11th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2015)

At the 14th International Semantic Web Conference (ISWC2015), Bethlehem, PA, USA October, 2015

SSWS 2015 PC Co-chairs' Message

SSWS 2015 is the eleventh edition of the successful Scalable Semantic Web Knowledge Base Systems workshop series. The workshop series is focussed on addressing scalability issues with respect to the development and deployment of knowledge base systems on the Semantic Web. Typically, such systems deal with information described in Semantic Web languages such as OWL and RDF(S), and provide services such as storing, reasoning, querying and debugging. There are two basic requirements for these systems. First, they have to satisfy the applications semantic requirements by providing sufficient reasoning support. Second, they must scale well in order to be of practical use. Given the sheer size and distributed nature of the Semantic Web, these requirements impose additional challenges beyond those addressed by earlier knowledge base systems. This workshop brought together researchers and practitioners to share their ideas regarding building and evaluating scalable knowledge base systems for the Semantic Web.

This year we received 6 submissions. Each paper was carefully evaluated by three workshop Program Committee members. Based on these reviews, we accepted 5 papers for presentation. We sincerely thank the authors for all the submissions and are grateful for the excellent work by the Program Committee members.

October 2015

Thorsten Liebig
Achille Fokoue

Program Committee

Achille Fokoue
IBM Watson Research Center, USA

Raúl García-Castro
Univ. Politecnica de Madrid, Spain

Bernado Cuenca Grau
University of Oxford, UK

Volker Haarslev
Condordia University, Canada

Anastasios Kementsietsidis
Google Research, Mountain View, USA

Pavel Klinov
Complexible Inc., USA

Adila A. Krisnadhi
Wright State University, Ohio, USA

Thorsten Liebig
derivo GmbH, Germany

Raghava Mutharaju
Wright State University, Ohio, USA

Padmashree Ravindra
North Carolina State University, USA

Mariano Rodríguez-Muro
IBM Watson Research Center, USA

Boris Motik
University of Oxford, UK

Supriyo Chakaraborty
IBM Watson Research Center, USA

Takahira Yamaguchi
Keio University, Japan

Additional Reviewers

Zixi Quan
Condordia University, Canada

Table of Contents

Invited Talk: Making a Silk Purse

David Mizell

Cray Inc., USA

Abstract. In this talk, I'll tell the history of the development of "Urika", Cray's supercomputer-based SPARQL query engine, from my point of view as the original prototyper and one of the developers ever since. Urika evolved from some experimental "lock-free synchronization" prototypes in 2008 to a full-blown "appliance" product in 2012. I'll describe this evolution, going into detail on some of the technical decisions we made, relate some of the product decisions the suits made, and speculate on what we think the future may hold.

The OWL Reasoner Evaluation (ORE) 2015 Competition Report

Bijan Parsia, Nicolas Matentzoglu, Rafael Gonçalves, Birte Glimm, and Andreas Steigmiller

{bijan.parsia, nicolas.matentzoglu}@manchester.ac.uk, rafael.goncalves@stanford.edu, {birte.glimm, andreas.steigmiller}@uni-ulm.de

Abstract. The OWL Reasoner Evaluation (ORE) Competition is an annual competition (with associated) workshop which pits OWL 2 compliant reasoners against each other on various standard reasoning tasks against corpora. The 2015 competition was the third of its sort and had 14 reasoners competing in 6 tracks comprising 3 tasks (consistency, classification, and realisation) over two profiles (OWL 2 DL and EL). In this paper, we discuss the design, execution and results of the 2015 competition with particular attention to lessons learned for benchmarking, comparative experiments, and future competitions.

1 Introduction

The Web Ontology Language (OWL) is in its second iteration (OWL 2) and has seen significant adoption especially in the Health Care and Life Sciences. OWL 2 DL can be seen as a variant of the description logic $\mathcal{SROIQ}$ with the various other profiles being either subsets (e.g., OWL 2 EL) or[1] extensions (e.g., OWL 2 Full). Description logics generally are designed to be *computationally practical* so that, even if they do not have tractable worst-case complexity for key services, they nevertheless admit implementations which seem to work well in practice [2]. Unlike the early days of description logics or even of the direct precursors of OWL (DAML+OIL), the reasoner landscape [17, 8] for OWL is rich, diverse, and highly compliant with the OWL spec. Thus, we have a large number of high performance, production quality reasoners with similar core capacities (with respect to language features and standard inference tasks).

Research on optimising OWL reasoning continues apace, though empirical work still lags theoretical and engineering work in breath, depth, and sophistication. There is, in general, a lack of shared understanding of test cases, test scenarios, infrastructure, or experiment design. A common strategy in research communities to help address these issues is to hold competitions, that is, experiments designed and hosted by third parties on an independent (often constrained, but sometimes expanded) infrastructure. Such competitions (in contrast to published benchmarks) typically do not directly provide strong empirical evidence

[1] Some related standardised logics are subsets and extensions (e.g., RDFS) which are proper subsets of OWL 2 Full.

about the competing tools. Instead, they serve two key functions: 1) they provide a clear, motivating event that helps drive tools development (e.g., for correctness or performance) and 2) *components* of the competition are useful for subsequent research. Finally, competitions can be great fun and help foster a strong community. They can be especially useful for newcomers by providing a simple way to gain some prima facie validation of their tools without the burden of designing and executing complex experiments themselves.

Toward these ends, we have been running a competition for OWL reasoners (with an associated workshop): The OWL Reasoner Evaluation (ORE) competition. ORE has been running, in substantively its current form, for three years and in this paper we describe the 2015 competition (held in conjunction with the 28th International Description Logic Workshop (DL2015)[2] in June 2015).

2 Competition Design

The ORE competition is inspired by and modelled on the CADE ATP System Competition (CASC) [14, 22] which has been running for 25 years and been heavily influential in the automated theorem proving community[3] (esp. for first order logic).

Key common elements:

1. A number of distinct tracks/divisions/disciplines characterised by problem type (e.g., "effectively propositional" or "OWL 2 EL ontology").
2. The test problems are derived from a large, neutral, updated yearly set of problems (e.g., for CASC, the TPTP library [21]).
3. Reasoners compete (primarily) on number of problems solved with a tight per problem timeout.

As description logics have a varied set of core inference services supported by essentially all reasoners, ORE also has track distinctions based on task (e.g., classification or realisation). Other CASC inspired elements:

1. ORE 2015 consisted entirely of a "live" competition run during the DL workshop.
2. There was a secondary competition among DL attendees to predict the results for various reasoners.
3. Competitors and organisers were given custom designed t-shirts.

We observe that central to such competitions is participation, thus various incentives to participate are critical especially in the early years of the competition as it is trying to get established. Hence the importance of "fun" elements, incentives (e.g., prizes, bragging rights), as well as a reasonable chance of winning at least *something*.

[2] The websites for DL2015 and ORE2015 are archived at `http://dl.kr.org/dl2015/` and `https://www.w3.org/community/owled/ore-2015-workshop` respectively.

[3] See the CASC website for details on past competitions: `http://www.cs.miami.edu/~tptp/CASC/`. Also of interest, though not directly inspirational for ORE, is the SAT competition `http://www.satcompetition.org//`

2.1 Tracks

ORE 2015 had 6 tracks based on three central reasoning services (consistency, classification, and realisation) and two OWL profiles (OWL 2 DL and EL). Classification is, almost certainly, the most common and important reasoning service for ontologies to date. Consistency is, in some sense, the most fundamental service. Realisation gets us at least a minimal form of instance reasoning. These services are not ubiquitously supported, with realisation not handled by some reasoners. These have standard definitions (though any consequence equivalent definition would do):

- An ontology $\mathcal{O}$ is *consistent* if $\mathcal{O} \not\models \top \sqsubseteq \bot$ and inconsistent otherwise.
- The classification of an ontology $\mathcal{O}$ ($Cl(\mathcal{O})$) is $\{\alpha | \alpha = A \sqsubseteq B; A, B \in N_c \cup \{\bot, \top\}; \mathcal{O} \models \alpha\}$ where N_c is the set of class names in $\mathcal{O}$.
- The realisation of an ontology $\mathcal{O}$ ($Rl(\mathcal{O})$) is $\{\alpha | \alpha = A(x); x \in N_i, A \in N_c; \mathcal{O} \models \alpha\}$ where N_c is the set of class names, N_i is the set of individual names in $\mathcal{O}$.

We split out a track into a tractable profile when we have enough participants which are specifically tuned for that profile. In prior years we have had an RL and QL track, but the number of RL and QL specific reasoners is very low. We believe this is, in part, due to the fact that RL and QL users tend to be conjunctive query oriented. We hope to introduce a conjunctive query track in future years, but see the discussion below for some of the challenges there. All reasoners purporting to handle the entirety of OWL 2 DL are entered in all tracks. Thus we have specialised EL reasoners competing against complete OWL DL reasoners.

For each track, we award prizes to the top three participants for a total of 18 possible winners.

2.2 Corpus

The full competition corpus contains 1,920 ontologies, sampled from three source corpora: A January 2015 snapshot of Bioportal [12] containing 330 biomedical ontologies, the Oxford Ontology Library[4] with 793 ontologies that were collected for the purpose of ontology related tool evaluation and MOWLCorp [7], a corpus based on a 2014 snapshot of a Web-Crawl containing around 21K unique ontologies. Each competition comes with its own random stratified sample of ontologies from this base corpus - this means that not all 1,920 ontologies actually made it into the live competition. Ontology processing was done using the OWL API (3.5.1) [4].

As a first step, the ontologies of all three source corpora were collected and serialised into OWL/XML with their imports closure merged into a single ontology. The merging is, from a competition perspective, necessary to mitigate the bottleneck of loading potentially large imports repeatedly over the network and

[4] `http://www.cs.ox.ac.uk/isg/ontologies/`

because the hosts of frequently imported ontologies sometimes impose restrictions on the number of simultaneous accesses.[5] After the collection, the entire pool of ontologies is divided into three groups: (1) Ontologies with less than 50 axioms, (2) OWL 2 DL ontologies, (3) OWL 2 Full ontologies. The first group is removed from the pool. As reasoner developers may chose to tune their reasoners towards the ontologies in the three publicly available source corpora, we included a number of approximations into our pool. The entire set of OWL 2 Full ontologies was approximated into OWL 2 DL, i.e., we used a (slightly modified) version of the OWL API Profile checker to drop enough axioms so that the remainder is in OWL 2 DL. As some degree of OWL Fullness comes from illegal axiom interaction,[6] we repeated the "DLification" process twice. The OWL DL group was then approximated into OWL 2 EL and OWL 2 QL, using the approximation method employed by TrOWL [15]. As the only syntax that is uniformly supported by all reasoners participating the competition, we then serialised the current pool (including the original OWL 2 DL ontologies, the EL/QL-approximated ontologies and the "DLified" OWL 2 Full ontologies) into Functional Syntax, and gathered all relevant ontology metrics again. As some ontologies are included in more than one of the source corpora, we excluded at this point (as a last preprocessing step) all duplicates from the entire pool of ontologies and removed ontologies with TBoxes containing less than 50 axioms. This left us with the full competition dataset of 1,920 unique OWL 2 DL ontologies. The random stratified sampling for the competition then was done as follows: All ontologies were binned by size into the following groups: Very small (50-99 axioms), small (100-999 axioms), medium (1,000-9,999 axioms), large (10,000-100,000 axioms) and very large (more than 100,000 axioms). From each group, we attempted to sample 60 original ontologies, and 15 approximated ones for each competition. For the OWL 2 EL related track, the ontologies had to fall under the OWL 2 EL profile, for the OWL 2 DL competition the ontologies had to fall under OWL 2 DL but *not* under any of the three OWL 2 profiles, and for the two realisation challenges we only considered those ontologies that had at least 100 ABox axioms. This process resulted in the following six live competition corpora: 109 for OWL 2 EL realisation, 298 for OWL 2 EL classification and consistency, 264 for DL realisation and 306 for DL consistency and classification.

Figures 1 and 2 show the ontology sizes in terms of axiom counts and the usage of constructs through the corpus.

The full competition corpus, and the execution order of the competition, can be obtained from Zenodo [9].

[5] Which may be exceeded considering that all reasoners in the competition run in parallel.

[6] For example, an added declaration might introduce an illegal punning.

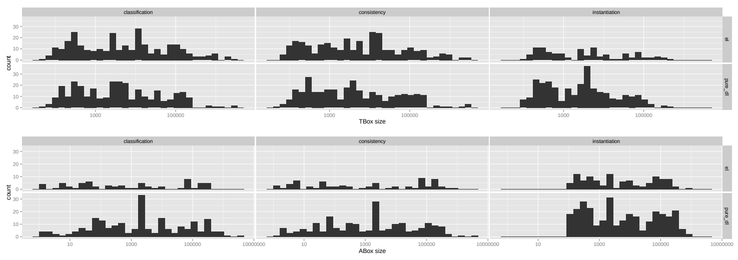


Fig. 1. Ontology size distribution of the full competition corpus.

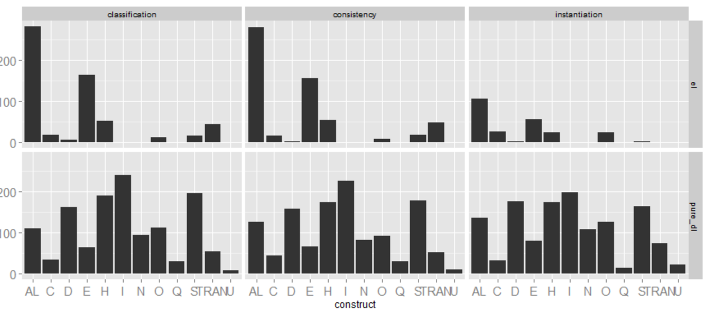


Fig. 2. Ontology size distribution of the full competition corpus.

2.3 Test Framework and Environment

The test framework used in ORE 2015 is a slightly modified version of the one used for ORE 2014 which is open sourced under the LGPL and available on Github.[7]

The framework takes a "script wrapper" approach to running reasoners instead of, for example, requiring all reasoners to use (a specific version of) the OWL API. While this puts some extra burden on established reasoners with good OWL API bindings this, combined with the requirement only to handle *some* OWL 2 standard syntax (with the very easy to parse and serialise Functional Syntax [11] as a fairly common choice), makes it very easy for new reasoners to participate even if they are written in hard-to-integrate with the JVM languages. The OWL API also is a very rich and rather heavyweight framework that is not tightly integrated with most reasoners. For example, systems using the OWL API generally consume more memory because they maintain the OWL API level representation of the ontology and the reasoner internal one. Thus, avoiding the OWL API can help competition performance. However, there is a standard script for OWL API based reasoners so it is fairly trivial to prepare an OWL API wrapped reasoner for competition.

[7] `https://github.com/andreas-steigmiller/ore-2014-competition-framework/`. A detailed description of the framework and how to run it is available there.

However, this is not necessarily a desirable outcome as encouraging reasoners to provide good OWL API support (thus supporting access to those reasoners by the plethora of tools which use the OWL API) is an outcome we want to encourage.

Reasoners report times, results, and any errors through the invocation script. Times are in wall clock time (CPU time is inappropriate because it will penalise parallel reasoners) and exclude "standard" parsing and loading of problems (i.e., without significant processing of the ontology). The framework enforces (configurable) timeouts for each reasoning problem. Results are validated by comparison between competitors with a majority vote/random tie breaking fallback strategy. Note, unlike CASC, we do not require reasoners to produce proofs of their results as this is not a standard feature of description logic reasoners and for many services (such as classification) it may be impractical. We are however experimenting with a more satisfactory justification-based technique for disagreement resolution [6] for future competitions.

The framework supports both serial and parallel execution of a competition. Parallel distributed mode is used for the live competition but serial mode is sufficient for testing or offline experiments. The framework also logs sufficient information to allow "replaying" the competition and includes scripts for a complete replay as well as jumping to the final results.

The competition was run on a cluster of 19 machines: 1 master machine that dispatched reasoners with problems to the 18 client machine as well as collecting and serving up results to a live display. Each machine sported an Intel Xeon 4-Core L5410 running at 2.33GHz with 12GB of RAM, for which 2GB were reserved for the operating system (i.e., 10GB could be used by the reasoners). The operating system was Ubuntu 14.04.02 LTS and the Java version was OpenJDK v1.7.0 64-bit. The reasoner execution was limited to 180s for each ontology in each track, where only 150s were allowed for reasoning and 30s could additionally be used for parsing and writing results in order to reduce the penalisation of reasoners with slow parsers. Hence, only if the time reported by the reasoner exceeded 150s was it interpreted as a timeout.

3 Participants

There were 14 reasoners participating with 11 purporting to cover OWL 2 DL and 3 being OWL EL specific, see Table 1. There is no specific penalty or test for being incomplete with respect to a profile and, indeed, one reasoner, TrOWL is intentionally incomplete for performance reasons.

The number of participants is fairly stable over the past three years ranging from 11 to 14. There is a stable core of participants with some fluctuation on the margin. Some reasoners are not entered by their original developers (e.g., Pellet) and ORE currently has no policy against that. We anticipate in the future that more coalition reasoners will be made available, though currently only MORe and Chainsaw use component reasoners (ELK and HermiT the former, FaCT++ the latter) which are also competing. MORe's coalition involves partitioning the ontology into an EL and DL part, dispatching each part to the respective tuned reasoner, and combining the results [16, 25]. Coalition reasoners that do not transform the ontology in any relevant way will need special consideration if they arrive.

Given the presence of deliberately incomplete (with respect to their purported profile) reasoners we are considering whether to modify the voting procedure to discount those reasoners' votes in certain cases. A full break-down of all tracks and competing reasoners can be seen in Table 2.

