distributions. All approximations have been done with the *ZunZun* data modeling tool (http://www.zunzun.com) and the *gnuplot* curve fitting module (http://www.gnuplot.info), whereas data extraction from the DBLP XML document was realized with the MonetDB/XQuery processor (http://monetdb.cwi.nl/XQuery/).

An important objective of this section is to provide insights into key characteristics of DBLP. We analyze the structure of DBLP entities and work out a variety of interesting correlations and distributions, gaining insights that establish a deep understanding of the benchmark queries and their specific challenges. As an example, $Q3a$, $Q3b$, and $Q3c$ (see Appendix) look similar, but pose different challenges based on the probability distribution of article properties; $Q7$, on the other hand, heavily depends on the DBLP citation system.

While the generated data is very similar to the original DBLP data for years up to the present, we can of course give no guarantees that they go hand in hand for future years. However, generated future data follows reasonable and well-known social-world distributions. The queries are built on top of these distributions, thus being realistic, predictable and understandable. For instance, we query the citation system, which is mirrored by our generator, but ignore the distribution of article release months, as this property is not mimicked.

In the rest of this section we study key characteristics of DBLP and introduce the function families that we use to model them. The interested reader will find more details and concrete instances of these functions in our Technical Report [33].

A. Structure of Document Classes

Our starting point for the discussion is the DBLP DTD and the February 25, 2008 version of DBLP. An extract of the DTD is provided in Figure 1. The dblp element defines eight child entities, namely ARTICLE, INPROCEEDINGS, ..., and WWW resources. We call these entities *document classes*, and instances thereof *documents*. Furthermore, we distinguish between PROCEEDINGS documents, called *conferences*, and instances of the remaining classes, called *publications*.

The DTD allows 22 possible child tags such as author or url for each document class. They *describe* documents, and we call them *attributes*. Documents might be described by arbitrary, even repeated, attribute combinations, e.g. an article might have several authors. In practice, however, only a subset of all document class/attribute combinations occurs. For

```
<!ELEMENT dblp
  (article|inproceedings|proceedings|book|
   incollection|phdthesis|masterthesis|www)*>
<!ENTITY % field
  "author|editor|title|booktitle|pages|year|address|
   journal|volume|number|month|url|ee|cdrom|cite|
   publisher|note|crossref|isbn|series|school|chapter">
<!ELEMENT article (%field;)*>...<!ELEMENT www (%field;)*>
```

Fig. 1. Extract of the DBLP DTD

instance, attribute pages is never associated with WWW, but typically with ARTICLE documents. In Table I we show, for selected document class/attribute pairs, the probability that the attribute describes a document of this class. For instance, about 92.61% of all ARTICLES are described by attribute pages.

This probability distribution constitutes the basis for generating instances of the individual document classes. Note that we simplify and ignore conditional probabilities among attributes. We will elaborate on this decision in Section VII.

Repeated Attributes. A study of DBLP reveals that, in practice, only few attributes occur repeatedly within single documents. For the majority of them, the number of repeated occurrences is diminishing, so we restrict ourselves on the most frequent *repeated attributes* cite, editor, and author.

Figure 2(a) exemplifies our analysis for attribute cite. For the set of documents with at least one cite attribute, we plot the probability (y-axis) that the attribute occurs exactly n times (x-axis). Note that, according to Table I only a small fraction of documents are described by attribute cite, e.g. 4.8% of all ARTICLE documents (arguably, in a complete scenario this value should be close to 100%). For this reason, in Figure 2(a) we exclude documents without outgoing citation; though, in order to mirror the original distribution when assigning citations, we first use the attribute probability distribution (Table I) to estimate the number of documents with at least one citation and afterwards apply the distribution in Figure 2(a).

Based on experiments with different function families, we decided to use bell-shaped Gaussian curves for the approximation of the citation distribution. These curves are typically used to model normal distributions. Strictly speaking, our data is not normally distributed (due to the left limit x=1), however, these curves nicely fit the data for $x \geq 1$ (cf. Figure 2(a)). Gaussian curves are described by functions

$$p_{gauss}^{(\mu,\sigma)}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-0.5(\frac{x-\mu}{\sigma})^2},$$

where $\mu \in \mathbb{R}$ fixes the x-position of the peak and $\sigma \in \mathbb{R}_{>0}$ specifies the statistical spread. The approximation for the cite distribution is $d_{cite}(x) \stackrel{def}{:=} p_{gauss}^{(16.82,10.07)}(x)$. Analogously, for the editor distribution we set $d_{editor}(x) \stackrel{def}{:=} p_{gauss}^{(2.15,1.18)}(x)$.

The approximation function for repeated author attributes bases on a Gaussian curve, too. However, we observed that the average number of authors per publication has increased over the years. In [22] this is explained by the increasing pressure to publish and the proliferation of new communication platforms. In order to mimic this property, we model parameters μ and σ as functions over time, which yield higher values for later years. We refer the interested reader to our Technical Report [33] for a more detailed discussion.

TABLE I

PROBABILITY DISTRIBUTION FOR SELECTED ATTRIBUTES

	Article	Inproc.	Proc.	Book	Incoll.	WWW
author	0.9895	0.9970	0.0001	0.8937	0.8459	0.9973
cite	0.0048	0.0104	0.0001	0.0079	0.0047	0.0000
editor	0.0000	0.0000	0.7992	0.1040	0.0000	0.0004
isbn	0.0000	0.0000	0.8592	0.9294	0.0073	0.0000
journal	0.9994	0.0000	0.0004	0.0000	0.0000	0.0000
month	0.0065	0.0000	0.0001	0.0008	0.0000	0.0000
pages	0.9261	0.9489	0.0000	0.0000	0.6849	0.0000
title	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

B. Key Characteristics of DBLP

We noticed that DBLP contains only few and incomplete information in its early years, but also found anomalies in the final years, mostly in form of lowered growth rates. It might be that, in the coming years, some more entries for these years will be added belatedly (i.e., data might still be missing), so we base our discussion on DBLP data in-between 1960-2005.

Figure 2(b) plots the number of PROCEEDINGS, JOURNAL, INPROCEEDINGS, and ARTICLE documents as a function of time. The y-axis is in log scale. Note that JOURNAL is not an explicit document class, but implicitly defined by the journal attribute of ARTICLES. Inproceedings and articles are closely coupled to the proceedings and journals, e.g. there is an average of about 50-60 inproceedings per proceeding.

We observe exponential growth in all cases, with decreasing growth rates of JOURNAL and ARTICLE documents in the final years. This suggests a limited growth scenario, which is typically modeled by logistic curves, i.e. functions with a lower and an upper asymptote that either continuously increase or decrease for increasing x. We use curves of the form

$$f_{logistic}(x) = \frac{a}{1+be^{-cx}},$$

where $a, b, c \in \mathbb{R}_{>0}$. Parameter a constitutes the upper asymptote and the x-axis forms the lower asymptote. The curve is "caught" in-between its asymptotes and increases continuously, i.e. is S-shaped. The logistic curve for the number of JOURNAL documents in Figure 2(b) is defined as

$$f_{journal}(yr) \stackrel{def}{:=} \frac{740.43}{1+426.28e^{-0.12(yr-1950)}}.$$

We omit the concrete approximations for the other classes.