Table 1. Participant list

Reasoner	Profile supported	New to ORE?
Chainsaw [25]	DL	No
ELepHant [18]	EL	No
ELK [5]	EL	No
FaCT++ [24]	DL	No
HermiT[8] [1]	DL	No
jcel [10]	EL	No
Jfact [13]	DL	No
Konclude [20]	DL	No
MORe [16]	DL	No
PAGOdA [26]	DL	Yes
Pellet [19]	DL	Yes
Racer [3]	DL	Yes
TrOWL [23]	DL	No

Table 2. Breakdown of the competition by track.

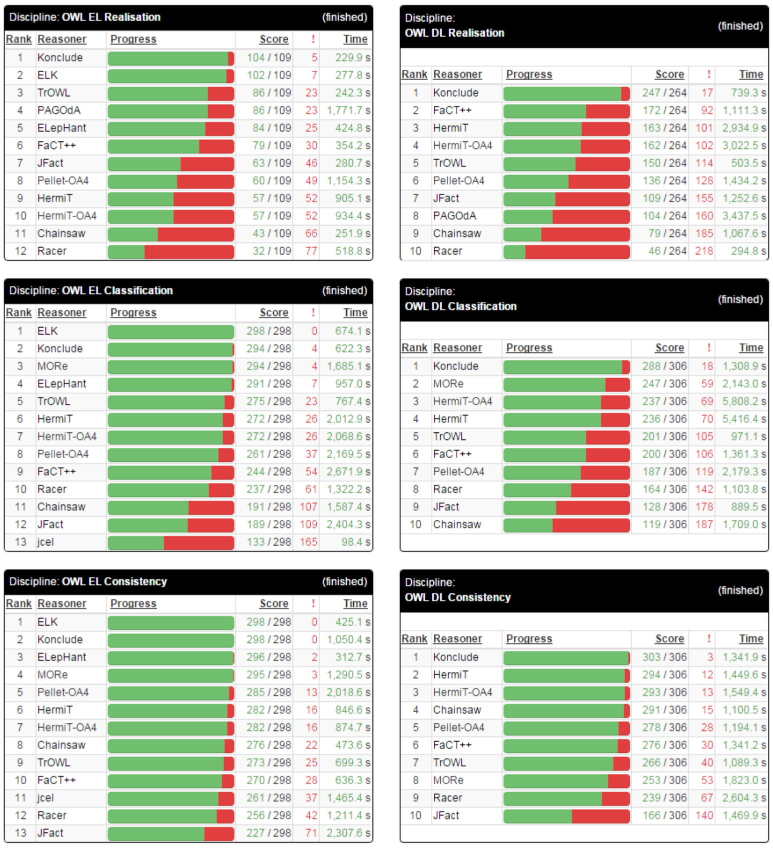
Task	Language	Competitors	Problems
Consistency	OWL 2 EL	13	298
	OWL 2 DL	10	306
Realisation	OWL 2 EL	12	109
	OWL 2 DL	10	264
Classification	OWL 2 EL	13	298
	OWL 2 DL	10	306

4 Results

Results, error reports and more details on the competition framework are available at `http://dl.kr.org/ore2015`. Figure 3 shows the results of all partici-

[8] HermiT was submitted with OWL API 3 and OWL API 4 bindings

pants in all tracks as displayed during the live competition. During the competition, these charts are dynamically updated as problems are being solved and reported.



Discipline: **OWL EL Realisation** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	Konclude		104 / 109	5	229.9 s
2	ELK		102 / 109	7	277.8 s
3	TrOWL		86 / 109	23	242.3 s
4	PAGOdA		86 / 109	23	1,771.7 s
5	ELepHant		84 / 109	25	424.8 s
6	FaCT++		79 / 109	30	354.2 s
7	JFact		63 / 109	46	280.7 s
8	Pellet-OA4		60 / 109	49	1,154.3 s
9	HermiT		57 / 109	52	905.1 s
10	HermiT-OA4		57 / 109	52	934.4 s
11	Chainsaw		43 / 109	66	251.9 s
12	Racer		32 / 109	77	518.8 s

Discipline: **OWL DL Realisation** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	Konclude		247 / 264	17	739.3 s
2	FaCT++		172 / 264	92	1,111.3 s
3	HermiT		163 / 264	101	2,934.9 s
4	HermiT-OA4		162 / 264	102	3,022.5 s
5	TrOWL		150 / 264	114	503.5 s
6	Pellet-OA4		136 / 264	128	1,434.2 s
7	JFact		109 / 264	155	1,252.6 s
8	PAGOdA		104 / 264	160	3,437.5 s
9	Chainsaw		79 / 264	185	1,067.6 s
10	Racer		46 / 264	218	294.8 s

Discipline: **OWL EL Classification** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	ELK		298 / 298	0	674.1 s
2	Konclude		294 / 298	4	622.3 s
3	MORe		294 / 298	4	1,685.1 s
4	ELepHant		291 / 298	7	957.0 s
5	TrOWL		275 / 298	23	767.4 s
6	HermiT		272 / 298	26	2,012.9 s
7	HermiT-OA4		272 / 298	26	2,068.6 s
8	Pellet-OA4		261 / 298	37	2,169.5 s
9	FaCT++		244 / 298	54	2,671.9 s
10	Racer		237 / 298	61	1,322.2 s
11	Chainsaw		191 / 298	107	1,587.4 s
12	JFact		189 / 298	109	2,404.3 s
13	jcel		133 / 298	165	98.4 s

Discipline: **OWL DL Classification** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	Konclude		288 / 306	18	1,308.9 s
2	MORe		247 / 306	59	2,143.0 s
3	HermiT-OA4		237 / 306	69	5,808.2 s
4	HermiT		236 / 306	70	5,416.4 s
5	TrOWL		201 / 306	105	971.1 s
6	FaCT++		200 / 306	106	1,361.3 s
7	Pellet-OA4		187 / 306	119	2,179.3 s
8	Racer		164 / 306	142	1,103.8 s
9	JFact		128 / 306	178	889.5 s
10	Chainsaw		119 / 306	187	1,709.0 s

Discipline: **OWL EL Consistency** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	ELK		298 / 298	0	425.1 s
2	Konclude		298 / 298	0	1,050.4 s
3	ELepHant		296 / 298	2	312.7 s
4	MORe		295 / 298	3	1,290.5 s
5	Pellet-OA4		285 / 298	13	2,018.6 s
6	HermiT		282 / 298	16	846.6 s
7	HermiT-OA4		282 / 298	16	874.7 s
8	Chainsaw		276 / 298	22	473.6 s
9	TrOWL		273 / 298	25	699.3 s
10	FaCT++		270 / 298	28	636.3 s
11	jcel		261 / 298	37	1,465.4 s
12	Racer		256 / 298	42	1,211.4 s
13	JFact		227 / 298	71	2,307.6 s

Discipline: **OWL DL Consistency** (finished)

Rank	Reasoner	Progress	Score	!	Time
1	Konclude		303 / 306	3	1,341.9 s
2	HermiT		294 / 306	12	1,449.6 s
3	HermiT-OA4		293 / 306	13	1,549.4 s
4	Chainsaw		291 / 306	15	1,100.5 s
5	Pellet-OA4		278 / 306	28	1,194.1 s
6	FaCT++		276 / 306	30	1,341.2 s
7	TrOWL		266 / 306	40	1,089.3 s
8	MORe		253 / 306	53	1,823.0 s
9	Racer		239 / 306	67	2,604.3 s
10	JFact		166 / 306	140	1,469.9 s

Fig. 3. Results of the competition by track as displayed from the live competition display. Score indicates the number of problems solved out of total problems for that track. The number of unsolved problems (whether by timeout, crash, or "wrong" results) are displayed in the next column. Time indicates the time actually taken to complete *solved* problems. Time is used to resolve ties in problems solved.

Out of the 6 competitions, 4 were won by the new hybrid reasoner Konclude [20], and two (EL-consistency and EL-classification) were won by ELK [5]. Figures 4 and 5 show how well the winning reasoners did in terms of reasoning time. There are a couple of observations to be made here. First, Konclude, the winner of all three DL disciplines, is doing consistently better on the majority of the easier ontologies, but towards the harder end on the right, other reasoners catch up.

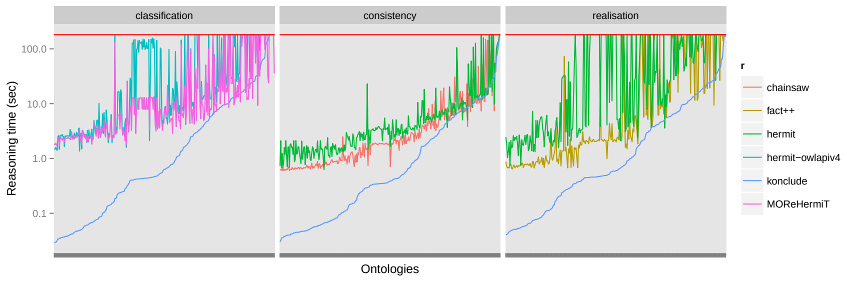


Fig. 4. Reasoning time of the three winning reasoners in each category: DL profile. Red line: Timeout. Ordered by speed of winning reasoner.

This is particularly obvious for the EL-classification competition. Up until a certain point, Konclude is doing much (sometime up to an order of magntitude) better than ELK (the winner of the discipline), but towards the harder end, ELK overtakes Konclude. Some of this may be due to JVM overhead for ELK and our "fire and forget" execution strategy. If we had a long running server based approach it might be that the JVM overhead for easy cases would be effectively amortised. Another interesting observation is the performance of ELepHants [18] consistency check, which regularly outperforms both ELK and Konclude. We speculate that this is due to differences in whether parsing time is incorporated in the reported time (e.g., ELK does this for all tasks and Konclude does this for consistency checking).

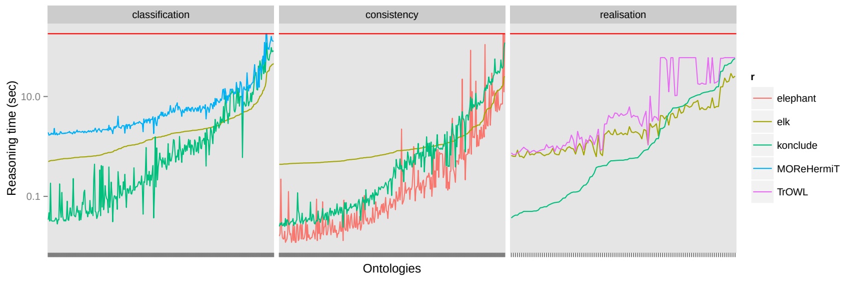


Fig. 5. Reasoning time of the three winning reasoners in each category: EL profile. Red line: Timeout. Ordered by speed of winning reasoner.

A full break-down for all reasoners by competition can be seen in Table 3.

The competition is reasonably challenging: In only two tracks (EL consistency and EL classification) did any reasoner solve all the problems in competition conditions. Figure 6 shows a detailed breakdown of how many problems were solved by how many reasoners (in percent).

Reasoner	Task	Success		Timeout		Error	
		DL	EL	DL	EL	DL	EL
Chainsaw	CL	122	191	168	94	16	13
Chainsaw	CON	292	276	3	19	11	3
Chainsaw	REAL	82	44	166	63	16	2
ELepHant	CL	NA	293	NA	5	NA	0
ELepHant	CON	NA	296	NA	2	NA	0
ELepHant	REAL	NA	109	NA	0	NA	0
ELK	CL	NA	298	NA	0	NA	0
ELK	CON	NA	298	NA	0	NA	0
ELK	REAL	NA	109	NA	0	NA	0
FaCT++	CL	202	244	87	51	17	3
FaCT++	CON	279	270	14	22	13	6
FaCT++	REAL	183	79	56	27	25	3
HermiT	CL	241	273	63	25	2	0
HermiT	CON	296	282	7	16	3	0
HermiT	REAL	167	57	92	52	5	0
HermiT-4	CL	241	273	63	25	2	0
HermiT-4	CON	296	282	6	16	4	0
HermiT-4	REAL	165	57	93	52	6	0
jcel	CL	NA	134	NA	158	NA	6
jcel	CON	NA	262	NA	34	NA	2
Jfact	CL	143	208	104	88	59	2
Jfact	CON	174	229	80	69	52	0
Jfact	REAL	128	66	89	43	47	0
Konclude	CL	298	298	7	0	1	0
Konclude	REAL	261	109	2	0	1	0
Konclude	CON	305	298	1	0	0	0
MORe	CL	266	296	38	2	2	0
MORe	CON	264	295	40	3	2	0
Pagoda	REAL	120	96	49	13	95	0
Pellet-4	CL	188	261	104	28	14	9
Pellet-4	REAL	187	75	53	32	24	2
Pellet-4	CON	280	286	26	12	0	0
Racer	CL	218	260	86	38	2	0
Racer	CON	257	258	48	40	1	0
Racer	REAL	186	78	75	31	3	0
TrOWL	CL	271	275	0	0	35	23
TrOWL	CON	270	273	0	0	36	25
TrOWL	REAL	221	87	0	0	43	22

Table 3. Full break-down of solved problems by reasoner and task. Note that "HermiT-4" refers to the current version of HermiT wrapped in the OWL API version 4.

It is interesting to observe that the union of all reasoners successfully process all EL reasoning problems. As one might expect, realisation is still challenging for reasoners. But in all tracks, for the majority of reasoners, the ORE problems provide a good target for optimisation. We know, from the results of the competition, that these problems are (almost) all in principle solvable on a modest machine in around 3 minutes.

5 Discussion

The top slots in all tracks have been dominated by Konclude (and to a lesser extend by ELK) for two years now. Konclude is an highly optimised, very efficient reasoner whose developers continuously test it against a vast set of available ontologies. Even so, there is interesting jockeying around second and third place

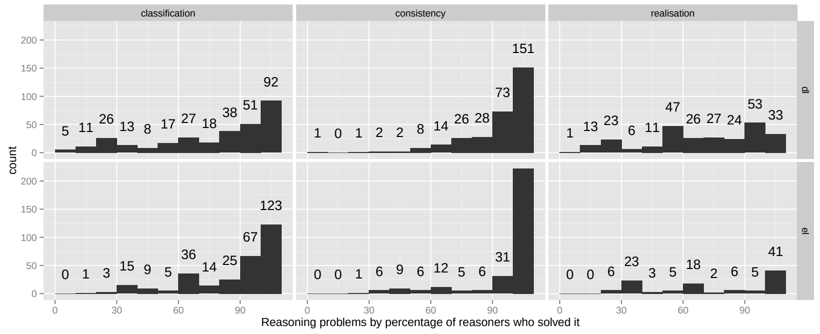


Fig. 6. Number of reasoning problems by percentage of reasoners solving them. For example, 5 DL-classification tasks where not solved by any reasoner, and 123 EL-classification tasks were solved by all reasoners.

for all tracks, and we were impressed with how well older reasoners, which have not been updated recently (notably Pellet and Racer), fared.

The robustness experiments in [2] used a much longer timeout (up to 2 hours per test), though the analysis clustered results by subdivisions of the timeout period. That suggests that a slightly longer timeout might significantly increase the total number of solved problems across reasoners. This needs to be balanced by the increased running time of the competition (which is bounded by the slowest reasoner). We prefer the bulk of the competition to be executed during a single day of the DL workshop to facilitate engagement which imposes fairly tight limits on the timeout and number of problems. (This year, due to technical issues, we were not able to do that.) Having a separate offline competition remains an option, but it is unclear that this extra significant effort produces much benefit.

However, the ORE workshop solicits "challenge" ontologies from ontology developers partly in the hopes of directing reasoner developer attention to real user performance needs. Unfortunately, we have not yet managed to do a "user ontology" track, though we are hoping to do so as a satellite event at OWLED 2015. This will almost certainly have to be offline and, of course, many of the submitted ontologies are currently unsolved by current reasoners.

The most important next expansion of tracks is to conjunctive query answering (CQA). Setting up a meaningful CQA competition is significantly more difficult, because we do not only have to consider ontologies, but also queries and data. Gathering suitable (meaningful) queries is probably the most difficult hurdle to overcome. However, we made significant progress toward a reasonable design this year and hope to incorporate it in next year's competition.

Another area of interest is application style benchmarks which would situate the reasoning task in the context of a pattern of use characteristic of a real or realistic application. This might include modification of the ontology or data during the competition run.

6 Conclusion

The ORE 2015 Reasoner Competition continues the success of its predecessors. Participants, workshop attendees, and interested bystanders all had fun, and the ORE 2015 corpus, whether used with the ORE framework or in a custom test harness, is a significant and distinct corpus for reasoner experimentation. Developers can easily rerun this years competition with new or updated reasoners to get a sense of their relative progress and we believe that solving all the problems in that corpus in similar or somewhat relaxed time constraints is a reliable indicator of a very high quality implementation.

Acknowledgments. The ORE competition has been the work of many people over the years and we would like to especially acknowledge the contributions of Ernesto Jiménez-Ruiz for running the very first primitive competition and being a PC chair for all workshops; Samantha Bail for being a PC chair as well as implementing the first "live results" screen; Ian Horrocks for helping getting the project started; and Yevgeny Kasakov for helpful discussions on the competition design as well as finding a critical bug *just before the competition started*. We also would like to acknowledge the generous support of B2i Healthcare[9] for their repeated donations of prize money and the DBOnto project[10] for funding compeition T-Shirts.

Finally, the competition would not have been possible without the donation of cluster time by Konstantin Korovin (funded by Royal Society research grant RG080491).