C. Author and Editor Characteristics

Using the yearly counts of the document classes, the probability distribution of attribute author (Table I), and the (yearly) average number of authors per paper (Section III-A) we can estimate the *total number of authors* per year, i.e. the number of author attributes in the data. We also approximate the number of distinct persons that appear as authors (called *distinct authors*) and the number of *new authors* in a given year, i.e. persons that have not published before.

Publications. In Figure 2(c) we plot (in log-log scale), for selected year and publication count x, the number of authors with exactly x publications in this year. We observe a typical power law distribution, i.e. only a couple of authors have a large number of publications, while lots of authors have few publications. Power law distributions are captured by functions

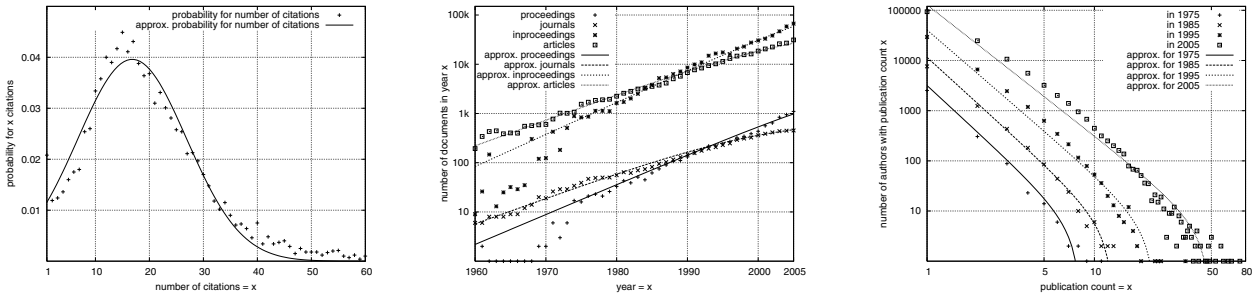

Fig. 2. (a) Distribution of citations, (b) Number of document class instances, and (c) Publication counts of authors

of the form $f_{powerlaw}(x) = ax^k + b$, with constants $a \in \mathbb{R}_{>0}$, exponent $k \in \mathbb{R}_{<0}$, and $b \in \mathbb{R}$. Parameter a affects the x-axis intercept, exponent k defines the gradient, and b constitutes a shift in y-direction. For the given parameter restriction, the functions decrease steadily for increasing $x \geq 0$.

Figure 2(c) indicates that, throughout the years, the curves move upwards. This means that the publication count of the leading author(s) has steadily increased over the last 30 years, and also reflects an increasing number of authors. We estimate the number of authors with x publications in year yr as

$$f_{awp}(x, yr) \stackrel{def}{:=} 1.50 f_{publ}(yr) x^{-f'_{awp}(yr)} - 5, \text{ where}$$
$$f'_{awp}(yr) \stackrel{def}{:=} \frac{-0.60}{1+216223e^{-0.20(yr-1936)}} + 3.08, \text{ and}$$

$f_{publ}(yr)$ returns the total number of publications in yr.

Coauthors. We also factor in relations between authors its coauthors. For space limitations, we omit the details.

Editors. We associate editors with authors by investigating the editors' number of publications in earlier venues. We observe that editors typically have published before, i.e. are known in the community. The concrete formula is omitted.

D. Citations

In Section III-A we have studied repeated occurrences of attribute cite, i.e. outgoing citations. Concerning the *incoming* citations (i.e. the count of incoming references for papers), we observed a characteristic power law distribution: Most papers have few incoming citations, while only few are cited often. Hence, the resulting formula is structurally similar to the distribution of publications among authors.

We found that, in the original data, there are much more outgoing than incoming citations, because DBLP contains many untargeted citations (in form of empty cite tags). Given that only few papers have outgoing citations (cf. Table I), we conclude that the DBLP citation system is very incomplete. The generated data reflects this incompleteness.

IV. Data Generation

The RDF Data Model. Figure 3(b) shows a sample RDF graph. Each edge in the graph encodes a single knowledge fact, e.g. the arc from node *Proceeding1* to node *_:John_Due* represents the RDF triple (*Proceeding1*,*swrc:editor*,*_:John_Due*), meaning that *John Due* is an editor of *Proceeding1*. Dashed lines denote edges with label rdf:type and *sc* is used as an abbreviation for the subclass specification rdfs:subClassOf.

RDF graphs may contain three types of elements: *URIs* (Uniform Resource Identifiers) are strings that uniquely identify abstract or physical resources, such as conferences or journals. *Blank nodes* have an existential character, are locally unique, and may be used in place of URIs. URIs and blank nodes are modeled by ellipses; we distinguish blank nodes by the prefix "_:". *Literals* constitute (possibly typed) values, such as strings or integers. They are represented by quoted strings.

The RDF standard [1] introduces a fixed base vocabulary, e.g. rdf:type for type specifications, or rdf:Bag to model bag containers. RDFS [25] extends RDF by meta-level vocabulary with predefined semantics, such as rdfs:subClassOf and rdf:subPropertyOf to specify subclass and subproperty relations. On top of RDF and RDFS vocabulary, user-defined domain-specific vocabulary collections can be developed. Our DBLP scheme builds upon such domain-specific vocabulary.

The DBLP RDF Scheme. Our RDF scheme basically follows the RDF encoding approach presented in [31], which provides an XML-to-RDF mapping of the original DBLP data set. However, as we want to generate arbitrarily-sized documents we provide lists of first and last names, publishers, and random words to our data generator. Conference and journal names are always of the form "*Conference $i ($year)*" and "*Journal $i ($year)*", where $\$i$ is a unique conference (respectively journal) number in the year *$year*.

We follow the approach from [31] and borrow vocabulary from FOAF, SWRC, and Dublin Core (DC) to describe persons and scientific resources.[4] Additionally, we introduce a namespace bench, which defines DBLP-specific document classes, such as `bench:Book` and `bench:Article`. Figure 3(a) shows the translation of attributes to RDF properties. For each attribute, we also list its range restriction, i.e. the type of elements it refers to. For instance, attribute author is mapped to dc:creator and references objects of type `foaf:Person`.