References

1. Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 Reasoner. *J. Autom. Reasoning*, 53(3):245–269, 2014.
2. Rafael S. Gonçalves, Nicolas Matentzoglu, Bijan Parsia, and Uli Sattler. The Empirical Robustness of Description Logic Classification. In *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*, pages 277–280, 2013.
3. Volker Haarslev, Kay Hidde, Ralf Möller, and Michael Wessel. The RacerPro knowledge representation and reasoning system. *Semantic Web*, 3(3):267–277, 2012.
4. Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.
5. Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. The Incredible ELK - From Polynomial Procedures to Efficient Reasoning with EL Ontologies. *J. Autom. Reasoning*, 53(1):1–61, 2014.
6. Michael Lee, Nicolas Matentzoglu, Bijan Parsia, and Uli Sattler. A multi-reasoner, justification-based approach to reasoner correctness. In *International Semantic Web Conference*, 2015.

[9] `http://b2i.sg`
[10] `http://www.cs.ox.ac.uk/projects/DBOnto/`

7. Nicolas Matentzoglu, Samantha Bail, and Bijan Parsia. A Snapshot of the OWL Web. In *The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, pages 331–346, 2013.
8. Nicolas Matentzoglu, Jared Leo, Valentino Hudhra, Uli Sattler, and Bijan Parsia. A survey of current, stand-alone owl reasoners. In Michel Dumontier, Birte Glimm, Rafael Gonçalves, Matthew Horridge, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Giorgos Stamou, and Giorgos Stoilos, editors, *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation*, volume 1387. CEUR-WS, 2015.
9. Nicolas Matentzoglu and Bijan Parsia. ORE 2015 reasoner competition dataset http://dx.doi.org/10.5281/zenodo.18578, June 2015.
10. Julian Mendez. jcel: A Modular Rule-based Reasoner. In *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK, July 1st, 2012*, 2012.
11. Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Mike Smith. Owl 2 web ontology language: Structural specification and functional-style syntax. W3C, 2009.
12. Natalya Fridman Noy, Nigam H. Shah, Patricia L. Whetzel, Benjamin Dai, Michael Dorf, Nicholas Griffith, Clement Jonquet, Daniel L. Rubin, Margaret-Anne D. Storey, Christopher G. Chute, and Mark A. Musen. BioPortal: ontologies and integrated data resources at the click of a mouse. *Nucleic Acids Research*, 37(Web-Server-Issue):170–173, 2009.
13. Ignazio Palmisano. JFact repository, 2015.
14. F.J. Pelletier, G. Sutcliffe, and C.B. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.
15. Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Soundness Preserving Approximation for TBox Reasoning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.
16. Ana Armas Romero, Bernardo Cuenca Grau, and Ian Horrocks. MORe: Modular Combination of OWL Reasoners for Ontology Classification. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, pages 1–16, 2012.
17. Uli Sattler and Nicolas Matentzoglu. List of Reasoners (owl.cs) http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/. Modified: 01/09/2014.
18. Baris Sertkaya. The ELepHant Reasoner System Description. In *Informal Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation (ORE-2013), Ulm, Germany, July 22, 2013*, pages 87–93, 2013.
19. Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.
20. Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: System description. *J. Web Sem.*, 27:78–85, 2014.
21. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
22. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.
23. Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 Reasoning Infrastructure. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II*, pages 431–435, 2010.

24. Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 292–297, 2006.
25. Dmitry Tsarkov and Ignazio Palmisano. Chainsaw: a Metareasoner for Large Ontologies. In *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK, July 1st, 2012*, 2012.
26. Yujiao Zhou, Yavor Nenov, Bernardo Cuenca Grau, and Ian Horrocks. Pay-as-you-go OWL query answering using a triple store. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1142–1148, 2014.

On the Evaluation of RDF Distribution Algorithms Implemented over Apache Spark

Olivier Curé, Hubert Naacke, Mohamed-Amine Baazizi, Bernd Amann

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris,
CNRS, UMR 7606, LIP6, F-75005, Paris, France
{firstName.lastname}@lip6.fr

Abstract. Querying very large RDF data sets in an efficient and scalable manner requires parallel query plans combined with appropriate data distribution strategies. Several innovative solutions have recently been proposed for optimizing data distribution with or without predefined query workloads. This paper presents an in-depth analysis and experimental comparison of five representative RDF data distribution approaches. For achieving fair experimental results, we are using Apache Spark as a common parallel computing framework by rewriting the concerned algorithms using the Spark API. Spark provides guarantees in terms of fault tolerance, high availability and scalability which are essential in such systems. Our different implementations aim to highlight the fundamental implementation-independent characteristics of each approach in terms of data preparation, load balancing, data replication and to some extent to query answering cost and performance. The presented measures are obtained by testing each system on one synthetic and one real-world data set over query workloads with differing characteristics and different partitioning constraints.

1 Introduction

During the last few years, an important number of papers have been published on data distribution issues in RDF database systems, [14], [8], [24], [10] and [11] to name a few. Repositories of hundreds of millions to billions of RDF triples are now more and more frequent and the main motivation of this research movement is the efficient management of ever growing size of produced RDF data sets. RDF being one of the prominent data models of the Big data ecosystem, RDF repositories have to cope with issues such as scalability, high availability, fault tolerance. Other systems addressing these issues like NoSQL systems [21], generally adopt a scale-out approach consisting of distributing both data storage and processing over a cluster of commodity hardware.

Depending on the data model, it is well-known that an optimal distribution in terms of data replication rate (characterizing the number of copies of a given information across the cluster), load balancing and/or query answering performance is hard to achieve. RDF data encode large graphs and obtaining a balanced partitioning into smaller components with specific properties is known to be an NP-hard problem in general. Hence, most distributed RDF systems are proposing heuristic-based approaches for producing optimal data distributions with respect to specific query processing environments and query workloads. In a distributed RDF data processing context, the total cost of a distributed query evaluation process is often dominated by the data exchange cost produced by a large number of triple pattern joins corresponding to complex SPARQL graph pattern. Therefore, one of the supreme goals of all distributed RDF query processing solutions is to limit the amount of data exchanged over the cluster network through optimal data partitioning and replication strategies. Each such strategy also comes with a set of data transformation, storage and indexing steps that are more or less cost intensive.

The first systems considering distributed storage and query answering for RDF data appeared quite early in the history of RDF. Systems like Edutella [18] and RDFPeers [2] were already tackling partitioning issues in the early 2000s. More recent systems like YARS2 [12], HadoopRDF [15] and Virtuoso [5] are based on hash partitioning schemes for distributing RDF triple indexes on different cluster nodes. In 2011, [14], henceforth denoted *nHopDB*, presented the first attempt to apply graph partitioning on RDF data sets for distributed SPARQL query evaluation. In the following, the database research community has proposed a large number of RDF triple data partitioning and replication strategies for different distributed data and query processing environments. Recent systems are either extending the graph partitioning approach [13] or are complaining about their limitations [17].

As a consequence of the plethora of distribution strategies, it is not always easy to identify the most efficient solution for a given context. The first objective of this paper is to clarify this situation by conducting evaluations of prominent RDF triple distribution algorithms. A second goal is to consider Apache Spark as the parallel computing framework for hosting and comparing these implementations. This is particularly relevant in a context where a large portion of existing RDF distributed databases, $e.g.$ nHopDB [14], Semstore [24], SHAPE [17], SHARD [20], have been implemented using Apache Hadoop, an open source MapReduce [4] reference implementation. These implementations suffer from certain [22] limitations of MapReduce for processing large data sets, some of them being related to the high rate of disk reads and writes. We have chosen Spark since it is up to 100 times more efficient than Hadoop through Resilient Distributed Datasets (RDD) implementing a new distributed and fault tolerant memory abstraction.

Our experimentation is conducted over a reimplementation of five data distribution approaches where two of them are hash-based, two of them are based on graph partitioning with and without query workload awareness and one is hybrid combing hash-based partitioning and query workload awareness. Each system is evaluated over a synthetic and a real-world data-set with varying cluster settings and on a total of six queries which differ in terms of their shape, $e.g.$, star and property chains, and selectivity. We present and analyze experimentations conducted in terms of data preparation cost, distribution load balancing, data replication rate and query answering performance.

2 Background knowledge

2.1 RDF - SPARQL

RDF is a schema-free data model that permits to describe data on the Web. It is usually considered as the cornerstone of the Semantic Web and the Web of Data. Assuming disjoint infinite sets U (RDF URI references), B (blank nodes) and L (literals), a triple (s,p,o) $\in$ (U $\cup$ B) x U x (U $\cup$ B $\cup$ L) is called an *RDF triple* with s, p and o respectively being the subject, predicate and object. Since subjects and objects can be shared among triples, a set of RDF triples generates an *RDF graph*.

SPARQL [1] is the standard query language for RDF graphs (triple collections) based on *graph patterns* for extracting information from RDF graphs. Let V be an infinite set of variables disjoint with U, B and L. Then, a triple $tp \in$ (U $\cup$ V) x (U $\cup$ V) x (U $\cup$ V $\cup$ L) followed by a dot '.' is a SPARQL triple pattern. The semantics of a triple pattern follows the standard *matching semantics* which consists in finding all mappings $\mu : V \rightarrow U \cup B \cup L$ such that $\mu(tp)$ is a triple in the input graphs. Graph patterns are defined recursively. A possibly empty set of triple patterns is a basic graph pattern. The semantics of a basic graph pattern gp is defined by the conjunctive extension of the triple matching semantics ($\mu(gp)$ is a connected or disconnected subgraph of the input graphs). If gp_1 and gp_2 are graph patterns, then $\{gp_1\}$ is a group pattern, gp_1 `OPTIONAL` $\{gp_2\}$ is an optional pattern, $\{gp_1\}$ `UNION` $\{gp_2\}$ is a pattern alternative. Finally, a graph pattern gp can contain any a constraint `FILTER` C where C is a built-in condition to restrict the solutions of a graph pattern match according to the expression C.

The complete SPARQL syntax follows the SELECT-FROM-WHERE syntax of SQL queries. The `SELECT` clause specifies the variables appearing in the query result set, the optional `FROM` clause specifies the input graphs (an input graph can be defined by default), the `WHERE` clause defines a graph pattern which is matched against the input RDF graphs.

2.2 Apache Spark

Apache Spark [26] is a cluster computing framework whose design and implementation started at UC Berkeley's AMPlab. Just like Apache Hadoop, Spark enables parallel computations on unreliable machines and automatically handles locality-aware scheduling, fault tolerance and load balancing tasks. While both systems are based on a data flow computation model, Spark is more efficient than Hadoop's MapReduce for applications requiring the reuse working data sets across multiple parallel operations. This efficiency is due to Spark's Resilient Distributed Dataset (RDD) [25], a distributed, lineage supported fault tolerant memory abstraction that enables in-memory computations more efficiently than Hadoop (which is mainly disk-based). The Spark API also simplifies data-centric programming by integrating set-oriented functions like $join$ and $filter$ which are not natively supported in Hadoop.

2.3 METIS graph partitioner

Finding a graph partitioning which is optimal with respect to certain constraints is an NP-hard problem which is practically solved by approximative algorithms like[7]. These algorithms are generally still not efficient for very large graphs hence motivating a multi-level propagation approach where the graph is coarsened until its size permits

[1] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/

to use one of the approximate solutions. The METIS system [16] follows this approach and is known to reach its limits for graphs of about half a billion triples. METIS takes as input an unlabeled, undirected graph and an integer value corresponding to the desired number of partitions. Its output provides a partition number for each node of the graph. As explained in the following section, nHopDB [14] and WARP [13] are two recent systems that are using METIS to partition RDF graphs.

3 RDF data partitioning methods

In this section, we present the main features and design principles of the RDF data partitioning methods we have chosen to compare with respect to their data preparation cost, storage load balancing, data replication and query processing costs. It is obvious that the performance results, and in particular the results concerning query processing performance, have to be considered with caution. Our goal is not to rank the different methods, but to analyze some general properties (including implementation effort) in the context of Apache Spark, which is a common modern scalable distributed data processing environment. More details about these implementations are described in Section 6.

As a starting point, we consider four different data partitioning approaches which can be characterized as hash and graph partitioning based. Each category is divided into two approaches which have been used in various systems and described in conference publications. Our fifth system corresponds to a new hybrid approach that mixes a hash-based approach with a replication strategy that enables to efficiently process long chain queries. Note that we do not consider range-based partitioning approaches since they are rarely used in existing systems due to their inefficiency.

3.1 Hash-based RDF data partitioning

The two approaches defined in this section correspond to families of RDF database systems rather than to specific systems (as in the next section). The basic of hash-based Data partitioning consists in applying to each RDF triple a hash function which returns for some triple-specific key value the node where the triple should be stored. One advantage of hash-based approaches is that they do not require any additional structure to locate the partition of a given triple except the hash function and the key value. Data replication can be achieved by defining several hash functions.

Random hashing: In a distributed random hash-based solution, the partitioning key does not correspond to a particular element of the data model like the graph, subject, property or object of a triple. For instance, the key can correspond to an internal triple identifier or to some other value obtained from the entire triple. The former solution is the one adopted by the Trinity.RDF system [27]. Some other forms of random partitioning exist and may require an additional structure for directed lookups to cluster nodes where triples are located, $e.g.$ round-robin approach. We do not consider such random partitioning approaches in our evaluation since they do not provide particularly useful data placement properties for any of the query shapes (star, property chains, tree, cycle or hybrid) used in our experiments (see Appendix A).

RDF triple element hashing: This approach has been adopted by systems like YARS2, Virtuoso, Jena ClusteredTDB and SHARD. In these systems, the hashing key provided to the hash function is composed of one or several RDF quadruple elements (graph, subject, property, object). Partitioning by subject provides the nice property of ensuring that star-shaped queries, $i.e.$ queries composed of a graph where one node has an out-degree greater than 1 and all other nodes are leaves, are performed locally on a given machine. Nevertheless they do not provide guarantees for queries composed of property chains or more complex query patterns. We will study the performance of a subject-hash based partitioning in Section 6.

3.2 Graph-based partitioning approaches

Hash-based data partitioning methods are likely to require a high data exchange rate over the network for more complex query patterns composed of longer property chains. One way to address this issue is to introduce data replication and/or to use more structured hashing functions adapted for a given query workload. Of course, these extensions come with at an additional processing cost which needs to be considered with attention. Systems corresponding to each of these approaches are considered next.

nHopDB: The data partitioning approach presented in [14] is composed of two steps. In a first stage, the RDF data set is transformed such that it can be sent to the METIS graph partitioner. This transformation removes properties and adds inverted subject-object edges to obtain an undirected graph. Then, the partitions obtained by METIS are translated into triple allocations over the cluster (all triples of the same partition are located on the same node). The partition state obtained at the end of this first stage is denoted as 1-hop. The second stage starts and corresponds to an overlap strategy which is performed using a so-called n-hop guarantee. Intuitively, for each partition, each leaf l is extended with triples whose subject correspond to l. This second replication stage can be performed several times on the successively generated partitions. Each execution increases the n-hop guarantee by a single unit.

[14] describes an architecture composed of a data partitioner and a set of local query engine workers implemented by RDF-3X [19] database instances. Some queries can be executed locally on a single node and thus enjoy all the optimization machinery of RDF-3X. For queries where the answer set spans multiple partitions, the Hadoop MapReduce system is used to supervise query processing.

WARP: The WARP system [13] has been influenced by nHopDB and the Partout system [8] (the two authors of WARP also worked on Partout). WARP borrows from nHopDB its graph partitioning approach and 2-hop guarantee. Like Partout, it then refines triple allocations by considering a given query workload of the most frequently performed queries over the given data set. The system considers that this query workload is provided in one way or another. More exactly, each of these queries is transformed into a set of query patterns (defining a class of equivalent queries) and WARP guarantees that frequent queries can be distributed over the cluster nodes and processed locally without exchanging data across machines (the final result is defined by the union of locally obtained results). WARP proceeds as follows for partitioning and replicating a RDF triple collection:

1. A first step partitions the transformed unlabeled and undirected RDF graph using the METIS graph partitioner as described in nHopDB.
2. The RDF triples are fragmented according to the partitions of their subjects and loaded into the corresponding RDF-3X [19] database instance.
3. A replication strategy is applied to ensure a 2-hop guarantee.
4. Finally, WARP chooses for each query pattern *qp* a partition which will receive all triples necessary for evaluating pattern *qp* locally. For this, WARP decomposes each query pattern obtained from the query workload into a set of sub-queries which are potential starting points or *seed queries* for the evaluation of the entire query pattern. Then, WARP estimates for each seed query and partition the cost of transferring missing triples into the current partition and selects the seed query candidate that minimizes this cost. An example is presented in Section 4.3.

The WARP system implements its own distributed join operator to combine the local sub-queries. Locally, the queries are executed using RDF-3X. As our experiments confirm, most of the data preparation effort for WARP is spent in the graph partitioning stage.

3.3 Hybrid partitioning approach

The design of this original hybrid approach has been motivated by our analysis of the WARP system as well as some hash-based solutions. We have already highlighted (as confirmed in the next section) that the hash-based solutions require short data preparation times but come with poor query answering performance for complex query patterns. On the other hand, the WARP system proposes an interesting analysis of query workloads which is translated into an efficient data distribution. We will present in our experiments a hybrid solution which combines RDF triple element hashing using subjects as hash keys with query workload aware triple replication is described in the last step of WARP.

4 Spark system implementations

4.1 Data set loading and encoding

All data sets are first loaded on the cluster's Hadoop File System(HDFS). The loading rate in our cluster averages 520.000 triples per second which allows us to load large data sets like LUBM 2K or Wikidata in less than 10 minutes.

Like in most RDF stores, each data set is encoded by providing a distinct integer value to each node and edge of the graph (see Chapter 4 in [3] for a presentation of RDF triple encoding methods). The encoding is performed in parallel in one step using the Spark framework.[2] The encoded data sets, together with their dictionaries (one for the properties and another for subjects and objects) are also loaded into HDFS.

[2] More implementation details can be found at http://www-bd.lip6.fr/wiki/doku.php?id=site:recherche:logiciels:rdfdist.

4.2 Hash-based data distribution

The implementation of hash-based partitioning approaches in Spark is relatively straightforward since the Spark API directly provides hash-based data distribution functionalities. We achieve random-hash partitioning by using the whole RDF triple as hash key. Triple element hashing is obtained by using the triple subject URI. In our experiments, we do not provide replication by adding other hashing function. The query answering evaluation is performed forthrightly following a translation from SPARQL to Spark scripts requiring a mix of `map`, `filter`, `join` and `distinct` methods performed over RDDs.

4.3 Graph partitioning-based data distribution

The two approaches in this partitioning category, nHopDB and WARP, require three METIS related steps for the preparation, computation and transformation of the results. Since METIS only can deal with unlabeled and undirected graphs, we start by removing predicates from the data sets and appending the reversed subject/object edges to the graph. Using METIS also imposes limitations in terms of accepted graph size. Indeed, the largest graph that can be processed contains about half a billion nodes. Consequently, we limit our experimentations to data sets of at most 250 million RDF triples provided that their undirected transformation yields graphs of 500 million nodes. The output of METIS is a set of mapping assertions between nodes and partitions. Based on these mappings, we allocate a triple to the partition of its subject. In terms of data encoding, we extend triples with partition identifiers yielding quads. Note that at this stage, the partition identifier can be considered as 'logical' and not 'physical' since the data is not yet stored on a given cluster node. We would like to stress that the preparation and transformation phases described above are performed in parallel using Spark programs.

nHopDB: In the Spark implementation of nHopDB, the n-hop guarantee is computed over the RDD corresponding to the generated quads. This Spark program can be executed (n-1) times to obtain an n-hop guarantee.

WARP: Our implementation of WARP analyzes the query workload generalization using Spark built-in operators. For instance, consider the following graph pattern of a query denoted Q1:

```
?x advisor ?y . ?y worksFor ?z . ?z subOrganisation ?t
```

For processing this pattern, the system uses the `filter` operator to select all triples that match the `advisor`, `worksFor` and `subOrganization` properties. Then, the `join` operator is used to perform equality join predicates on variables `y` and `z`. The query result is a set of variable bindings. We extend the notion of variable bindings with the information regarding the partition identifier of each triple. For instance, an extract of a Q1's result (in an decoded readable form) is represented as {`(Bob,Alice,1), (Alice, DBteam,3),(DBteam, Univ1,1)`}. The result for pattern `?y worksFor ?z` contains the triple binding {`(Alice, DBteam,` `3)}` which means that `"Alice"` and `"DBTeam"` are bound to variables `?x` and `?y` and the triple is located on partition 3. The two other triples for triple patterns `?x advisor ?y` and `?z subOrganisation ?t` are located on partition 1. It is easy to see that by choosing the seed query `?x advisor ?y` or `?z subOrganisation ?t`, we need to copy only triple `(Alice, worksFor, DBteam)` in partition 3 to partition 1 whereas by choosing pattern `?y worksFor ?z` two triples have to be copied to partition 1. As specified earlier in Section 3.2, we consider all the candidate seeds to choose the seed that implies the minimal number of triples to replicate.

Finally, for querying purposes, each query is extended with a predicate enforcing local evaluation by joining triples with the same partition identifier.

4.4 Hybrid approach

This approach is mixing the subject-based hashing method with the WARP workload-aware processing. Hence, using our standard representations of triples and quads together with Spark's data transformation facilities made our coding effort for this experiment relatively low.

5 Experimental setting

5.1 Data sets and queries

In this evaluation, we are using one synthetic and one real world data set. The synthetic data set corresponds to the well-established LUBM [9]. We are using three instances of LUBM, denoted LUBM1K, LUBM2K and LUBM10K which are parameterized respectively with 1000, 2000 and 10000 universities. The real world data set consists in Wikidata [23], a free collaborative knowledge base which will replace Freebase [1] in 2015. Table 2 presents the number of triples as well as the size of each of these data sets.

Data set	#triples	nt File Size
LUBM 1K	133 M	22 GB
LUBM 2K	267 M	43 GB
LUBM 10K	1,334 M	213 GB
Wikidata	233 M	37 GB

Table 1. Data set statistics of our running examples

Concerning queries, we have selected three SPARQL queries from LUBM (namely queries #2, #9 and #12 respectively denoted Q2, Q3 and Q4) extended by a new query, denoted Q1, which requires a 3-hop guarantee to be performed locally on the nHopDB, WARP and hybrid implementations. To complement the query evaluation, we have created two queries for the Wikidata experiments, resp. Q5 and Q6. The first one takes the form of a 3-hop property chain query that shows to be much more selective than the LUBM ones, the second one is shaped as a simple star and was motivated by the absence of such a form in our query set. All six queries are presented in Appendix A.

5.2 Computational environment

Our evaluation was deployed on a cluster consisting of 21 DELL PowerEdge R410 running a Debian distribution with a 3.16.0-4-amd64 kernel version. Each machine has 64GB of DDR3 RAM, two Intel Xeon E5645 processors each of which is equipped with 6 cores running at 2.40GHz and allowing to run two threads in parallel (hyperthreading). Hence, the number of virtual cores amounts to 24 but we used only 15 cores per machine. In terms of storage, each machine is equipped with a 900GB 7200rpm SATA disk. The machines are connected via a 1GB/s Ethernet Network adapter. We used Spark version 1.2.1 and implemented all experiments in Scala, using version 2.11.6. The Spark setting requires that the total number of cores of the cluster to be specified. Since in our experiments we considered clusters of 5, 10 and 20 machines respectively, we had to set the number of cores to 75, 150 and 300 cores respectively.

6 Experimentation

Since we could not get any query workloads for Wikidata, it was not possible to conduct any experimentation with WARP and the hybrid approach over this data sets. Moreover, since METIS is limited to data sets of half a million edges, it was not possible to handle nHopDB and WARP over LUBM10K. Given the fact that the hybrid system relies on subject hashing, and not METIS, it was possible to conduct this experimentation over LUBM10K for that system.

6.1 Data preparation costs

Figure 1 presents the data preparation processing times for the different systems. As one would expect, the hash-based approaches are between 6 and 30 times faster (depending on the number of partitions) than the graph partition-based approaches. This is mainly due to the fact that METIS runs on a single machine (we have not tested parMETIS, a parallelized version of METIS) while the hash operations are being performed in parallel on the Spark cluster. The evaluation also emphasizes that the hybrid approach presents an interesting compromise between these distribution method families. By evaluating the different processing steps in each of the solutions, we also could find out that, for hash-based approaches, around 15% of processing time is spent on loading the data sets whereas the remaining 85% of time is spent on partitioning the data. For the graph partitioning approaches, 85 to 90% corresponds to the time spent by METIS for creating the partitions; the durations increase with the larger data set sizes. This explains that the time spent by graph partitioning approaches are slightly increasing even when more machines are added. This does not apply for the other solutions where more machines lead to a reduction of the preparation processing time.

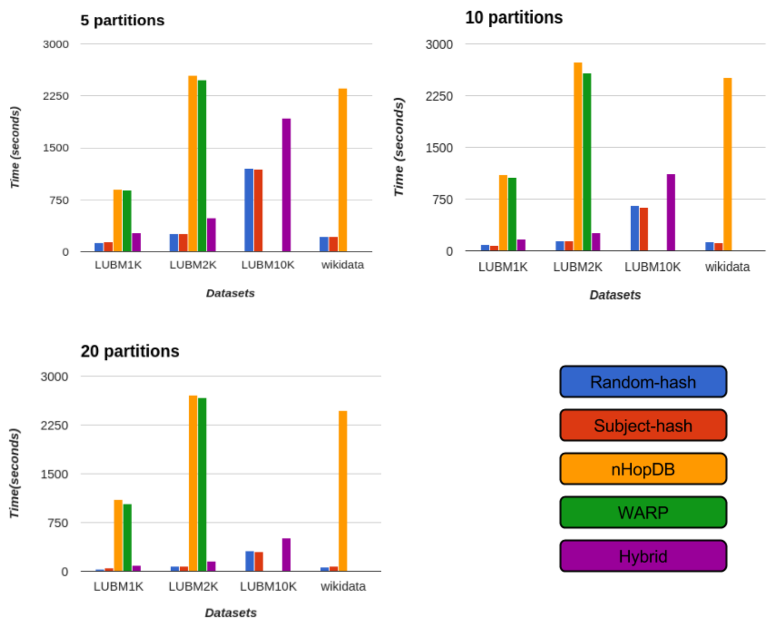


Fig. 1. Data preparation times

6.2 Storage load balancing

Load balancing is an important aspect when distributing data for storage and querying purposes. In Figure 2, we present the standard deviations over all partition sizes (in log scale) for the different implementation. For the graph partitioning-based and hybrid approaches, we only consider the standard deviation of the partition sizes at the end of the partitioning process, $i.e.$, METIS partitioning and n-hop guarantee application.

The two hash-based approaches and the hybrid approach are the best solutions and are close to each other. This is rather obvious since the hash partitioning approaches are concentrating on load balancing while a graph partitioning tries to reduce the number of edges cut during the fragmentation process. The hybrid approach is slightly less balanced due to the application of the WARP query workload-aware strategy. The random-based hashing has 5 to 12% less deviation than subject hashing. This is due to high degree nodes that may increase the size of some partitions. The nHopDB approach is the less efficient graph partitioning solution when considering load balancing. We believe that this is highly related to the structure and the number of queries one considers in the query workload. We consider that further analysis needs to be conducted on real world data sets and query workloads to confirm these nevertheless interesting conclusions.

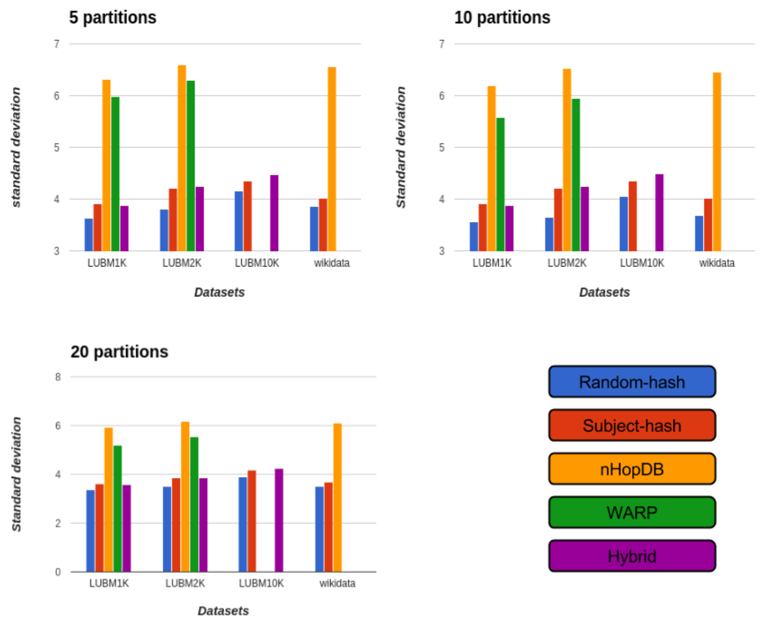


Fig. 2. Standard deviation

6.3 Data replication

Intrinsically, all solutions present some node replications since a given node can be an object in one partition and a subject in another one. This corresponds to the 1-hop guarantee that ensures validity of data, i.e., no triples are lost during the partitioning phase. In this section, we are only interested in triple replication. Only the nHopDB, WARP and hybrid solutions present such replications.

Table 2 provides the replication rates for each of these systems for the LUBM 1K and 2K data sets. Several conclusions can be drawn from this table. First, METIS-based approaches are more efficient than the subject-hashing of the hybrid system. Remember that by minimizing edge cut, a graph partitioner groups the nodes that are close to each other in the input graph. Secondly, the more partitions the cluster contains, the more overall replication one obtains. The n-hop guarantee replicates less than the query workload-aware method of WARP. Finally, we can stress that the replication of the hybrid approach can be considered quite acceptable given the data replication duration highlighted in Section 6.1.

6.4 Query processing

In order to efficiently process local queries and to fairly support performance comparison in a distributed setting, we must use the same computing resources for local and distributed runs. A parallel query runs locally when every machine only has to access its

Part. scheme	nHopDB			WARP			Hybrid		
Data set	5 part.	10 part.	20 part.	5 part.	10 part.	20 part.	5 part.	10 part.	20 part.
LUBM 1K	0.12	0.16	0.17	0.26	0.54	0.57	0.54	1.33	1.84
LUBM 2K	0.12	0.16	0.18	0.34	0.52	0.54	0.54	1.33	1.94

Table 2. Replication rate comparison for three partitioning schemes and three cluster sizes

own partition (inter-partition parallelism). To fully exploit the multi-core machines on which we perform our experiments, it would be interesting to consider not only inter-partition parallelism but intra-partition parallelism exploiting all cores as well. Unfortunately, intra-partition parallelism is not fully supported in Spark since a partition is the unit of data that one core is processing. Thus, to use 15 cores on a machine, we must split a partition into 15 sub partitions. Spark does not allow to specify that such sub-partitions must reside together on the same machine[3]. In the absence of any triple replication, the hash-based solutions are not impacted by this limitation. This is not the case for the systems using replication and where local queries might be evaluated on different partitions. For the two query workload-aware solutions (i.e., WARP and hybrid), we conducted our experiment using a workaround that forces Spark to use only one machine for processing one partition: for each local query, we run Spark with only one slave node. Then we load only the data of one partition and process the query using all the cores of the slave node. To be fair and take into account the possibility that the execution time of a local query might depend on the choice of the partition, we repeat the experiment for every partition and report the maximum response time. The case of nHopDB is more involved and requires to develop a special dedicated query processor, specialized for Spark, to fully benefit from the data fragmentation. In a nutshell, that system would have to combine intra and inter-partition query processors. The former would run for query subgraphs that can run locally and the second one would perform joins over all partitions with retrieved temporary results. Since the topic of this paper concerns the evaluation of distribution strategies, we do not detail the implementation of such a query processor in this work and hence we do not present any results for the nHopDB system.

Table 3 presents the query processing times for our data set. Due to space limitation, we only present the execution time obtained over the 20 partitions experiment. The web site companion (see [6]) highlights that the more partitions are used the more efficient is the query processing. The table clearly highlights that the WARP systems are more efficient than the hash-based solutions. Obviously, the simpler the query, *e.g.* Q4 and Q6, run locally while the others require inter-partition communication. Spark version 1.2.1 `shuffle read` measure indicates the total information exchange (locally on a node and globally over the network) and we could not measure the inter node information communication cost.

[3] We expect that future version of Spark will allow such a control.

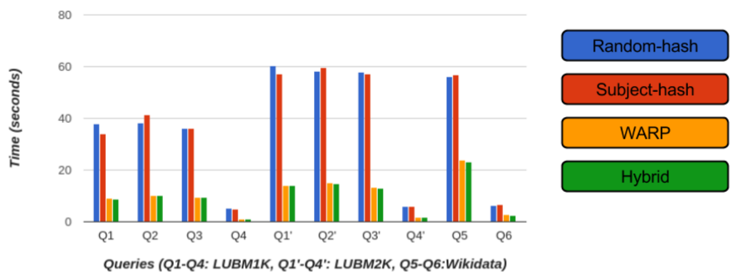


Fig. 3. Query Evaluation on 20 partitions

7 Other related work

Some other interesting works have recently been published on the distribution of RDF data. Systems such as Semstore [24] and SHAPE [17] take some original position.

Instead of using the common query workload, Semstore divides a complete RDF graph into a set of paths which cover all the original graph nodes, possibly with node overlapping between paths. These paths are denoted Rooted Sub Graph (RSG in short) since they are generated starting from nodes with a null in-degree, $i.e.$, roots, to all their possible leaves. A special workaround is used to handle cycles that may occur at the root position, $i.e.$, cycles that are not reachable from any root. The idea is then to regroup these RSG into different partitions. This is obviously a hard problem for which the authors propose an approximated solution. Their solution uses the K-means clustering approach which regroups RSG with common segments together in the same partition. A first limitation of this approach is the high dimensionality of the vectors handled by the K-means algorithm, $i.e.$, the size of any vector corresponds to the number of nodes in the graph. A second limitation is related to the lack of an efficient balancing of the number triples across the partitions. In fact, the system operates at the coarse-grained level of RSG and provides a balancing at this level only. Semstore is finally limited in terms of join patterns. It can efficiently handle S-O (subject-object) and S-S (subject-subject) join patterns but other patterns, such as O-O (object-object) may require inter node communication.

The motivation of the SHAPE system is that graph partitioning approaches do not scale. Just like in our hybrid solution, they propose to replace the graph partitioning step by a hash partitioning one. Then, just like in the nHopDB system, they replicate according to the n-hop guarantee. Hence, they do not consider any query workload and take the risk of inter-partition communication for long chain queries longer than their n-hop guarantee.

8 Conclusions and perspectives

This paper presents an evaluation of different RDF graph distribution methods which are ranging over two important partitioning categories: hashing and graph partitioning.