We want to test our queries on a rich set of RDF constructs, so we model authors as blank nodes (instead of URIs) of the form "*_:givenname_lastname*". All outgoing references of a publication are grouped together using the RDF standard container class `rdf:Bag`. We also enriched a small fraction of Article and Inproceedings documents with the new property bench:abstract (about 1%, keeping the modification low). Abstracts are not contained in the original DBLP data;

[4] See http://www.foaf-project.org/, http://ontoware.org/projects/swrc/, and http://dublincore.org/.

attribute	mapped to prop.	refers to
address	swrc:address	xsd:string
author	dc:creator	foaf:Person
booktitle	bench:booktitle	xsd:string
cdrom	bench:cdrom	xsd:string
chapter	swrc:chapter	xsd:integer
cite	dcterms:references	foaf:Document
crossref	dcterms:partOf	foaf:Document
editor	swrc:editor	foaf:Person
ee	rdfs:seeAlso	xsd:string
isbn	swrc:isbn	xsd:string
journal	swrc:journal	bench:Journal
month	swrc:month	xsd:integer
note	bench:note	xsd:string
number	swrc:number	xsd:integer
page	swrc:pages	xsd:string
publisher	dc:publisher	xsd:string
school	dc:publisher	xsd:string
series	swrc:series	xsd:integer
title	dc:title	xsd:string
url	foaf:homepage	xsd:string
volume	swrc:volume	xsd:integer
year	dcterms:issued	xsd:integer

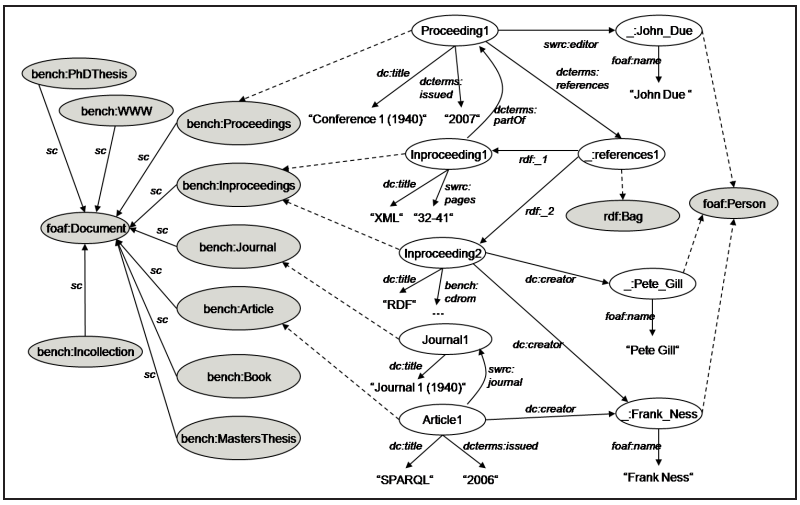

Fig. 3. (a) Translation of attributes, and (b) DBLP sample instance in RDF format

they contribute comparably large strings (we use a Gaussian distribution with $\mu = 150$ expected words and $\sigma = 30$).

Figure 3(b) shows a sample DBLP instance. On the logical level, we distinguish between *schema* layer and *instance* layer (gray vs. white). Reference lists are blank nodes of type `rdf:Bag` (see node *_:references1*). Authors and editors are typed with `foaf:Person`. The superclass `foaf:Document` splits up into the individual document classes `bench:Journal`, `bench:Article`, and so on. The sample graph defines three persons, one proceeding, two inproceedings, one journal, and one article. For readability reasons, we plot only selected predicates. As also illustrated, property dcterms:partOf links inproceedings and proceedings together, while swrc:journal connects articles to their journals.

In order to provide an entry point for queries that access authors and to define a person with fixed and known characteristics, we created a special author, named after the famous mathematician Paul Erdös. Per year, we assign 10 publications and 2 editor activities to this prominent person, starting from year 1940 up to 1996. For the ease of access, Paul Erdös is modeled by a fixed URI. As an example query consider $Q8$, which extracts all persons with *Erdös Number*[5] 1 or 2.

Data Generation. Our data generator takes into account all relationships and characteristics that have been studied in Section III. As shown in Figure 4, we simulate year by year and generate data according to the structural constraints in a carefully selected order. Data generation is incremental, i.e. small documents are always contained in larger documents.

The generator is implemented in C++ and offers two parameters, to fix either a triple count limit or the year up to which data will be generated. When the triple count limit is set, we make sure to end up in a "consistent" state, e.g. when proceedings are written, their conference also will be included.

The generation process is simulation-based. Among others, this means that we assign life times to authors, and individually estimate their future behavior, taking into account global

[5]See http://www.oakland.edu/enp/.

foreach year:
  calculate counts for and generate document classes;
  calculate nr of total, new, distinct, and retiring authors;
  choose publishing authors;
  assign nr of new publications, nr of coauthors, and nr of distinct coauthors to publishing authors;
  // s.t. constraints for nr of publications/author hold
  assign from publishing authors to papers;
  // satisfying authors per paper/co authors constraints
  choose editors and assign editors to papers;
  // s.t. constraints for nr of publications/editors hold
  generate outgoing citations;
  assign expected incoming/outgoing citations to papers;
  write output until done or until output limit reached;
  // permanently keeping output consistent

Fig. 4. Data generation algorithm

publication and coauthor characteristics, as well as the fraction of distinct and new authors (cf. Section III-C).

All random functions (which, for example, are used to assign the attributes according to Table I) base on a fixed seed, which makes data generation deterministic. Moreover, the implementation is platform-independent, so we ensure that experimental results from different machines are comparable.

V. Benchmark Queries

The SPARQL Query Language. SPARQL is a declarative language and bases upon a powerful graph matching facility, allowing to match query subexpressions against the RDF input graph. The very basic SPARQL constructs are triple patterns $(subject, predicate, object)$, where variables might be used in place of fixed values for each of the three components. In evaluating SPARQL, these patterns are mapped against one or more input graphs, thereby binding variables to matching nodes or edges in the graph(s). Since we are primarily interested in database aspects, such as operator constellations and access patterns, we focus on queries that access a single graph.

The SPARQL standard [3] defines four query forms. SELECT queries retrieve all possible variable-to-graph mappings, while ASK queries return *yes* if at least one such mapping

TABLE II

SELECTED PROPERTIES OF THE BENCHMARK QUERIES; SHORTCUTS ARE INDICATED BY BOLD FONT

	Query	1	2	3abc	4	5ab	6	7	8	9	10	11	12c
1	Operators: **A**ND,**F**ILTER,**U**NION,**O**PTIONAL	A	A,O	A,F	A,F	A,F	A,F,O	A,F,O	A,F,U	A,U	-	-	-
2	Modifiers: **D**ISTINCT,**L**IMIT,**Of**FSET,**O**RDER **b**Y	-	Ob	-	D	D		D	D	D	-	L,Ob,Of	-
4	Filter Pushing Possible?	-	-	✓	-	✓/-	✓	✓	✓	-	-	-	-
5	Reusing of Graph Patternss Possible?	-	-	-	✓	-	✓	✓	✓	✓	-	-	-
6	Data Access: **B**LANK NODES,**L**ITERALS,**URI**S, **La**RGE LITERALS,**C**ONTAINERS	L,U	L,U,La	L,U	B,L,U	B,L,U	B,L,U	L,U,C	B,L,U	B,L,U	U	L,U	U

exists, and *no* otherwise. DESCRIBE extracts additional information related to the result mappings (e.g. adjacent nodes), while CONSTRUCT transforms the result mappings into an RDF graph. Most appropriate for our purpose is SELECT, which best reflects SPARQL core evaluation. The interesting challenge in ASK queries is to efficiently locate a witness. CONSTRUCT and DESCRIBE basically build upon the core evaluation of SELECT, i.e. transform its result in a post-processing step, which is not very challenging from a database perspective. Therefore, we focus on SELECT and ASK queries.