We have implemented five different approaches over the Apache Spark framework. Due to its efficient main memory usage, Spark is considered to provide better performances than Hadoop's MapReduce. While several RDF stores have been designed on top of Hadoop, we are not aware of any systems running on top of Spark. The main motivation of the experiments is that existing partitioning solutions do not scale gracefully to several billion triples. For instance, the METIS partitioner is limited to less than half a billion triples and SemStore (cf. related works section) relies on K-Means clustering of vectors whose dimension amount to the number of nodes of the data to be processed ($i.e.$, 32 millions in the case of LUBM1K). Computing a distance at such high dimension is currently not possible within Spark, even when using sparse vectors. Moreover, applying a dimension reduction algorithm to all the vectors is not tractable.

The conclusion of our experiment is that basic hash-based partitioning solutions are viable for scalable RDF management: they come at no preparation cost, $i.e.$ only require to load the triples into the right machine, and are fully supported by the underlying Spark system. As emphasized by our experimentation, Spark scales out to several billion triples by simply adding extra machines. Nevertheless, without any replication, these systems may hinder availability and reduce the parallelism of query processing. They also involve a lot of network communications for complex queries which require to retrieve data from many partitions. Nonetheless, by making intensive use of main memory, we believe that Spark provides a high potential for these systems. Clearly, with the measures we have obtained in this evaluation, we can stress that if one needs a fast access to large RDF data sets and is, to some extent, ready to sacrifice the performance of processing complex query patterns then the hash-based solution over Spark is a good compromise.

Concerning the nHopDB and WARP approaches, we consider that using a graph partitioning system like METIS has an important drawback due to the performance limitations. Based in these observations, we investigated the hybrid candidate solution which does not involve a heavy preparation step and retains the interesting query workload aware replication strategy. This approach may be particularly interesting for data warehouses where the most common queries (materialized views) are well identified. With this hybrid solution we may get the best of worlds, the experiments clearly emphasize that the replication overhead compared to the pure WARP approach is marginal but the gain in data preparation is quite important.

Concerning Spark, we highlighted that it can process distributed RDF queries efficiently. Moreover, the system can be used for the two main steps, data preparation and query processing, in an homogeneous way. Rewriting SPARQL queries into the Scala language (supported by Spark) is rather easy and we consider that there is still much room for optimization. The next versions of Spark which are supposed to provide more feedback on data exchange over the network should help fine-tune our experiments and design a complete production-ready system.

References

1. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.

2. M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proc. 13th International World Wide Web Conference*, New York City, NY, USA, May 2004.
3. O. Curé and G. Blin. *RDF Database Systems, 1st Edition*. Morgan Kaufmann, Nov. 2014.
4. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.
5. O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
6. http://webia.lip6.fr/∼baazizi/research/iswc2015eval/expe.html.
7. C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 175–181, 1982.
8. L. Galarraga, K. Hose, and R. Schenkel. Partout: A distributed engine for efficient RDF processing. *CoRR*, abs/1212.5636, 2012.
9. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
10. S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD 2014, USA, June 22-27, 2014*, pages 289–300, 2014.
11. M. Hammoud, D. A. Rabbou, R. Nouri, S. Beheshti, and S. Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.
12. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, pages 211–224, 2007.
13. K. Hose and R. Schenkel. WARP: workload-aware replication and partitioning for RDF. In *Workshops Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013*, pages 1–6, 2013.
14. J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.
15. M. F. Husain, J. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Transactions on Knowledge and Data Engineering*, 23:1312–1327, 2011.
16. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.
17. K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.
18. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, USA*, pages 604–615, 2002.
19. T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
20. K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, pages 4:1–4:5, New York, NY, USA, 2010. ACM.

21. P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.
22. M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
23. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.
24. B. Wu, Y. Zhou, P. Yuan, H. Jin, and L. Liu. Semstore: A semantic-preserving distributed rdf triple store. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 509–518, New York, NY, USA, 2014. ACM.
25. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.
26. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.
27. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

A Appendix : Query workload

Q1: (property path)
```
SELECT ?x ?y ?z WHERE { ?x lubm:advisor ?y.
?y lubm:worksFor ?z. ?z lubm:subOrganisation ?t. }
```
Q2: (typed, star and property path)
```
SELECT ?x ?y ?z WHERE {
?x rdf:type lubm:GraduateStudent.
?y rdf:type lubm:University.
?z rdf:type lubm:Department. ?x lubm:memberOf ?z.
?z lubm:subOrganizationOf ?y.
?x lubm:undergraduateDegreeFrom ?y }
```
Q3: (typed, star and property path)
```
SELECT ?x ?y ?z WHERE { ?x rdf:type lubm:Student.
?y rdf:type lubm:Faculty. ?z rdf:type lubm:Course.
?x lubm:advisor ?y. ?y lubm:teacherOf ?z.
?x lubm:takesCourse ?z }
```
Q4: (typed, property path)
```
SELECT ?x ?y WHERE { ?x rdf:type lubm:Chair.
?y rdf:type lubm:Department. ?x lubm:worksFor ?y.
?y lubm:subOrganizationOf <http://www.University0.edu>
}
```
Q5: (property path)
```
SELECT ?x ?y ?z WHERE { ?x entity:P131s ?y.
?y entity:P961v ?z. ?z entity:P704s ?w }
```
Q6: (star)
```
SELECT ?x ?y ?z WHERE { ?x entity:P39v ?y.
?x entity:P580q ?z. ?x rdf:type ?w }
```

Reifying RDF: What Works Well With Wikidata?

Daniel Hernández[1], Aidan Hogan[1], and Markus Krötzsch[2]

[1] Department of Computer Science, University of Chile
[2] Technische Universität Dresden, Germany

Abstract. In this paper, we compare various options for reifying RDF triples. We are motivated by the goal of representing Wikidata as RDF, which would allow legacy Semantic Web languages, techniques and tools – for example, SPARQL engines – to be used for Wikidata. However, Wikidata annotates statements with qualifiers and references, which require some notion of reification to model in RDF. We thus investigate four such options: (1) standard reification, (2) n-ary relations, (3) singleton properties, and (4) named graphs. Taking a recent dump of Wikidata, we generate the four RDF datasets pertaining to each model and discuss high-level aspects relating to data sizes, etc. To empirically compare the effect of the different models on query times, we collect a set of benchmark queries with four model-specific versions of each query. We present the results of running these queries against five popular SPARQL implementations: 4store, BlazeGraph, GraphDB, Jena TDB and Virtuoso.

1 Introduction

Wikidata is a collaboratively edited knowledge-base under development by the Wikimedia foundation whose aim is to curate and represent the factual information of Wikipedia (across all languages) in an interoperable, machine-readable format [20]. Until now, such factual information has been embedded within millions of Wikipedia articles spanning hundreds of languages, often with a high degree of overlap. Although initiatives like DBpedia [15] and YAGO [13] have generated influential knowledge-bases by applying custom extraction frameworks over Wikipedia, the often ad hoc way in which structured data is embedded in Wikipedia limits the amount of data that can be cleanly captured. Likewise, when information is gathered from multiple articles or multiple language versions of Wikipedia, the results may not always be coherent: facts that are mirrored in multiple places must be manually curated and updated by human editors, meaning that they may not always correspond at a given point in time.

Therefore, by allowing human editors to collaboratively add, edit and curate a centralised, structured knowledge-base directly, the goal of Wikidata is to keep a single consistent version of factual data relevant to Wikipedia. The resulting knowledge-base is not only useful to Wikipedia – for example, for automatically generating articles that list entities conforming to a certain restriction (e.g., female heads of state) or for generating infobox data consistently across all languages – but also to the Web community in general. Since the launch of Wikidata in October 2012, more than 80 thousand editors have contributed information on 18 million entities (data items and properties). In comparison, English Wikipedia has around 6 million pages, 4.5 million of which are considered proper articles. As of July 2015, Wikidata has gathered over 65 million statements.

One of the next goals of the Wikidata project is to explore methods by which the public can query the knowledge-base. The Wikidata developers are currently investigating the use of RDF as an exchange format for Wikidata with SPARQL query functionality. Indeed, the factual information of Wikidata corresponds quite closely with the RDF data model, where the main data item (entity) can be viewed as the subject of a triple and the attribute–value pairs associated with that item can be mapped naturally to predicates and objects associated with the subject. However, Wikidata also allows editors to annotate attribute–value pairs with additional information, such as qualifiers and references. Qualifiers provide context for the validity of the statement in question, for example providing a time period during which the statement was true. References point to authoritative sources from which the statement can be verified. About half of the statements in Wikidata (32.5 million) already provide a reference, and it is an important goal of the project to further increase this number.

Hence, to represent Wikidata in RDF while capturing meta-information such as qualifiers and references, we need some way in RDF to describe the RDF triples themselves (which herein we will refer to as "*reification*" in the general sense of the term, as distinct from the specific proposal for reification defined in the 2004 RDF standard [3], which we refer to as "*standard reification*").

In relation to Wikidata, we need a method that is compatible with existing Semantic Web standards and tools, and that does not consider the domain of triple annotation as fixed in any way: in other words, it does not fix the domain of annotations to time, or provenance, or so forth [22]. With respect to general methods for reification within RDF, we identify four main options:

standard reification (SR) whereby an RDF resource is used to denote the triple itself, denoting its subject, predicate and object as attributes and allowing additional meta-information to be added [16,4].

***n*-ary relations (NR)** whereby an intermediate resource is used to denote the relationship, allowing it to be annotated with meta-information [16,8].

singleton properties (SP) whereby a predicate unique to the statement is created, which can be linked to the high-level predicate indicating the relationship, and onto which can be added additional meta-information [18].

Named Graphs (NG) whereby triples (or sets thereof) can be identified in a fourth field using, e.g., an IRI, onto which meta-information is added [10,5].

Any of these four options would allow the qualified statements in the Wikidata knowledge-base to be represented and processed using current Semantic Web norms. In fact, Erxleben et al. [8] previously proposed an RDF representation of the Wikidata knowledge-base using a form of *n*-ary relations. It is important to note that while the first three formats rely solely on the core RDF model, Named Graphs represents an extension of the traditional triple model, adding a fourth element; however, the notion of Named Graphs is well-supported in the SPARQL standard [10], and as "RDF Datasets" in RDF 1.1 [5].

Thus arises the core question tackled in this paper: *what are the relative strengths and weaknesses of each of the four formats*? We are particularly interested in exploring this question quantitatively with respect to Wikidata. We thus take a recent dump of Wikidata and create four RDF datasets: one for each of the formats listed above. The focus of this preliminary paper is to gain empirical insights on how these formats affect query execution times for off-the-shelf tools. We thus (attempt to) load these four datasets into five popular SPARQL engines – namely 4store [9], BlazeGraph (formerly BigData) [19], GraphDB (formerly (Big)OWLIM) [2], Jena TDB [21], and Virtuoso [7] – and apply four versions of a query benchmark containing 14 queries to each dataset in each engine.

2 The Wikidata Data-model

Figure 1a provides an example statement taken from Wikidata describing the entity Abraham Lincoln. We show internal identifiers in grey, where those beginning with Q refer to entities, and those referring to P refer to properties. These identifiers map to IRIs, where information about that entity or relationship can be found. All entities and relationships are also associated with labels, where the English versions are shown for readability. Values of properties may also be datatype literals, as exemplified with the dates.

The snippet contains a primary relation, with Abraham Lincoln as subject, position held as predicate, and President of the United States of America as object. Such binary relations are naturally representable in RDF. However, the statement is also associated with some *qualifiers* and their *values*. Qualifiers are property terms such as start time, follows, etc., whose values may scope the validity of the statement and/or provide additional context. Additionally, statements are often associated with one or more *references* that support the claims and with a *rank* that marks the most important statements for a given property. The details are not relevant to our research: we can treat references and ranks as special types of qualifiers. We use the term *statement* to refer to a primary relation and its associated qualifications; e.g., Fig. 1a illustrates a single statement.

Conceptually, one could view Wikidata as a "Property Graph": a directed labelled graph where edges themselves can have attributes and values [11,6]. A related idea would be to consider Wikidata as consisting of *quins* of the form (s, p, o, q, v), where (s, p, o) refers to the *primary relation*, q is a *qualifier property*, and v is a *qualifier value* [12]. Referring to Fig. 1a again, we could encode a quin (`:Q91`, `:P39`, `:Q11696`, `:P580`, `"1861/03/14"^^xsd:date`), which states that Abraham Lincoln had relation position held to President of the United States of America under the qualifier property start time with the qualifier value 4 March 1861. All quins with a common primary relation would constitute a statement. However, quins of this form are not a suitable format for Wikidata since a given primary relation may be associated with different groupings of qualifiers. For example, Grover Cleveland was President of the United States for two non-consecutive terms (i.e., with different start and end times, different predecessors and successors). In Wikidata, this is represented as two separate statements whose primary relations are both identical, but where the qualifiers (start time, end time, follows, followed by) differ. For this reason, reification schemes based conceptually on quins – such as RDF* [12,11] – may not be directly suitable for Wikidata.

A tempting fix might be to add an additional column and represent Wikidata using sextuples of the form (s, p, o, q, v, i) where i is an added statement identifier. Thus in the case of Grover Cleveland, his two non-consecutive terms would be represented as two statements with two distinct statement identifiers. While in principle sextuples would be sufficient, in practice (i) the relation itself may contain NULLs, since some statements do not currently have qualifiers or may not even support qualifiers (as is the case with labels, for example), (ii) qualifier values may themselves be complex and require some description: for example, dates may be associated with time precisions or calendars.[3]

For this reason, we propose to view Wikidata conceptually in terms of two tables: one containing quads of the form (s, p, o, i) where (s, p, o) is a primary relation and i is an identifier for that statement; the other a triple table storing (i) primary relations that can never be qualified (e.g., labels) and thus do not need to be identified, (ii) triples of the form (i, q, v) that specify the qualifiers associated to a statement, and (iii) triples of the form (v, x, y) that further describe the properties of qualifier values. Table 1 provides an example that encodes some of the data seen in Fig. 1a – as well as some further type information not shown – into two such tables: the quads table on the left encodes qualifiable primary relations with an identifier, and the triples table on the right encodes (i) qualifications using the statement identifiers, (ii) non-qualifiable primary relations, such as those that specify labels, and (iii) type information for complex values, such as to provide a precision, calendar, etc.

Compared to sextuples, the quad/triple schema only costs one additional tuple per statement, will lead to dense instances (even if some qualifiable primary relations are not currently qualified), and will not repeat the primary relation for each qualifier; conversely, the quad/triple schema may require more joins for certain query patterns (e.g., find primary relations with a follows qualifier).

Likewise, the quad/triple schema is quite close to an RDF-compatible encoding. As per Fig. 1b, the triples from Table 1 are already an RDF graph; we can thus focus on encoding quads of the form (s, p, o, i) in RDF.

3 From Higher Arity Data to RDF (and back)

The question now is: how can we represent the statements of Wikidata as triples in RDF? Furthermore: how many triples would we *need* per statement? And how might we know for certain that we don't lose something in the translation?

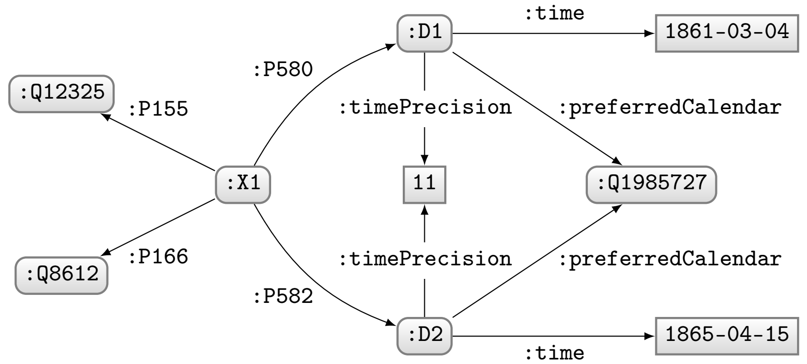
The transformation from Wikidata to RDF can be seen as an instance of schema translation, where these questions then directly relate to the area of

[3] See `https://www.wikidata.org/wiki/Special:ListDatatypes`; retr. 2015/07/11.

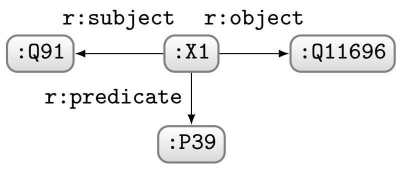
Abraham Lincoln [Q91]

position held [P39]	President of the United States of America [Q11696]
start time [P580]	"4 March 1861"
end time [P582]	"15 April 1865"
follows [P155]	James Buchanan [Q12325]
followed by [P156]	Andrew Johnson [Q8612]

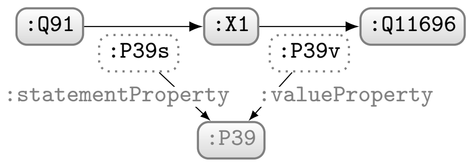
(a) Raw Wikidata format



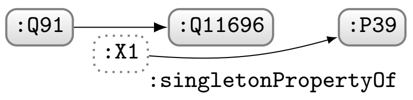
(b) Qualifier information common to all formats



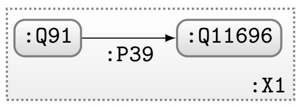
(c) Standard reification



(d) *n*-ary relations



(e) Singleton properties



(f) Named Graphs

Fig. 1: Reification examples

Table 1: Using quads and triples to encode Wikidata

Quads

s	p	o	i
`:Q91`	`:P39`	`:Q11696`	`:X1`
...	...	...	...

Triples

s	p	o
`:X1`	`:P580`	`:D1`
`:X1`	`:P582`	`:D2`
`:X1`	`:P155`	`:Q12325`
`:X1`	`:P156`	`:Q8612`
...	...	...
`:D1`	`:time`	`"1861-03-04"^^xsd:date`
`:D1`	`:timePrecision`	`"11"`
`:D1`	`:preferredCalendar`	`:Q1985727`
...	...	...
`:Q91`	`rdfs:label`	`"Abraham Lincoln"@en`
...	...	...

relative information capacity in the database community [14,17], which studies how one can translate from one database schema to another, and what sorts of guarantees can be made based on such a translation. Miller et al. [17] relate some of the theoretical notions of information capacity to practical guarantees for common schema translation tasks. In this view, the task of translating from Wikidata to RDF is a unidirectional scenario: we want to query a Wikidata instance through an RDF view without loss of information, and to recover the Wikidata instance from the RDF view, but we do not require, e.g., updates on the RDF view to be reflected in Wikidata. We therefore need a transformation whereby the RDF view dominates the Wikidata schema, meaning that the transformation must map a unique instance of Wikidata to a unique RDF view.

We can formalise this by considering an instance of Wikidata as a database with the schema shown in Table 1. We require that any instance of Wikidata can be mapped to a RDF graph, and that any conjunctive query (CQ; select-project-join query in SQL) over the Wikidata instance can be translated to a conjunctive query over the RDF graph that returns the same answers. We call such a translation *query dominating*. We do not further restrict how translations are to be specified so long as they are well-defined and computable.

A query dominating translation of a relation of arity 4 (R_4) to a unique instance of a relation of arity 3 (R_3) must lead to a higher number of tuples in the target instance. In general, we can always achieve this by encoding a quad into four triples of the form (s, p_y, o_z), where s is a unique ID for the quad, p_y denotes a position in the quad (where $1 \leq y \leq 4$), and o_z is the term appearing in that position of the quad in R_4.

Example 1. *The quad* `:Q91 :P39 :Q11696 :X1` *can be mapped to four triples:*

```
:S1 :P1 :Q91
:S1 :P2 :P39
:S1 :P3 :Q11696
:S1 :P4 :X1
```

Any conjunctive query over any set of quads can be translated into a conjunctive query over the corresponding triple instance that returns the same answer: for each tuple in the former query, add four tuples to the latter query with a fresh common subject variable, with `:P1` ... `:P4` as predicate, and with the corresponding terms from the sextuple (be they constants or variables) as objects. The fresh subject variables can be made existential/projected away. □

With this scheme, we require $4k$ triples to encode k quads. This encoding can be generalised to encode database tables of arbitrary arity, which is essentially the approach taken by the *Direct Mapping* of relational databases to RDF [1].

Quad encodings that use fewer than four triples usually require additional assumptions. The above encoding requires the availability of an unlimited amount of identifiers, which is always given in RDF, where there is an infinite supply of IRIs. However, the technique does not assume that auxiliary identifiers such as `:S1` or `:P4` do not occur elsewhere: even if these IRIs are used in the given set of quads, their use in the object position of triples would not cause any confusion. If we make the additional assumption that some "reserved" identifiers are not used in the input quad data, we can find encodings with fewer triples per quad.

Example 2. *If we assume that IRIs* `:P1` *and* `:P2` *are not used in the input instance, the quad of Example 1 can be encoded in the following three triples:*

```
:S1 :P1     :Q91
:S1 :P2     :P39
:S1 :Q11696 :X1
```

The translation is not faithful when the initial assumption is violated. For example, the encoding of the quad `:Q91 :P39 :P1 :Q92` *would contain triples* `:S2 :P1 :Q91` *and* `:S2 :P1 :Q92`*, which would be ambiguous.* □

The assumption of some reserved IRIs is still viable in practice, and indeed three of the encoding approaches we look at in the following section assume some reserved vocabulary to be available. Using suitable naming strategies, one can prevent ambiguities. Other domain-specific assumptions are also sometimes used to define suitable encodings. For example, when constructing quads from Wikidata, the statement identifiers that are the fourth component of each quad functionally determine the first three components, and this can be used to simplify the encoding. We will highlight these assumptions as appropriate.

4 Existing Reification Approaches

In this section, we discuss how legacy reification-style approaches can be leveraged to model Wikidata in RDF, where we have seen that all that remains is to model quads in RDF. We also discuss the natural option of using named graphs, which support quads directly. The various approaches are illustrated in Fig. 1, where 1b shows the qualifier information common to all approaches, i.e., the triple data, while 1c–1f show alternative encodings of quads.

Standard Reification The first approach we look at is standard RDF reification [16,4], where a resource is used to denote the statement, and where additional information about the statement can be added. The scheme is depicted in Fig. 1c. To represent a quad of the form (s, p, o, i), we add the following triples: $(i,$ `r:subject`$, s)$, $(i,$ `r:predicate`$, p)$, $(i,$ `r:object`$, o)$, where `r:` is the RDF vocabulary namespace. We omit the redundant declaration as type `r:Statement`, which can be inferred from the domain of `r:subject`. Moreover, we simplify the encoding by using the Wikidata statement identifier as subject, rather than using a blank node. We can therefore represent n quadruples with $3n$ triples.

***n*-ary Relation** The second approach is to use an n-ary relation style of modelling, where a resource is used to identify a relationship. Such a scheme is depicted in Fig. 1d, which follows the proposal by Erxleben et al. [8]. Instead of stating that a subject has a given value, the model states that the subject is involved in a relationship, and that that relationship has a value and some qualifiers. The `:subjectProperty` and `:valueProperty` edges are important to be able to query for the original name of the property holding between a given subject and object.[4] For identifying the relationship, we can simply use the statement identifier. To represent a quadruple of the form (s, p, o, i), we must add the triples (s, p_s, i), (i, p_v, o), $(p_v,$ `:valueProperty`$, p)$, $(p_s,$ `:statementProperty`$, p)$, where p_v and p_s are fresh properties created from p. To represent n quadruples of the form (s, p, o, i), with m unique values for p, we thus need $2(n+m)$ triples. Note that for the translation to be query dominating, we must assume that predicates such as `:P39s` and `:P39v` in Fig. 1c, as well as the reserved terms `:statementProperty` and `:valueProperty`, do not appear in the input.

Singleton Properties Originally proposed by Nguyen et al. [18], the core idea behind singleton properties is to create a property that is only used for a single statement, which can then be used to annotate more information about the statement. The idea is captured in Fig. 1e. To represent a quadruple of the form (s, p, o, i), we must add the triples (s, i, o), $(i,$ `:singletonPropertyOf`$, p)$. Thus to represent n quadruples, we need $2n$ triples, making this the most concise scheme so far. To be query dominating, we must assume that the term `:singletonPropertyOf` cannot appear as a statement identifier.

Named Graphs Unlike the previous three schemes, Named Graphs extends the RDF triple model and considers sets of pairs of the form (G, n) where G is an RDF graph and n is an IRI (or a blank node in some cases, or can even be omitted for a *default graph*). We can flatten this representation by taking the union over $G \times \{n\}$ for each such pair, resulting in quadruples. Thus we can encode a quadruple (s, p, o, i) directly using N-Quads, as illustrated in Fig. 1f.

[4] Referring to Fig. 1c, another option to save some triples might be to use the original property `:P39` (position held) instead of `:P39s` or `:P39v`, but this could be conceptually ugly since, e.g., if we replaced `:P39s`, the resulting triple would be effectively stating that (Abraham Lincoln,position held,[*a statement identifier*]).

Table 2: Number of triples needed to model quads ($n = 57,088,184$, $p = 1,311$)

Schema:	**sr** ($3n$)	**nr** ($2(n+p)$)	**sp** ($2n$)	**ng** (n)
Tuples:	171,264,552	114,178,990	114,176,368	57,088,184

Other possibilities? Having introduced the most well-known options for reification, one may ask if these are all the reasonable alternatives for representing quadruples of the form (s, p, o, i) – where i functionally determines (s, p, o) – in a manner compatible with the RDF standards. As demonstrated by singleton properties, we can encode such quads into two triples, where i appears somewhere in both triples, and where s, p and o each appear in one of the four remaining positions, and where a reserved term is used to fill the last position. This gives us 108 possible schemes[5] that use two triples to represent a quad in a similar manner to the singleton properties proposal. Just to take one example, we could model such a quad in two triples as $(i,$ `r:subject`$, s), (i, p, o)$—an abbreviated form of standard reification. As before, we should assume that the properties p and `r:subject` are distinct from qualifier properties. Likewise, if we are not so concerned with conciseness and allow a third triple, the possibilities increase further. To reiterate, our current goal is to evaluate existing, well-known proposals, but we wish to mention that many other such possibilities do exist in theory.

5 SPARQL Querying Experiments

We have looked at four well-known approaches to annotate triples, in terms of how they are formed, what assumptions they make, and how many triples they require. In this section, we aim to see how these proposals work in practice, particularly in the context of querying Wikidata. Experiments were run on an Intel E5-2407 Quad-Core 2.2GHz machine with a standard SATA hard-drive and 32 GB of RAM. More details about the configuration of these experiments are available from `http://users.dcc.uchile.cl/~dhernand/wrdf/`.

To start with, we took the RDF export of Wikidata from Erxleben et al. [8] (2015-02-23), which was natively in an n-ary relation style format, and built the equivalent data for all four datasets. The number of triples common to all formats was 237.6 million. With respect to representing the quads, Table 2 provides a breakdown of the number of output tuples for each model.

[5] There are 3! possibilities for where i appears in the two triples: ss, pp, oo, sp, so, po. For the latter three configurations, all four remaining slots are distinguished so we have 4! ways to slot in the last four terms. For the former three configurations, both triples are thus far identical, so we only have half the slots, making 4×3 permutations. Putting it all together, $3 \times 4! + 3 \times 4 \times 3 = 108$.

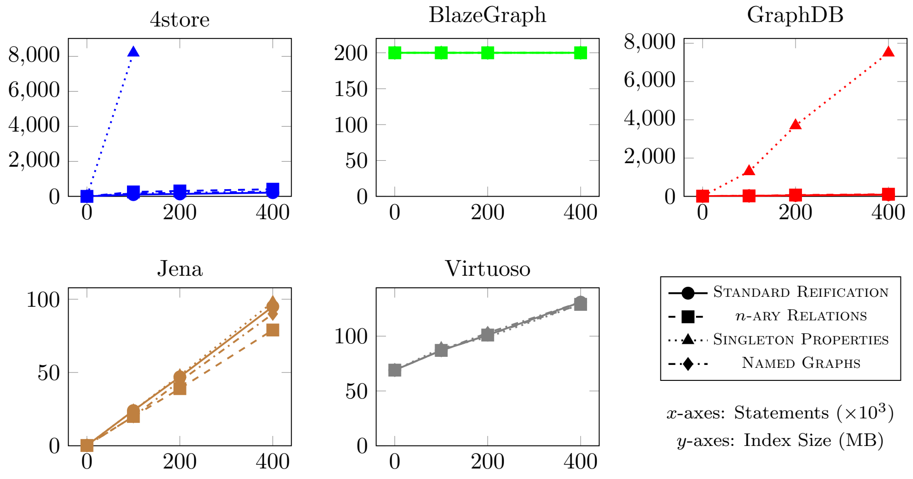


Fig. 2: Growth in index sizes for first 400,000 statements

Loading data: We selected five RDF engines for experiments: 4store, BlazeGraph, GraphDB, Jena and Virtuoso. The first step was to load the four datasets for the four models into each engine. We immediately started encountering problems with some of the engines. To quantify these issues, we created three collections of 100,000, 200,000, and 400,000 raw statements and converted them into the four models.[6] We then tried to load these twelve files into each engine. The resulting growth in on-disk index sizes is illustrated in Figure 2 (measured from the database directory), where we see that: (i) even though different models lead to different triple counts, index sizes were often nearly identical: we believe that since the entropy of the data is quite similar, compression manages to factor out the redundant repetitions in the models; (ii) some of the indexes start with some space allocated, where in fact for BlazeGraph, the initial allocation of disk space (200MB) was not affected by the data loads; (iii) 4store and GraphDB both ran into problems when loading singleton properties, where it seems the indexing schemes used assume a low number of unique predicates.[7] With respect to point (iii), given that even small samples lead to blow-ups in index sizes, we decided not to proceed with indexing the singleton properties dataset in 4store or GraphDB. While later loading the full named graphs dataset, 4store slowed to loading 24 triples/second; we thus also had to kill that index load.[8]

[6] We do not report the times for full index builds since, due to time constraints, we often ran these in parallel uncontrolled settings.

[7] In the case of 4store, for example, in the database directory, two new files were created for each predicate.

[8] See `https://groups.google.com/forum/#!topic/4store-support/uv8yHrb-ng4`; retr. 2015/07/11.

Benchmark queries: From two online lists of test-case SPARQL queries, we selected a total of 14 benchmark queries.[9] These are listed in Table 3; since we need to create four versions of each query for each reification model, we use an abstract quad syntax where necessary, which will be expanded in a model-specific way such that the queries will return the same answers over each model. An example of a quad in the abstract syntax and its four expansions is provided in Table 4. In a similar manner, the 14 queries of Table 3 are used to generate four equivalent query benchmarks for testing, making a total of 56 queries.

In terms of the queries themselves, they exhibit a variety of query features and number/type of joins; some involve qualifier information while some do not. Some of the queries are related; for example, Q1 and Q2 both ask for information about US presidents, and Q4 and Q5 both ask about female mayors. Importantly, Q4 and Q5 both require use of a SPARQL property path (`:P31/:P279*`), which we cannot capture appropriately in the abstract syntax. In fact, these property paths cannot be expressed in either the singleton properties model or the standard reification model; they can only be expressed in the n-ary relation model (`:P31s/:P31v/(:P279s/:P279v)*`), and the named graph model (`:P31/:P279*`) *assuming* the default graph can be set as the union of all graphs (since one cannot do property paths across graphs in SPARQL, only within graphs).

Query results: For each engine and each model, we ran the queries sequentially (Q1–14) five times on a cold index. Since the engines had varying degrees of caching behaviour after the first run – which is not the focus of this paper – we present times for the first "cold" run of each query.[10] Since we aim to run $14 \times 4 \times 5 \times 5 = 1,400$ query executions, to keep the experiment duration manageable, all engines were configured for a timeout of 60 seconds. Since different engines interpret timeouts differently (e.g., connection timeouts, overall timeouts, etc.), we considered any query taking longer than 60 seconds to run as a timeout. We also briefly inspected results to see that they corresponded with equivalent runs, ruling queries that returned truncated results as failed executions.[11]

The query times for all five engines and four models are reported in Figure 3, where the y-axis is in log scale from 100 ms to 60,000 ms (the timeout) in all cases for ease of comparison. Query times are not shown in cases where the query could not be run (for example, the index could not be built as discussed previously, or property-paths could not be specified for that model, or in the case of 4store, certain query features were not supported), or where the query failed (with a timeout, a partial result, or a server exception). We see that in terms of engines, Virtuoso provides the most reliable/performant results across all models. Jena failed to return answers for singleton properties, timing-out on all queries (we expect some aspect of the query processing does not perform well

[9] `https://www.mediawiki.org/wiki/Wikibase/Indexing/SPARQL_Query_Examples` and `http://wikidata.metaphacts.com/resource/Samples`

[10] Data for other runs are available from the web-page linked earlier.

[11] The results for queries such as Q5, Q7 and Q14 may (validly) return different answers for different executions due to use of `LIMIT` without an explicit order.