The most important SPARQL operator is AND (denoted as "."). An expressions A AND B is evaluated by joining the result mappings of A and B on their shared variables [4]. Let us consider $Q1$ from the Appendix, which defines three triple patterns interconnected through AND. When first evaluating the patterns individually, variable *?journal* is bound to nodes with (1) edge rdf:type pointing to the URI `bench:Journal`, (2) edge dc:title pointing to the Literal "*Journal 1 (1940)*" of type string, and (3) edge dcterms:issued, respectively. The next step is to join the individual mapping sets on variable *?journal*. The result then contains all mappings from *?journal* to nodes that satisfy all three patterns. Finally SELECT projects for variable *?yr*, which has been bound in the third pattern.

Other SPARQL operators are UNION, OPTIONAL, and FILTER, akin to relational unions, left outer joins, and selections, respectively. For space limitations, we omit an explanation of these constructs and refer the reader to the official SPARQL specification [3]. Beyond all these operators, SPARQL provides functions to be used in FILTER expressions, e.g. for regular expression testing. We expect these functions to only marginally affect engine performance, since their implementation is mostly straightforward (or might be realized through efficient libraries). They are unlikely to bring insights into the core evaluation capabilities, so we omit them intentionally. This decision also facilitates benchmarking of research prototypes, which typically do not implement the full standard.

Our queries also cover SPARQL solution modifiers (such as DISTINCT, ORDER BY), as they might affect the choice of the execution plan. We point out that the previous discussion captures virtually all key features of the SPARQL query language. In particular, SPARQL (currently) does not support aggregation, nesting, recursion, and inferencing.

SPARQL Characteristics. Rows 1 and 2 in Table II survey the operators used in the benchmark queries ($Q12a$ and $Q12b$ share the characteristics of $Q5a$ and $Q8$, resp., and are not shown). As shown, we cover various operator constellations, combined with different solution modifiers combinations.

One characteristic SPARQL feature is operator OPTIONAL. An expression A OPTIONAL B joins result mappings from A with mappings from B and retains all mappings from A for which no join partner in B is found, leaving variables that occur only in B unbound. By combining OPTIONAL with FILTER and BOUND (which checks for a variable being bound), *closed world negation* can be encoded. Many interesting queries involve such an encoding (cf. $Q6$ and $Q7$).

SPARQL operates on graph-structured data, thus engines should perform well on different kinds of graph patterns. By now there exist only few real world SPARQL scenarios and a meaningful analysis of graph patterns that frequently arise in practice cannot yet be performed. In the absence of this possibility, we distinguish between *long path chains*, i.e. nodes linked to each other via a long path, *bushy patterns*, i.e. single nodes linked to several other nodes, and *combinations* thereof. Clearly, a precise definition of "long" and "bushy" is not possible, so we designed meaningful queries with *comparably* long chains (i.e. $Q4$, $Q6$) and bushy patterns (i.e. $Q2$). They contribute to the broad variety of characteristics we cover.

SPARQL Optimization. Our objective is to design queries that are amenable to a variety of SPARQL optimization approaches. To this end, we discuss possible optimization techniques. One promising approach is the *reordering of triple patterns* based on selectivity estimations [5], akin to relational join reordering. Closely related is FILTER *pushing*, which aims at an early evaluation of filter conditions, similar to projection pushing in Relational Algebra. Both techniques might speed up evaluation by decreasing the size of intermediate results. Join reordering might apply to most of our queries. Row 4 in Table II lists queries that support FILTER pushing.

Another idea is to *reuse evaluation results of triple patterns* (or even combinations thereof). This is possible wherever the same pattern is used multiple times (for instance, the first two triple patterns in $Q4$ select exactly the same nodes). We survey the applicability of this technique in Table II, row 5.

RDF Characteristics and Storage. Recalling that persons are modeled as blank nodes, all queries that deal with persons access blank nodes. Query $Q7$ operates on top of the RDF bag container for reference lists, while $Q2$ accesses the large abstract literals ($Q2$). Row 6 in Table II provides a survey.

A comparison of RDF storage strategies is provided in [12]. Storage scheme and indices imply a selection of efficient *data access paths*. Our queries impose varying challenges to the storage scheme, e.g. test data access through the subject, predicate, or object. Typically, predicates are fixed and subjects or objects vary, but we also test uncommon access patterns.

We will resume this discussion when discussing $Q9$ and $Q10$.

Benchmark Queries. Our queries also vary in general characteristics like *selectivity*, *query and output size*, and *different types of joins*. We will point come back to these issues in the subsequent discussion of the benchmark queries. In the following, we distinguish between *in-memory* engines, which load documents from file and process queries in main memory, and *native* engines, which rely on a physical database. When discussing evaluation strategies for native engines, we assume that the document has been loaded into the database before.

While, in this paper we focus on the SPARQL versions of the SP^{2}Bench queries, we also point the interested reader to the SQL translations of these queries. They are available online at our project page and a discussion can be found in [34].

Q1. *Return the year of publication of "Journal 1 (1940)".*

This simple query returns exactly one result (for arbitrarily large documents). Native engines might use index lookups in order to answer this query in (almost) constant time, i.e. execution time should be independent from document size.

Q2. *Extract all inproceedings with properties* dc:creator, bench:booktitle, dcterms:issued, dcterms:partOf, rdfs:seeAlso, dc:title, swrc:pages, foaf:homepage, *and optionally* bench:abstract, *including their values.*

This query implements a bushy graph pattern. It contains a single OPTIONAL expression, and accesses large strings (i.e. the abstracts). Result size grows with database size, and a final result ordering is necessary due to operator ORDER BY. Both native and in-memory engines should scale linearly to document size.

Q3abc. *Select all articles with property (a)* swrc:pages, *(b)* swrc:month, *or (c)* swrc:isbn.

This query tests FILTER expressions with varying selectivity. Following Table I, the filter in $Q3a$ retains about 92.61% of all articles. While data access through an unclustered index would be inefficient here, it might pay off for $Q3b$, which retains only about 0.65% of the articles. The filter in $Q3c$ is never satisfied, since no articles are described by swrc:isbn. Statistics might be used to answer $Q3c$ in constant time, even without data access.

Q4. *Select all distinct pairs of article author names for authors that have published in the same journal.*

$Q4$ contains a comparably long graph chain, i.e. variables *?name1* and *?name2* are linked through articles that (different) authors have published in the same journal. It is obvious that the query computes very large result sets. Instead of evaluating the outer pattern block and applying the FILTER afterwards, engines should embed the FILTER expression into this computation. Ultimately, the DISTINCT modifier further complicates the query. We expect superlinear behavior for both native and in-memory engines.

Q5ab. *Return the names of all persons that occur as author of at least one inproceeding and at least one article.*

Queries $Q5a$ and $Q5b$ test different variants of joins. $Q5a$ implements an implicit join on author names, which is encoded in the FILTER condition, while $Q5b$ explicitly joins the authors on variable *?name*. While different in general, the one-to-one mapping between authors and their names (i.e. author names constitute primary keys) in our scenario implies equivalence of the queries. In [14], semantic optimization for RDF has been proposed. Such an approach might detect this equivalence and might always execute the more efficient representation.