Table 3: Evaluation queries in abstract syntax

#**Q1**: US presidents and their wives

```
SELECT ?up ?w ?l ?wl WHERE { <:Q30 :P6 ?up _:i1> . <?up :P26 ?w ?i2> . OPTIONAL {
   ?up rs:label ?l . ?w rs:label ?wl . FILTER(lang(?l) = "en" && lang(?wl) = "en") } }
```

#**Q2**: US presidents and causes of death

```
SELECT ?h ?c ?hl ?cl WHERE { <?h :P39 :Q11696 ?i1> .  <?h :P509 ?c ?i2> . OPTIONAL {
 ?h rs:label ?hl . ?c rs:label ?cl . FILTER(lang(?hl) = "en" && lang(?cl) = "en") } }
```

#**Q3**: People born before 1880 with no death date

```
SELECT ?h ?date WHERE { <?h :P31 :Q5 ?i1> . <?h :P569 ?dateS ?i2> . ?dateS :time ?date .
 FILTER NOT EXISTS { ?h :P570s [ :P570v ?d ] . }
 FILTER (datatype(?date) = xsd:date && ?date < "1880-01-01Z"^^xsd:date) } LIMIT 100
```

#**Q4**: Cities with female mayors ordered by population

```
SELECT DISTINCT ?city ?citylabel ?mayorlabel (MAX(?pop) AS ?max_pop) WHERE {
 ?city :P31/:P279* :Q515 . <?city :P6 ?mayor ?i1> . FILTER NOT EXISTS { ?i1 :P582q ?x }
 <?mayor :P21 :Q6581072 _:i2> .  <?city :P1082 ?pop _:i3> .  ?pop :numericValue ?pop .
 OPTIONAL { ?city rs:label ?citylabel . FILTER ( LANG(?citylabel) = "en" ) }
 OPTIONAL { ?mayor rs:label ?mayorlabel . FILTER ( LANG(?mayorlabel) = "en" ) } }
GROUP BY ?city ?citylabel ?mayorlabel ORDER BY DESC(?max_pop) LIMIT 10
```

#**Q5**: Countries ordered by number of female city mayors

```
SELECT ?country ?label (COUNT(*) as ?count) WHERE { ?city :P31/:P279* :Q515 .
 <?city :P6 ?mayor ?i1> . FILTER NOT EXISTS { ?i1 :P582q ?x }
 <?mayor :P21 :Q6581072 ?i2> .  <?city :P17 ?country ?i3> .  ?pop :numericValue ?pop .
 OPTIONAL { ?country rs:label ?label .  FILTER ( LANG(?label) = "en" ) } }
GROUP BY ?country ?label ORDER BY DESC(?count) LIMIT 100
```

#**Q6**: US states ordered by number of neighbouring states

```
SELECT ?state ?label ?borders WHERE { { SELECT ?state (COUNT(?neigh) as ?borders)
   WHERE { <?state :P31 :Q35657 ?i1> . <?neigh :P47 ?state _:i2> .
     <?neigh :P31 :Q35657 ?i3 > . } GROUP BY ?state }
 OPTIONAL { ?state rs:label ?label . FILTER(lang(?label) = "en") } } ORDER BY DESC(?borders)
```

#**Q7**: People whose birthday is “today”

```
SELECT DISTINCT ?entity ?year  WHERE { <?entityS :P569 ?value ?i1> . ?value :time ?date .
 ?entityS rs:label ?entity . FILTER(lang(?entity)="en")
 FILTER(date(?date)=month(now()) && date(?date)=day(now())) } LIMIT 10
```

#**Q8**: All property–value pairs for Douglas Adams

```
SELECT ?property ?value  WHERE {  <:Q42 ?property ?value ?i> }
```

#**Q9**: Populations of Berlin, ordered by least recent

```
SELECT ?pop ?time WHERE { <:Q64 wd:P1082 ?popS ?i> . ?popS :numericValue ?pop .
 ?i wd:P585q [ :time ?time ] . } ORDER BY (?time)
```

#**Q10**: Countries without an end-date

```
SELECT ?country ?countryName WHERE { <?country wd:P31 wd:Q3624078 ?i> .
 FILTER NOT EXISTS { ?g :P582q ?endDate } ?country rdfs:label ?countryName .
 FILTER(lang(?countryName)="en") }
```

#**Q11**: US Presidents and their terms, ordered by start-date

```
SELECT ?president ?start ?end WHERE { <:Q30 :P6 ?president ?i > .
 ?g :P580q [ :time ?start ] ; :P582q [ :time ?end ] . }  ORDER BY (?start)
```

#**Q12**: All qualifier properties used with "head of government" property

```
SELECT DISTINCT ?q_p WHERE { <?s :P6 ?o ?i > . ?g ?q_p ?q_v . }
```

#**Q13**: Current countries ordered by most neighbours

```
SELECT ?countryName (COUNT (DISTINCT ?neighbor) AS ?neighbors) WHERE {
 <?country :P31 :Q3624078 ?i1> . FILTER NOT EXISTS { ?i1 :P582 ?endDate }
 ?country rs:label ?countryName FILTER(lang(?countryName)="en") OPTIONAL {
   <?country :P47 ?neighbor ?i2 > <?neighbor :P31 :Q3624078 ?i3> .
   FILTER NOT EXISTS { ?i3 :P582q ?endDate2 } } }
GROUP BY(?countryName) ORDER BY DESC(?neighbors)
```

#**Q14**: People who have Wikidata accounts

```
 SELECT ?person ?name ?uname WHERE { <?person :P553 wd:Q52 ?i > .
 ?i :P554q ?uname . ?person rs:label ?name . FILTER(LANG(?name) = "en") . } LIMIT 100
```

Table 4: Expansion of abstract syntax quad for getting US presidents

ABSTRACT SYNTAX:	`<:Q30 :P6 ?up ?i> .`
STD. REIFICATION:	`?i r:subject :Q30 ; r:predicate :P6 ; r:object ?up .`
n-ARY RELATIONS:	`:Q30 :P6s ?i . ?i :P6v ?up .`
SING. PROPERTIES:	`:Q30 ?i ?up . ?i1 r:singletonPropertyOf :P6 .`
NAMED GRAPHS:	`GRAPH ?i { :Q30 :P6 ?up . }`

assuming many unique predicates). We see that both BlazeGraph and GraphDB managed to process most of the queries for the indexes we could build, but with longer runtimes than Virtuoso. In general, 4store struggled with the benchmark and returned few valid responses in the allotted time.

Returning to our focus in terms of comparing the four reification models, Table 5 provides a summary of how the models ranked for each engine and overall. For a given engine and query, we look at which model performed best, counting the number of firsts, seconds, thirds, fourths, failures (FA) and cases where the query could not be run (NR). For example, referring back to Figure 3, we see that for Virtuoso, **Q1**, the fastest models in order were standard reification, named graphs, singleton properties, n-ary relations. Thus, in Table 5, under Virtuoso, we add a one to standard reification in the 1st column, a one to named graphs in the 2st column, and so forth. Along these lines, for example, the score of 4 for singleton-properties (SP) in the 1st column of Virtuoso means that this model successfully returned results faster than all other models in 4 out of the 14 cases. The total column then adds the positions for all engines. From this, we see that looking across all five engines, named graphs is probably the best supported (fastest in 17/70 cases), with standard reification and and n-ary relations not far behind (fastest in 16/70 cases). All engines aside from Virtuoso seem to struggle with singleton properties; presumably these engines make some (arguably naive) assumptions that the number of unique predicates in the indexed data is low.

Although the first three RDF-level formats would typically require more joins to be executed than named graphs, the joins in question are through the statement identifier, which is highly selective; assuming "equivalent" query plans, the increased joins are unlikely to overtly affect performance, particularly when forming part of a more complex query. However, additional joins do complicate query planning, where aspects of different data models may affect established techniques for query optimisation differently. In general, we speculate that in cases where a particular model was much slower for a particular query and engine, that the engine in question selected a (comparatively) worse query plan.

6 Conclusions

In this paper, we have looked at four well-known reification models for RDF: standard reification, n-ary relations, singleton properties and named graphs. We were particularly interested in the goal of modelling Wikidata as RDF, such that

Table 5: Ranking of reification models for query response times

№	4store				BlazeGraph				GraphDB				Jena				Virtuoso				**Total**			
	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG	SR	NR	SP	NG
1st	2	3	0	0	6	2	0	3	2	2	0	7	2	7	0	3	4	2	4	4	16	16	4	17
2rd	2	1	0	0	1	3	2	4	5	2	0	4	6	0	0	5	5	2	1	6	19	8	3	19
3rd	–	–	–	–	3	3	1	2	4	7	0	0	2	2	0	3	3	3	2	3	12	15	3	8
4th	–	–	–	–	0	1	6	2	–	–	–	–	0	0	0	0	2	4	4	1	2	5	10	3
FA	5	5	0	0	4	3	3	3	3	1	0	3	4	3	12	3	0	1	1	0	16	13	16	9
NR	5	5	14	14	0	2	2	0	0	2	14	0	0	2	2	0	0	2	2	0	5	13	34	14

it can be indexed and queried by existing SPARQL technologies. We sketched a conceptual overview of a Wikidata schema based on quads/triples, thus reducing the goal of modelling Wikidata to that of modelling quads in RDF (quads where the fourth element functionally specifies the triple), and introduced the four reification models in this context. We found that singleton-properties offered the most concise representation on a triple level, but that *n*-ary predicates was the only model with built-in support for SPARQL property paths. With respect to experiments over five SPARQL engines – 4store, BlazeGraph, GraphDB, Jena and Virtuoso – we found that the former four engines struggled with the high number of unique predicates generated by singleton properties, and that 4store likewise struggled with a high number of named graphs. Otherwise, in terms of query performance, we found no clear winner between standard reification, *n*-ary predicates and named graphs. We hope that these results may be insightful for Wikidata developers – and other practitioners – who wish to select a practical scheme for querying reified RDF data.

Acknowledgements This work was partially funded by the DFG in projects DIAMOND (Emmy Noether grant KR 4381/1-1) and HAEC (CRC 912), by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004 and by Fondecyt Grant No. 11140900.

References

1. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J. (eds.): Direct Mapping of Relational Data to RDF. W3C Recommendation (27 September 2012), `http://www.w3.org/TR/rdb-direct-mapping/`
2. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: a family of scalable semantic repositories. Sem. Web J. 2(1), 33–42 (2011)
3. Brickley, D., Guha, R. (eds.): RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (10 February 2004), `http://www.w3.org/TR/rdf-schema/`
4. Brickley, D., Guha, R. (eds.): RDF Schema 1.1. W3C Recommendation (25 February 2014), `http://www.w3.org/TR/rdf-schema/`
5. Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J.J., McBride, B. (eds.): RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (25 February 2014), `http://www.w3.org/TR/rdf11-concepts/`

6. Das, S., Srinivasan, J., Perry, M., Chong, E.I., Banerjee, J.: A tale of two graphs: Property Graphs as RDF in Oracle. In: EDBT. pp. 762–773 (2014), `http://dx.doi.org/10.5441/002/edbt.2014.82`
7. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. 35(1), 3–8 (2012)
8. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the linked data web. In: ISWC. pp. 50–65 (2014)
9. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered RDF store. In: Workshop on Scalable Semantic Web Systems. CEUR-WS, vol. 517, pp. 94–109 (2009)
10. Harris, S., Seaborne, A., Prud'hommeaux, E. (eds.): SPARQL 1.1 Query Language. W3C Recommendation (21 March 2013), `http://www.w3.org/TR/sparql11-query/`
11. Hartig, O.: Reconciliation of RDF* and Property Graphs. CoRR abs/1409.3288 (2014), `http://arxiv.org/abs/1409.3288`
12. Hartig, O., Thompson, B.: Foundations of an alternative approach to reification in RDF. CoRR abs/1406.3399 (2014), `http://arxiv.org/abs/1406.3399`
13. Hoffart, J., Suchanek, F.M., Berberich, K., Weikum, G.: YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. Artif. Intell. 194, 28–61 (2013)
14. Hull, R.: Relative information capacity of simple relational database schemata. In: PODS. pp. 97–109 (1984)
15. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. Sem. Web J. 6(2), 167–195 (2015)
16. Manola, F., Miller, E. (eds.): Resource Description Framework (RDF): Primer. W3C Recommendation (10 February 2004), `http://www.w3.org/TR/rdf-primer/`
17. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: Schema equivalence in heterogeneous systems: bridging theory and practice. Inf. Syst. 19(1), 3–31 (1994)
18. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like RDF reification? Making statements about statements using singleton property. In: WWW. pp. 759–770. ACM (2014)
19. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014)
20. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Comm. ACM 57, 78–85 (2014)
21. Wilkinson, K., Sayers, C., Kuno, H.A., Reynolds, D., Ding, L.: Supporting scalable, persistent Semantic Web applications. IEEE Data Eng. Bull. 26(4), 33–39 (2003)
22. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated Semantic Web data. J. Web Sem. 11, 72–95 (2012)

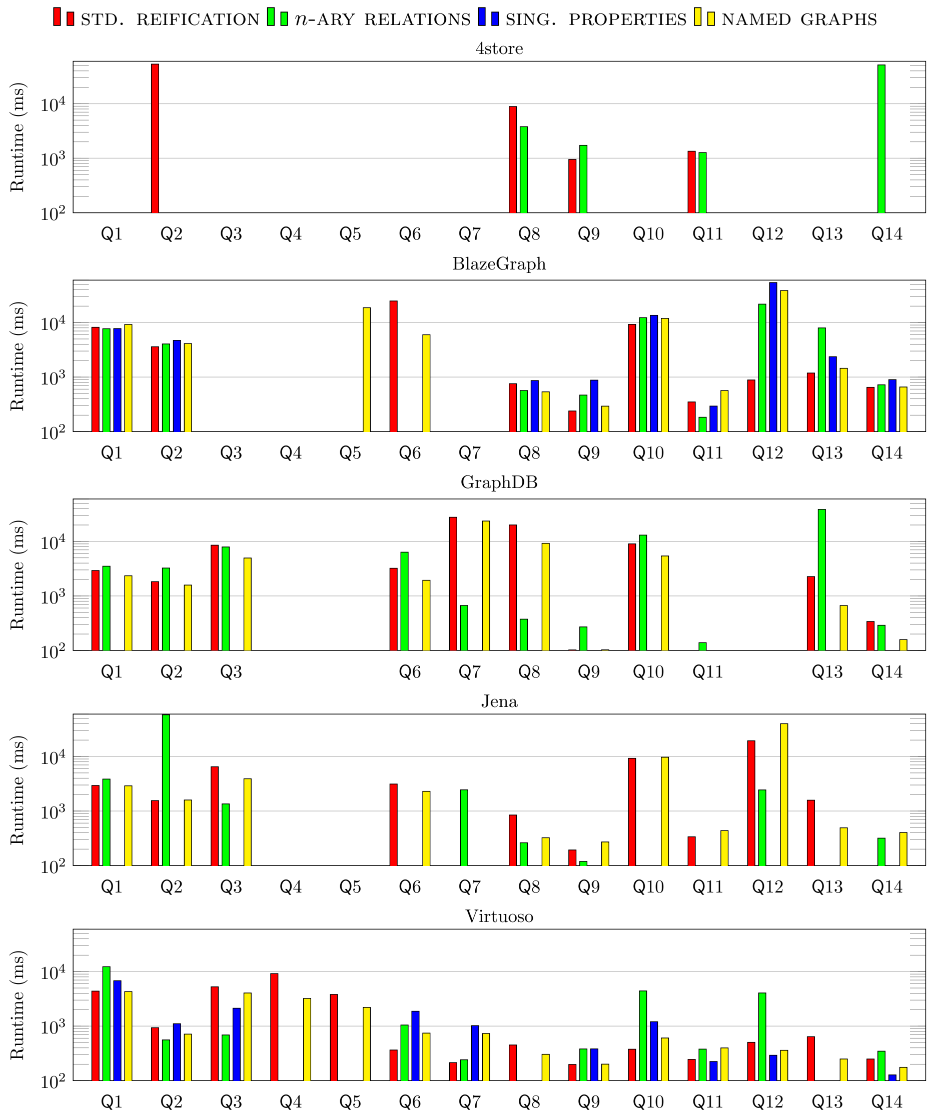


Fig. 3: Query results for all five engines and four models (log scale)

Dynamic join order optimization for SPARQL endpoint federation

Hongyan Wu, Atsuko Yamaguchi, and Jin-Dong Kim

Database Center for Life Science, Research Organization of Information and Systems, Japan
{wu,atsuko,jdkim}@dbcls.rois.ac.jp

Abstract. The existing web of linked data inherently has distributed data sources. A federated SPARQL query system, which queries RDF data via multiple SPARQL endpoints, is expected to process queries on the basis of these distributed data sources. During a federated query, each data source may consist of a search space of nontrivial size. Therefore, finding the optimal join order to minimize the size of intermediate results from different sources is key to optimizing the performance of such federated queries. In this study, we present a dynamic optimization approach to determining join order, which can find more optimized join plans than static optimization approaches. Our experimental results show that our proposed approach stably improves the performance of a federated query as the query becomes increasingly complex.

Keywords: linked data, SPARQL, federated query, dynamic join order optimization

1 Introduction

Linked data technology has substantially contributed to the freeing of data confined in individual silos; however, searching over such data is still performed within a single SPARQL endpoint, making it difficult to truly affirm that data are truly freed from their respective silos even in the linked data space.

A number of federated query systems have been developed to enable search across multiple endpoints. Although it is difficult to assert that the performance of these query systems is close to production level, the research community is continuously trying to improve such performance [2,3,5,6,8,10]. In this paper, we propose a novel technique, i.e., dynamic join order optimization, to significantly improve the performance of federated search.

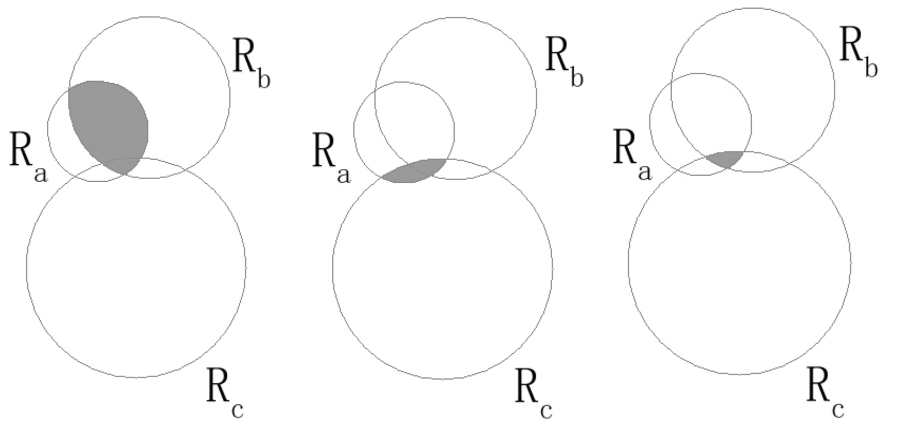
A federated query inherently has to explore multiple endpoints, and while traversing these endpoints, results from one endpoint must be joined with results from the next endpoint and so on. Here each endpoints may consist of a search space of nontrivial size. To efficiently perform the search across these multiple search spaces, determining the optimal join order is key to good performance.

Join order optimization has been a research topic for a number of years [4,11–13]; however, in these studies, the common approach is to somehow try to find the optimal join order before beginning actual exploration into the endpoints. We therefore call this static join optimization. Considering the importance of join order on the performance of a SPARQL query, we argue that join order cannot be sufficiently optimized at the onset of the query; further, by utilizing intermediate results obtained during search, join order can be significantly improved. We present a simple algorithm for dynamic join order optimization as well as an implementation in the form of an extension to FedX.

Our experimental results show that dynamic join order optimization is effective in controlling the search space size, thereby avoiding explosions in size. We also developed a new benchmark for evaluating the join optimization of federated query performance. This benchmark is developed to include more complex join operations than those introduced in FedBench [8]. Our experimental results here show that our dynamic join order approach stably improves the performance of federated search.