Q6. *Return, for each year, the set of all publications authored by persons that have not published in years before.*

$Q6$ implements (closed world) negation, expressed through a combination of operators OPTIONAL, FILTER, and BOUND. The idea of the construction is that the block outside the OPTIONAL expression computes all publications, while the inner one constitutes earlier publications from authors that appear outside. The outer FILTER expression then retains publications for which *?author2* is unbound, i.e. exactly the publications of those authors that have not published in earlier years.

Q7. *Return the titles of all papers that have been cited at least once, but not by any paper that has not been cited itself.*

This query implements double negation. We expect only few results, due to the incomplete citation system (cf. Section III-D). Though, double negation makes the query very challenging. Engines might also reuse graph pattern results here, e.g. the block ?class[i] rdf:type foaf:Document. ?doc[i] rdf:type ?class[i]. occurs three times, for empty $[i]$, $[i]$=3, and $[i]$=4.

Q8. *Compute authors that have published with Paul Erdös or with an author that has published with Paul Erdös.*

For this query, the evaluation of the second UNION part is "contained" in the evaluation of the first part. Again, graph pattern (or subexpression) results might be reused. It might also be promising to decompose the filter expressions and push down its components, in order to decrease the size of intermediate results.

Q9. *Return incoming and outgoing properties of persons.*

$Q9$ has been designed to test non-standard data access patterns. Naive implementations would compute the triple patterns in the UNION subexpressions separately, thus evaluating patterns where no component is bound. As an improvement, engines might start with the first triple in each UNION subexpression and use the resulting bindings for variable *?person* to evaluate the second one more efficiently. In this case, joins on the *?subject* and *?object* variable are necessary, in which only the *?subject* (resp. the *?object*) variables are bound. The query extracts schema information and result size is (at most) 4, Native engines might also use summary statistics about incoming/outgoing properties of *Person*-typed objects to answer this query in constant time without data access. In-memory engines must load the document and, in the best case, might scale linearly to document size.

Q10. *Return all subjects that stand in any relation to person "*Paul Erdös*". In our scenario the query can be reformulated as Return publications and venues in which "*Paul Erdös*" is involved either as author or as editor.*

$Q10$ implements an object bound-only access pattern. In contrast to $Q9$, statistics are not immediately useful, since the result includes subjects. Recall that "Paul Erdös" is active only between 1940 and 1996, so result size stabilizes for sufficiently large documents (cf. Table V). Native engines could exploit indices and might reach (almost) constant execution time.

Q11. *Return (up to) 10 electronic edition URLs starting from the 51^{th} publication, in lexicographical order.*

This query focuses on the combination of solution modifiers ORDER BY, LIMIT, and OFFSET. In-memory engines have to read, process, and sort electronic editions prior to processing LIMIT and OFFSET. In contrast, native engines might exploit indices to access only a fraction of all electronic editions and, as the result is limited to 10, could reach constant runtimes.

Q12. *(a) Return yes if a person is an author of at least one inproceeding and article; (b) Return yes if an author has published with Paul Erdös or with an author that has published with "*Paul Erdös*"; (c) Return yes if person "*John Q. Public*" exists.*

$Q12a$ and $Q12b$ share the properties of their SELECT counterparts $Q5a$ and $Q8$, respectively. Both return *yes* for sufficiently large documents. The challenge in evaluating ASK queries is to efficiently locate a witness, to break as soon as possible. To this end, engines might even adapt the query execution plan, e.g. it might be favorable to evaluate the second part of the UNION in $Q12b$ first. Both native and in-memory engines should answer these queries very fast, independent from document size.
$Q12c$ asks for a single triple that is not present in the database. With indices, native engines might reach constant time. In-memory engines must scan and load the whole document.

VI. EXPERIMENTS

All experiments were conducted under Linux ubuntu v7.10 gutsy, on top of an Intel Core2 Duo E6400 2.13GHz CPU and 3*GB* DDR2 667 MHz nonECC physical memory. We used a 250*GB* Hitachi P7K500 SATA-II hard drive with 8*MB* Cache. The Java engines were executed with JRE v1.6.0_04.

Data Generator. We measured generation times for documents of different sizes. Our generator is very efficient, e.g. creates one billion triples (about $100GB$ of data) in less than four hours. It scales almost linearly to document size and gets by with a constant main memory consumption, using (at most) $1.2GB$ RAM for arbitrarily-sized documents. We exemplarily generated RDF documents up to 3 billion triples.

In addition, we verified that the characteristics extracted in Section III are reflected in the generated data. Table III shows selected properties of generated documents up to $25M$ triples. We list the size of the output file, the year in which simulation ended, the number of total and distinct authors contained in the data (cf. Section III-C), and counts for the individual document types (cf. Section III-B). The superlinear growth of the authors is primarily caused by the increasing average number of authors per paper (cf. Section III-A). The growth rate of proceedings and inproceedings is also superlinear, whereas journals and articles increase sublinearly. This reflects the yearly document class counts in Figure 2(b). Like in the original DBLP data, in the early years several document classes (e.g. BOOK and WWW) are not yet contained.

Table V surveys the result sizes for the queries on documents up to $5M$ triples. We observe for example that the outcome of $Q3a$, $Q3b$, and $Q3c$ reflects the selectivities of their FILTER attributes swrc:pages, swrc:month, and swrc:isbn (cf. Table I and III). We will come back to the result size listing when discussing the benchmark results later in this section.

Metrics. We report on user time (`usr`), system time (`sys`), and the high watermark of resident memory (`rmem`). These values were extracted from the *proc* file system. Furthermore, we assess elapsed time (`tme`) through timers. Please note that experiments were carried out on a DuoCore CPU, where the linux kernel sums up `usr` and `sys` of the processor units. As a consequence, the sum `usr+sys` might be greater than `tme`.

In our Technical Report [33] we propose several performance metrics, including success rate reports, loading time, memory consumption, per-query performance, and arithmetic/geometric mean. They capture different aspects of evaluation and help to report on results in a standardized way.

TABLE III
CHARACTERISTICS OF GENERATED DOCUMENTS

#Triples	10k	50k	250k	1M	5M	25M
file size [MB]	1.0	5.1	26	106	533	2694
data up to	1955	1967	1979	1989	2001	2015
#Tot.Auth.	1.5k	6.8k	34.5k	151.0k	898.0k	5.4M
#Dist.Auth.	0.9k	4.1k	20.0k	82.1k	429.6k	2.1M
#Journals	25	104	439	1.4k	4.6k	11.7k
#Articles	916	4.0k	17.1k	56.9k	207.8k	642.8k
#Proc.	6	37	213	903	4.7k	24.4k
#Inproc.	169	1.4k	9.2k	43.5k	255.2k	1.5M
#Incoll.	18	56	173	442	1.4k	4.5k
#Books	0	0	39	356	973	1.7k
#Other	0	0	0	186	424	802

TABLE IV
SUCCESS RATES FOR QUERIES $Q4$, $Q5a$, $Q5b$, $Q6$, AND $Q7$, WHERE +:=SUCCESS, T:=TIMEOUT, M:=MEMORY EXHAUSTION, AND E:=ERROR

	ARQ	Sesame$_M$	Sesame$_{DB}$	Virtuoso
Query	4 5 6 7	4 5 6 7	4 5 6 7	4 5 6 7
250k	T++++	+T+T+	+T+TT	TT+E+
1M	TT+TT	+T+TT	+T+TT	TTTET
5M	TT+TT	TT+TT	MT+TT	TTTET
25M	TTTTT	MMTMM	TT+TT	(loading failure)

Benchmark Results. It is beyond the scope of this paper to provide an in-depth comparison of existing SPARQL engines. Rather, we want to give first insights into the state-of-the art and highlight deficiencies of engines based on the benchmark outcome. In this line, we are not primarily interested in concrete values (which, however, in general might be of great interest), but focus on the principal behavior of engines, using the metrics mentioned before. For space restrictions, we discuss only selected cases and refer the reader to our Technical Report for the complete benchmark results.

We conducted experiments for (1) the Java engine **ARQ** v2.2 on top of Jena 2.5.5, (2) the **Redland** RDF Processor v1.0.7 (written in C), using the Raptor Parser Toolkit v.1.4.16 and Rasqal Library v0.9.15, (3) **SDB**, which links ARQ to an SQL database back-end (i.e., we used mysql v5.0.34), (4) the Java implementation **Sesame** v2.2beta2, and finally (5) the OpenLink **Virtuoso** system v5.0.6 (written in C).[6]

For Sesame we tested two configurations: *Sesame*$_M$, which processes queries in memory, and *Sesame*$_{DB}$, which physically stores data in a database, using the *Mulgara* SAIL v1.3beta1 as backend. We thus distinguish between the in-memory engines (*ARQ*, *Sesame*$_M$) and native engines (*Redland*, *SBD*, *Sesame*$_{DB}$, *Virtuoso*). For native engines we created indices wherever possible (after loading the documents) and consider loading and execution time separately (we include index creation time in the loading times).

We performed three cold runs over all queries and docu-

[6]ARQ: http://jena.sourceforge.net/ARQ/, Redland: http://librdf.org/, SDB: http://jena.sourceforge.net/SDB/, Sesame: http://www.openrdf.org/, Virtuoso: http://www.openlinksw.com/virtuoso/

TABLE V
NUMBER OF QUERY RESULTS ON DOCUMENTS UP TO 5 MILLION TRIPLES

Query	$Q1$	$Q2$	$Q3a$	$Q3b$	$Q3c$	$Q4$	$Q5a$	$Q5b$	$Q6$	$Q7$	$Q8$	$Q9$	$Q10$	$Q11$
$10k$	1	147	846	9	0	23226	155	155	229	0	184	4	166	10
$50k$	1	965	3647	25	0	104746	1085	1085	1769	2	264	4	307	10
$250k$	1	6197	15853	127	0	542801	6904	6904	12093	62	332	4	452	10
$1M$	1	32770	52676	379	0	2586733	35241	35241	62795	292	400	4	572	10
$5M$	1	248738	192373	1317	0	18362955	210662	210662	417625	1200	493	4	656	10

ments of $10k$, $50k$, $250k$, $1M$, $5M$, and $25M$ triples, i.e. in-between each two runs we restarted the engines and cleared the database. We set a timeout of 30min (`tme`) per query and a memory limit of 2.6GB, either using *ulimit* or restricting the JVM (for higher limits, the initialization of the JVM failed). Negative and positive variation of the average (over the runs) was $< 2\%$ in almost all cases, so we omit error bars. For *SDB* and *Virtuoso*, which follow a client-server architecture, we monitored both processes and sum up these values.

We verified all results by comparing the outputs, observing that *SDB* and *Redland* returned wrong results for a couple of queries, so we restrict ourselves on the discussion of the remaining four engines. Table IV shows the success rates. All queries that are not listed succeeded, except for *ARQ* and $Sesame_M$ on the $25M$ document (either due to timeout or memory exhaustion) and Virtuoso on $Q6$ (due to missing standard compliance). Hence, $Q4$, $Q5a$, $Q6$, and $Q7$ are the most challenging queries, where we observe many timeouts even for small documents. Note that we did not succeed in loading the $25M$ triples document into the *Virtuoso* database.

Main Memory. For the in-memory engines we observe that the high watermark of main memory consumption during query evaluation increases sublinearly to document size. For instance, for ARQ we measured an average (over all runs and queries) of 85*MB* on $10k$, 166*MB* on $50k$, 318*MB* on $250k$, 526*MB* on $1M$, and 1.3*GB* on $5M$ triples. Somewhat surprisingly, also the memory consumption of the native engines *Virtuoso* and $Sesame_{DB}$ increased with document size.

Arithmetic and Geometric Mean. For the in-memory engines we found that $Sesame_M$ is superior to *ARQ* regarding both means. For instance, the arithmetic (T_a) and geometric (T_g) mean on the $1M$ document over all queries (we penalized failure queries with $3600s$) are $T_a^{SesM} = 683.16s$, $T_g^{SesM} = 106.84s$, $T_a^{ARQ} = 901.73s$, and $T_g^{ARQ} = 179.42s$.

The means for the native engines on $1M$ triples are: $T_a^{SesDB} = 653.17s$, $T_g^{SesDB} = 10.17s$, $T_a^{Virt} = 850.06s$, and $T_g^{Virt} = 3.03s$. The arithmetic mean of $Sesame_{DB}$ is superior, which is mainly due to the fact that it failed only on 4 (vs. 5) queries. The geometric mean moderates the impact of these outliers. *Virtuoso* shows a better overall performance for the success queries, so its geometric mean is superior.

In-memory Engines. Figure 5 (top) plots selected results for in-memory engines. We start with $Q5a$ and $Q5b$. Although both queries yield the same result, the engines perform much better for the explicit join in $Q5b$. We may suspect that the implicit join in $Q5a$ is not recognized, i.e. that both engines compute the cartesian product and apply the filter afterwards.

$Q6$ and $Q7$ implement simple and double negation, respectively. Both engines show insufficient behavior. At the first glance, we might expect that $Q7$ (which involves double negation) is more complicated to evaluate, but we observe that $Sesame_M$ scales even worse for $Q6$. We identify two possible explanations. First, $Q7$ "negates" documents with incoming citations, but – according to Section III-D – only a small fraction of papers has incoming citations at all. In contrast, $Q6$ negates arbitrary documents, i.e. a much larger set. Another reasonable cause might be the non-equality filter subexpression `?yr2 < ?yr` inside the inner FILTER of $Q6$.

For ASK query $Q12a$ both engines scale linearly with document size. However, from Table V and the fact that our data generator is incremental and deterministic, we know that a "witness" is already contained in the first $10k$ triples of the document. It might be located even without reading the whole document, so the strategies of both engines are suboptimal.

Native Engines. The leftmost plot at the bottom of Figure 5 shows the loading times for the native engines $Sesame_{DB}$ and *Virtuoso*. Both engines scale well concerning `usr` and `sys`, essentially linear to document size. For $Sesame_{DB}$, however, `tme` grows superlinearly (e.g., loading of the $25M$ document is about ten times slower than loading of the $5M$ document). This trend might cause serious problems for larger documents.

The running times for $Q2$ increase superlinear for both engines (in particular for larger documents). This reflects the superlinear growth of inproceedings and the growing result size (cf. Tables III and V). What is interesting here is the significant difference between `usr+sys` and `tme` for *Virtuoso*, which indicates disproportional disk I/O. Since *Sesame* does not exhibit this peculiar behavior, it might be an interesting starting point for further optimizations in the *Virtuoso* engine.

Queries $Q3a$ and $Q3c$ have been designed to test the intelligent choice of indices in the context of FILTER expressions with varying selectivity. *Virtuoso* gets by with an economic consumption of `usr` and `sys` time for both queries, which suggests that it makes heavy use of indices. While this strategy pays off for $Q3c$, the elapsed time for $Q3a$ is unreasonably high and we observe that $Sesame_M$ scales better for this query.

$Q10$ extracts subjects and predicates that are associated with *Paul Erdös*. First recall that, for each year up to 1996, *Paul Erdös* has exactly 10 publications and occurs twice as editor (cf. Section IV). Both engines answer this query in about constant time, which is possible due to the upper result size bound (cf. Table V). Regarding `usr+sys`, *Virtuoso* is even more efficient: These times are diminishing in all cases. Hence, this query constitutes an example for desired engine behavior.

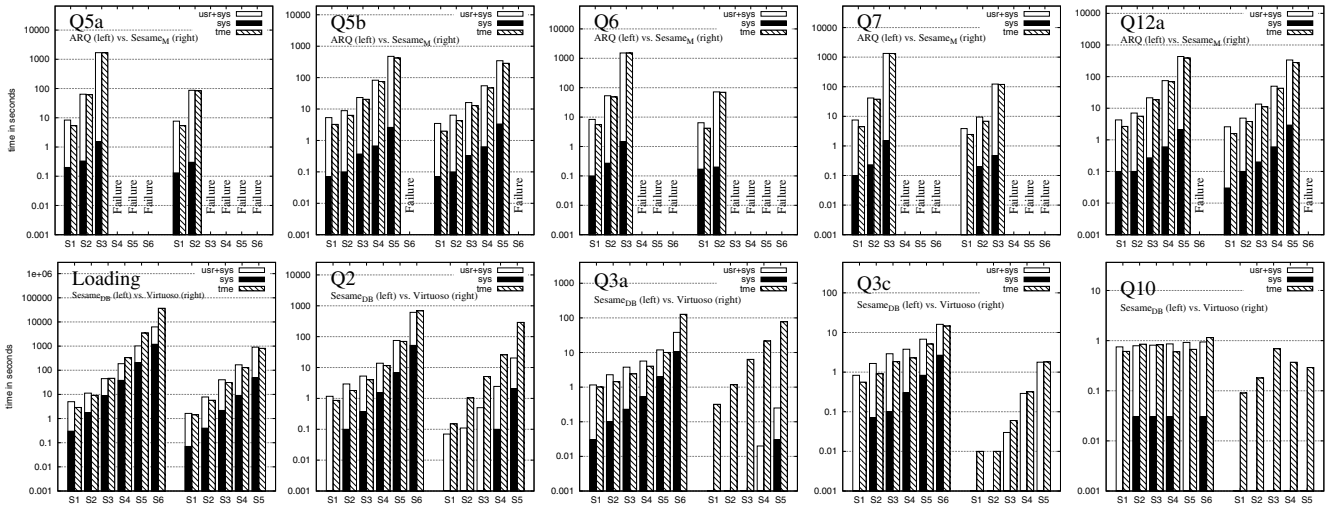

Fig. 5. Results for in-memory engines (top) and native engines (bottom) on S1=10k, S2=50k, S3=250k, S4=1M, S5=5M, and S6=25M triples

VII. Conclusion

We have presented the SP²Bench performance benchmark for SPARQL, which constitutes the first methodical approach for testing the performance of SPARQL engines w.r.t. different operator constellations, RDF access paths, typical RDF constructs, and a variety of possible optimization approaches.

Our data generator relies on a deep study of DBLP. Although it is not possible to mirror *all* correlations found in the original DBLP data (e.g., we simplified when assuming independence between attributes in Section III-A), many aspects are modeled in faithful detail and the queries are designed in such a way that they build on exactly those aspects, which makes them realistic, understandable, and predictable.

Even without knowledge about the internals of engines, we identified deficiencies and reasoned about suspected causes. We expect the benefit of our benchmark to be even higher for developers that are familiar with the engine internals.

To give another proof of concept, in [34] we have successfully used SP²Bench to identify previously unknown limitations of RDF storage schemes: Among others, we identified scenarios where the advanced vertical storage scheme from [12] was slower than a simple triple store approach.

With the understandable DBLP scenario we also clear the way for coming language modifications. For instance, SPARQL update and aggregation support are currently discussed as possible extensions.[7] Updates, for instance, could be realized by minor extensions to our data generator. Concerning aggregations, the detailed knowledge of the document class counts and distributions (cf. Section III) facilitates the design of challenging aggregate queries with fixed characteristics.

References

[1] "Resource Description Framework (RDF): Concepts and Abstract Syntax," http://www.w3.org/TR/rdf-concepts/. W3C Rec. 02/2004.

[2] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," Scientific American, May 2001.

[3] "SPARQL Query Language for RDF," http://www.w3.org/TR/rdf-sparql-query/. W3C Rec. 01/2008.

[4] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and Complexity of SPAQRL," in *ISWC*, 2006, pp. 30–43.

[5] M. Stocker et al., "SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation," in *WWW*, 2008, pp. 595–604.

[6] O. Hartwig and R. Heese, "The SPARQL Query Graph Model for Query Optimization," in *ESWC*, 2007, pp. 564–578.

[7] S. Groppe, J. Groppe, and V. Linnemann, "Using an Index of Precomputed Joins in order to speed up SPARQL Processing," in *ICEIS*, 2007, pp. 13–20.

[8] A. Harth and S. Decker, "Optimized Index Structures for Querying RDF from the Web," in *LA-WEB*, 2005, pp. 71–80.

[9] S. Alexaki et al., "On Storing Voluminous RDF descriptions: The case of Web Portal Catalogs," in *WebDB*, 2001.

[10] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *ISWC*, 2002, pp. 54–68.

[11] S. Harris and N. Gibbins, "3store: Efficient Bulk RDF Storage," in *PSSS*, 2003.

[12] D. J. Abadi et al., "Scalable Semantic Web Data Management Using Vertical Partitioning," in *VLDB*, 2007, pp. 411–422.

[13] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," in *VLDB*, 2008.

[14] G. Lausen, M. Meier, and M. Schmidt, "SPARQLing Constraints for RDF," in *EDBT*, 2008, pp. 499–509.

[15] R. Cyganiac, "A relational algebra for SPARQL," HP Laboratories Bristol, Tech. Rep., 2005.

[16] A. Chebotko et al., "Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns," TR-DB-052006-CLJF.

[17] A. Polleres, "From SPARQL to Rules (and back)," in *WWW*, 2007, pp. 787–796.

[18] Z. P. Yuanbo Guo and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics: Science, Services and Agents on the WWW*, vol. 3, pp. 158–182, 2005.

[19] D. J. Abadi et al., "Using the Barton libraries dataset as an RDF benchmark," TR MIT-CSAIL-TR-2007-036, MIT.

[20] A. Schmidt et al., "XMark: A Benchmark for XML Data Management," in *VLDB*, 2002, pp. 974–985.

[21] M. Ley, "DBLP Database," http://www.informatik.uni-trier.de/~ley/db/.

[22] E. Elmacioglu and D. Lee, "On Six Degrees of Separation in DBLP-DB and More," *SIGMOD Record*, vol. 34, no. 2, pp. 33–40, 2005.

[23] J. Gray, *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 1993.

[24] M. J. Carey, D. J. DeWitt, and J. F. Naughton, "The OO7 Benchmark," in *SIGMOD*, 1993, pp. 12–21.

[25] "RDF Vocabulary Description Language 1.0: RDF Schema," http://www.w3.org/TR/rdf-schema/. W3C Rec. 02/2004.

[26] "RDF Semantics," http://www.w3.org/TR/rdf-mt/. W3C Rec. 02/2004.

[7] See http://esw.w3.org/topic/SPARQL/Extensions.

[27] C. Bizer and A. Schultz, "Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints," in *SSWS*, 2008.
[28] A. Magkanaraki et al., "Benchmarking RDF Schemas for the Semantic Web," in *ISWC*, 2002, p. 132.
[29] Y. Theoharis et al., "On Graph Features of Semantic Web Schemas," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 5, pp. 692–702, 2008.
[30] S. Wang et al., "Rapid Benchmarking for Semantic Web Knowledge Base Systems," in *ISWC*, 2005, pp. 758–772.
[31] C. Bizer and R. Cyganiak, "D2R Server publishing the DBLP Bibliography Database," 2007, http://www4.wiwiss.fu-berlin.de/dblp/.
[32] A. J. Lotka, "The Frequency Distribution of Scientific Production," *Acad. Sci.*, vol. 16, pp. 317–323, 1926.
[33] M. Schmidt et al., "SP2Bench: A SPARQL Performance Benchmark," Freiburg University, arXiv:0806.4627 cs.DB, Tech. Rep., 2008.
[34] M. Schmidt et al., "An Experimental Comparison of RDF Data Management Approaches in a SPAQL Benchmark Scenario," in *ISWC*, 2008.

APPENDIX

Q1

```
SELECT ?yr
WHERE {
 ?journal rdf:type bench:Journal.
 ?journal dc:title "Journal 1 (1940)"^^xsd:string.
 ?journal dcterms:issued ?yr }
```

Q2

```
SELECT ?inproc ?author ?booktitle ?title
       ?proc ?ee ?page ?url ?yr ?abstract
WHERE {
  ?inproc rdf:type bench:Inproceedings.
  ?inproc dc:creator ?author.
  ?inproc bench:booktitle ?booktitle.
  ?inproc dc:title ?title.
  ?inproc dcterms:partOf ?proc.
  ?inproc rdfs:seeAlso ?ee.
  ?inproc swrc:pages ?page.
  ?inproc foaf:homepage ?url.
  ?inproc dcterms:issued ?yr
  OPTIONAL { ?inproc bench:abstract ?abstract }
} ORDER BY ?yr
```

Q3

(a)
```
SELECT ?article
WHERE { ?article rdf:type bench:Article.
        ?article ?property ?value
        FILTER (?property=swrc:pages) }
```
(b) Q3a, but "swrc:month" instead of "swrc:pages"
(c) Q3a, but "swrc:isbn" instead of "swrc:pages"

Q4

```
SELECT DISTINCT ?name1 ?name2
WHERE { ?article1 rdf:type bench:Article.
        ?article2 rdf:type bench:Article.
        ?article1 dc:creator ?author1.
        ?author1 foaf:name ?name1.
        ?article2 dc:creator ?author2.
        ?author2 foaf:name ?name2.
        ?article1 swrc:journal ?journal.
        ?article2 swrc:journal ?journal
        FILTER (?name1<?name2) }
```

Q5

(a)
```
SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
        ?article dc:creator ?person.
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?person2.
        ?person foaf:name ?name.
        ?person2 foaf:name ?name2
        FILTER(?name=?name2) }
```
(b)
```
SELECT DISTINCT ?person ?name
WHERE { ?article rdf:type bench:Article.
        ?article dc:creator ?person.
        ?inproc rdf:type bench:Inproceedings.
        ?inproc dc:creator ?person.
        ?person foaf:name ?name }
```

Q6

```
SELECT ?yr ?name ?doc
WHERE {
  ?class rdfs:subClassOf foaf:Document.
  ?doc rdf:type ?class.
  ?doc dcterms:issued ?yr.
  ?doc dc:creator ?author.
  ?author foaf:name ?name
  OPTIONAL {
    ?class2 rdfs:subClassOf foaf:Document.
    ?doc2 rdf:type ?class2.
    ?doc2 dcterms:issued ?yr2.
    ?doc2 dc:creator ?author2
    FILTER (?author=?author2 && ?yr2<?yr) }
  FILTER (!bound(?author2)) }
```

Q7

```
SELECT DISTINCT ?title
WHERE {
  ?class rdfs:subClassOf foaf:Document.
  ?doc rdf:type ?class.
  ?doc dc:title ?title.
  ?bag2 ?member2 ?doc.
  ?doc2 dcterms:references ?bag2
  OPTIONAL {
    ?class3 rdfs:subClassOf foaf:Document.
    ?doc3 rdf:type ?class3.
    ?doc3 dcterms:references ?bag3.
    ?bag3 ?member3 ?doc
    OPTIONAL {
      ?class4 rdfs:subClassOf foaf:Document.
      ?doc4 rdf:type ?class4.
      ?doc4 dcterms:references ?bag4.
      ?bag4 ?member4 ?doc3 }
    FILTER (!bound(?doc4)) }
  FILTER (!bound(?doc3)) }
```

Q8

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes rdf:type foaf:Person.
  ?erdoes foaf:name "Paul Erdoes"^^xsd:string.
  { ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?doc2 dc:creator ?author.
    ?doc2 dc:creator ?author2.
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?doc2!=?doc &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  } UNION {
    ?doc dc:creator ?erdoes.
    ?doc dc:creator ?author.
    ?author foaf:name ?name
    FILTER (?author!=?erdoes) } }
```

Q9

```
SELECT DISTINCT ?predicate
WHERE {
  { ?person rdf:type foaf:Person.
    ?subject ?predicate ?person } UNION
  { ?person rdf:type foaf:Person.
    ?person ?predicate ?object } }
```

Q10

```
SELECT ?subj ?pred
WHERE { ?subj ?pred person:Paul_Erdoes }
```

Q11

```
SELECT ?ee
WHERE { ?publication rdfs:seeAlso ?ee }
ORDER BY ?ee LIMIT 10 OFFSET 50
```

Q12

(a) Q5a as ASK query
(b) Q8 as ASK query
(c) ASK {person:John_Q_Public rfd:type foaf:Person}