2 Related work

In relational databases, associated data entries are maintained in tables consisting of any number of columns; in RDF, data pieces are maintained in triples, the smallest unit of representation for typed binary relationships. Therefore, join operations generally occur much more frequently when processing a SPARQL query than when processing a corresponding SQL query.



(a) Static approach (b) Dynamic approach (c) Final results for two approaches

Fig. 1. Intermediate results for the static and dynamic approaches

Suppose we have a query that can be decomposed into three subqueries, Qa, Qb, and Qc, which have answers Ra, Rb, and Rc, respectively, from three different endpoints. Then, final answers are to be those that satisfy the constraints set by the three subqueries. In Figure 1(c), the three circles Ra, Rb, and Rc represent the sets of results of the three subqueries, with the gray area representing the final results. To reach the set of final results, there are six distinct join orders, i.e., (1) A $\rightarrow$ B $\rightarrow$ C; (2) A $\rightarrow$ C $\rightarrow$ B; (3) B $\rightarrow$ A $\rightarrow$ C; (4) B $\rightarrow$ C $\rightarrow$ A; (5) C $\rightarrow$ A $\rightarrow$ B; and (6) C $\rightarrow$ B $\rightarrow$ A. Regardless of which join order is selected, the final set of results is the same; however, the number of intermediate results that must be handles varies on the basis of the different join orders. For example, if subquery Qc is executed first, Rc must be handled as the initial set of intermediate results; however, we would like to avoid that choice because $|Rc|$ produces the largest set of intermediate results among the three possible subqueries.

If the size of the intermediate results is known or can be estimated in advance, the join order may be optimized. For example, the result size of the individual subqueries may be estimated in advance as $|R_a| < |R_b| < |R_c|$. Based on this information, the join order may be optimized as A $\rightarrow$ B $\rightarrow$ C. Below are the necessary operations that must occur in the given order:

1. Receive result set R_a.
2. Bind variables in query Q_b using result set R_a and then submit intermediate results to E_b.
3. Receive result set $R_a \cap R_b$.
4. Bind variables in query Q_c using result set $R_a \cap R_b$ and then submit intermediate results to E_c.
5. Receive final result $R_a \cap R_b \cap R_c$.

With the given join order, the size of the intermediate result sets that must be handled is $|R_a| + |R_a \cap R_b|$. This is more or less the scenario in which most federated search systems have been developed in terms of join order optimization, i.e., to better optimize the join order, attempt to estimate the result set sizes of individual subqueries with heuristics or statistical information.

In this paper, we argue that even if the initial estimation is performed perfectly, there is still large room for further optimization. Note that after Qa is first executed, there are two choices for the next execution, i.e., Qb and Qc. Although $|Rb|$ is estimated to be smaller than $|Rc|$, choosing Qc for the next execution is in fact a more optimal choice because $|Ra \cap Rc|$ (i.e., Figure 1(b)) is smaller than $|Ra \cap Rb|$ (i.e., Figure 1(a)). To select the optimal choice in this case, we propose a dynamic join order optimization approach that evaluates queries as follows: (1) evaluate the size of all subqueries, obtaining $|R_a| < |R_b| < |R_c|$; (2) evaluate Qa, then apply Ra to Qb and Qc, noting that $|R_a \cap R_c|$ is less than $|R_a \cap R_b|$; (3) evaluate $|R_a \cap R_c|$; and (4) join Qc. Therefore, the join order is A $\rightarrow$ C $\rightarrow$ B. Here the dynamic approach obviously performs better than the static approach because the intermediate result space $|R_a| + |R_a \cap R_c|$ is smaller than the static approach space (i.e., $|R_a| + |R_a \cap R_b|$).

To date, research regarding join order optimization, both in relational database and RDF data management systems, has been centered on static optimization in which optimization is performed only once before queries are actually executed. As an example, FedX builds a subquery for a group of triple patterns in which each triple exclusively shares a single relevant source. FedX assumes this type of exclusive subquery, and the subquery with fewer free variables has a high selectivity ranking. The assumed selectivity ranking and variable counting technologies are not suitable for all situations as queries become complex. DARQ [6], SPLENDID [3], ADERIS [5], Avalanche [2], and other similar systems use pre-computed information, such as service description or VoID, to estimate selectivity and optimize join order; however, none of these can overcome the fragility of static optimization techniques. More specifically, the search space changes as the query is processed. Based on this and the frequency of join operations in a SPARQL query, we argue that join order should be optimized by utilizing intermediate results with a dynamic approach.

3 Dynamic join order optimization model

3.1 Static join order optimization

To best introduce our dynamic join order optimization algorithm, we first show a simple algorithm that uses the static join order strategy. Here we assume the existence of a *sortSubQueries* operation to sort subqueries by some measure and an *evaluateQuery* operation to output a set *preResults* of results for variables appearing in a given SPARQL query.

Algorithm 1 Query execution with static join order optimization

```
1: function StaticJoin($setSubQueries$: a set of subqueries)
2:     $listSubQueries \leftarrow sortSubQueries(setSubQueries)$
3:     $preResult \leftarrow \emptyset$
4:     while $listSubQueries$ is not empty do
5:         $curSubQuery \leftarrow pop(listSubQueries)$
6:         $preResult \leftarrow evaluateQuery(preResult, curSubQuery)$
7:     end while
8:     return $preResult$
9: end function
```

Algorithm 1 shows the flow of query execution when a static join order optimization scheme is applied. Given the *setSubQueries* set of subqueries, the algorithm first sorts the subqueries on the basis of estimations of their result sizes and then executes the subqueries in the given order. In other words, the optimal join order is determined before the execution of any subqueries, and the join order does not change during execution, which is why we call it a "static" optimization strategy.

3.2 Dynamic join order optimization

Algorithm 2 shows the flow of query execution with dynamic join order optimization. Unlike the static optimization strategy described above, the optimal subquery to be executed next is determined at each step of query execution by considering the intermediate results obtained thus far. We therefore call this approach a "dynamic" optimization strategy.

Algorithm 2 Query execution with static join order optimization

```
1: function DynamicJoin($setSubQueries$: a set of subqueries)
2:     $preResult \leftarrow \emptyset$
3:     while $setSubQueries$ is not empty do
4:         $curSubQuery \leftarrow$ findOptimalSubQuery($setSubQueries, preResult$)
5:         $preResult \leftarrow evaluateQuery(preResult, curSubQuery)$
6:         $setSubQueries \leftarrow setSubQueries - \{curSubQuery\}$
7:     end while
8:     return $preResult$
9: end function
```

In the algorithm, **findOptimalSubQuery** finds the subquery with the highest selectivity among all subqueries (line 4). On line 6, the executed subquery is removed from the subquery set, and then this process repeats until all subqueries finish.

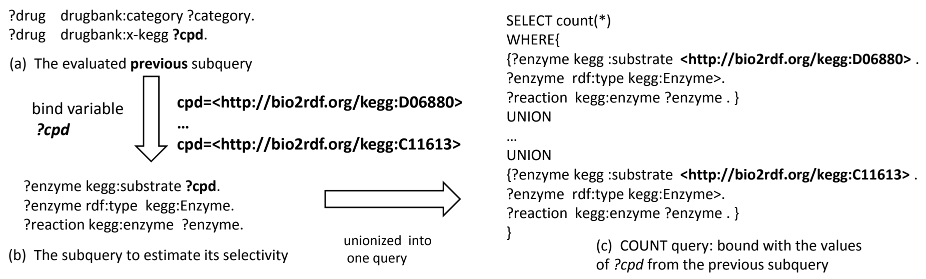
Finding the optimal subquery In Algorithm 3, we apply a greedy strategy at each step to find the subquery that has the smallest result size.

Algorithm 3 Finding the optimal subquery

```
1: function findOptimalSubQuery($setSubQueries$, $preResult$)
2:     $optimalSubQuery \leftarrow setSubQueries[0]$
3:     $minSize \leftarrow MAX\_VALUE$
4:     for each $subQuery$ in $setSubQueries$ do
5:         if $|setSubQueries|$ equals 1 then
6:             break
7:         end if
8:         $size \leftarrow$ estimateResultSize($subQuery, preResult$)
9:         if $size < minSize$ then
10:            $optimalSubQuery \leftarrow subQuery$
11:            $minSize \leftarrow size$
12:        end if
13:    end for
14:    return $optimalSubQuery$
15: end function
```

Estimating result size There are many approaches for estimating the result size of a subquery, for example, using pre-computed statistical information. In this paper, our implementation uses COUNT queries that do not need any precomputed information. More specifically, we bind previous subquery results to each remaining subquery, construct a COUNT query, and send it on the fly to the relevant sources to determine under the current conditions how many intermediate results they will produce.

Note that a COUNT query is a SPARQL query with the form "select count(*)..." that evaluates the result size of a subquery. We construct COUNT queries for all subqueries as follows: (1) search the triple pattern with a bound value from previous results *preResult*; (2) bind the variables in the remaining subqueries, i.e., *setSubQueries*, with their corresponding values; and (3) use UNION keywords to combine multiple small queries for a subquery into a large query to decrease the number of COUNT queries. An example of our approach here is shown in Figure 2; note that this example comes from our benchmark Q7 and that the bold font portion represents the bound variable and its value.



```
?drug    drugbank:category ?category.
?drug    drugbank:x-kegg ?cpd.
```

(a) The evaluated **previous** subquery

```
cpd=<http://bio2rdf.org/kegg:D06880>
...
cpd=<http://bio2rdf.org/kegg:C11613>
```

```
?enzyme kegg:substrate ?cpd.
?enzyme rdf:type  kegg:Enzyme.
?reaction kegg:enzyme  ?enzyme.
```

(b) The subquery to estimate its selectivity

```
SELECT count(*)
WHERE{
{?enzyme kegg :substrate  <http://bio2rdf.org/kegg:D06880> .
?enzyme  rdf:type kegg:Enzyme>.
?reaction  kegg:enzyme ?enzyme . }
UNION
...
UNION
{?enzyme kegg :substrate  <http://bio2rdf.org/kegg:C11613> .
?enzyme  rdf:type kegg:Enzyme>.
?reaction  kegg:enzyme ?enzyme . }
}
```

(c) COUNT query: bound with the values of *?cpd* from the previous subquery

Fig. 2. An example using a COUNT query to evaluate selectivity for a subquery

For dynamic join order optimization, the system must apply all previous query results to the candidate subqueries; however, when there are a large number of values in the intermediate results, it is costly to bind all values to the remaining subqueries and execute the large query. Note that for join order optimization, we need only a rough estimate of the size of the query results on which the subqueries may be ordered. This estimation does not need to be very precise because a small difference in the size of results will not significantly impact the overall performance.

Thus, rather than exhaustively consider the entire set of intermediate results, we take a small sample of size n and order the subqueries by the size of the results after binding relevant variables with the sample values. In this work, we simply set the size of n to be 3. While it may be necessary to estimate the optimal sample size, at this point, we assume that it is not a critical factor for the reason noted above.

Instead of estimating the cost of expressions with VoID as SPLENDID, the **estimateResultsSize** function actually sends the COUNT query to its relevant data sources. Here, we note two important observations: (1) the performance cost of a COUNT query at its local endpoint is not very large and (2) a COUNT query returns only one number, which is far less information than that if a full result set was returned.

4 Evaluation

4.1 Evaluation of join optimization

We investigated how dynamic join optimization influences the query performance in comparison with the static join. As we noted above, FedX is the fastest engine among the current federated SPARQL endpoint query systems according to recent benchmarks. We therefore implemented all the functions, including source selection, on the basis of the FedX system, and compared the differences before and after using dynamic join optimization in conjunction with the FedX system. Further, we evaluated SPLENDID, which is expected to produce a good join order plan using statistical information and optimizing plans on the basis of dynamic programming techniques.

FedBench is a comprehensive benchmark suite for federated semantic data that considers the evaluation of UNION, FILTER, and OPTIONAL clauses; however, we note that almost all queries in this benchmark have a common characteristic, i.e., they include a single triple pattern with two bound variables and only one free variable, as shown in the query below from Cross Domain evaluation CD6.

```
SELECT ?name ?location ?news
WHERE {
?artist <http://xmlns.com/foaf/0.1/name> ?name .                          (1)
?artist <http://xmlns.com/foaf/0.1/based_near> ?location .                (2)
?location <http://www.geonames.org/ontology#parentFeature> ?germany .     (3)
?germany <http://www.geonames.org/ontology#name> 'Federal Republic of Germany' (4)
}
```

Triple pattern (4) with two bound variables usually has a higher selectivity. A good join optimization plan should execute this type of triple pattern at an earlier stage in a sequence of joins; however, this type of triple pattern can be simply identified even with very simple optimization technologies, such as the variable counting technique used in the FedX system to count the number of bound variables. To better evaluate the influence of dynamic and static joins, we designed a benchmark to evaluate join optimization for federated SPARQL endpoint queries.

Benchmark setup For our benchmarks, we used five real biological SPARQL endpoints from the Bio2RDF project [1], which is a different setup than FedBench [8], SP^{2}Bench [9], and the fine-grained evaluation of SPARQL endpoint federation systems [7], all of which use a simulated federated environment and synthetic data or a subset of real data. For the life science field, FedBench uses three biological datasets, namely KEGG, ChEBI, and Drugbank. Because the

SPARQL endpoint for CHEBI in the Bio2RDF project [1] is still under construction, we selected KEGG, Drugbank, SIDER, OMIM, and PharmGKB.

These datasets connect to one another closely by relationships between gene, drug, disease, reaction, side effect, and others. Table 1 presents the details of each dataset. The data are far more complicated than the FedBench life science data. The largest biological dataset in FedBench is a subset of ChEBI that includes 7.33 million triples, 28 predicates, and a single type. In the Bio2RDF project, the server of each endpoint is set to return a maximum of 10,000 results at a time, regardless of the real result size. This restriction is commonplace to lessen the burden on the server. Note that all settings in the Bio2RDF servers are beyond our control.

Table 1. Bio2RDF dataset

Dataset	Endpoint	#Triples(M)	#Pred	#Types
Drugbank	http://cu.kegg.bio2rdf.org/sparql	3.48	105	91
OMIM	http://cu.drugbank.bio2rdf.org/sparql	8.35	101	34
SIDER	http://cu.pharmgbk.bio2rdf.org/sparql	16.81	39	16
KEGG	http://cu.sider.bio2rdf.org/sparql	47.87	141	63
PharmGKB	http://cu.omim.bio2rdf.org/sparql	265.17	88	50

This benchmark focuses on testing the join operation in the SPARQL endpoint federation. We consider the following points in designing the queries: (1) the number of triple patterns (#Tp) varies from two to nine; (2) the number of queried endpoints (#Src) has a size ranging from two to five; and (3) the number of returned results (#Res) ranges from 1 k to 109 k. Queries returning large result set sizes are very useful when integrating data from multiple data sources. Table 2 shows the query characteristics in detail.

Table 2. Bio2RDF query characteristics

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
#Tp	2	4	4	4	8	9	6	8
#Src	2	2	2	2	5	5	5	5
#Res	1	5	5	5	9492	132	32003	111962

In addition, the query set considers the numbers of variables in a triple pattern. RDF data could connect to each other via different paths, which brings about more free variables. The fewer bound variables, the more difficult it is to estimate selectivity. Here Q2 and Q3 include one triple pattern in which all variables are free, Q3 changes the position of the triple pattern, and Q4 increases another such triple pattern. The rest of the queries consider the influence of the triple pattern with two bound variables. In this case, Q5, Q7, and Q8 have only one triple pattern with two bound variables, whereas Q6 has two such triple patterns. Next, Q7 is a variation of LS4 from FedBench, with Q7 obtained by slightly modifying the bound variables, thereby increasing the result size. Further, Q5 is a variation of Q7 obtained by changing the connected dataset and constructing a more complicated star Q8 subquery. Here Q6 tests a query with complicated star subqueries, evaluating the query connecting three datasets. Finally, Q1 is designed to evaluate the extreme case with only two join triple patterns.

We sequentially executed each query five times, removing the largest and smallest values, calculating the mean value of the three remaining values.

Query performance Figure 3 summarizes query performance, and Table 3 shows how intermediate results changed. Dynamic join optimization outperformed the original static FedX system during all queries, except for Q5. As for the time cost, for Q1, Q2, Q6, and Q8, the dynamic approach was faster than the original FedX system. FedX failed on Q4, which has two triple patterns in which all variables are free [1]. With regard to the result completeness, the dynamic approach returned all results for all queries, whereas Fedx returned incomplete results for Q2, Q3, Q6, and Q7. Finally, SPLENDID returned all results for Q1, Q2, and Q4, in which Q2 and Q4 were slower than the dynamic join and Q1 was slightly faster; note that SPLENDID failed all other queries by reaching the one-hour timeout limitation.

Intermediate results shown in Table 3 detail the query performance of both FedX and our dynamic join approach. Intermediate results for the first step, namely the results of the first subquery, show the selectivity of the first subquery. In the table, the number outside the bracket shows the real intermediate result size that the subquery should return, whereas the number inside the bracket shows the actual intermediate size returned within the 10,000-result limitation of the server.

The real intermediate result sizes of Q1, Q2, Q3, Q4, Q6, Q7, and Q8 of FedX were far larger than those of the dynamic join optimization; therefore, FedX was much slower for queries Q1, Q2, Q6, and Q8. For Q7, because of the restrictions on the returned size, the returned intermediate results size was 10,000 (though it should be 80,460), which was less than that of the dynamic join; therefore, the query seemed faster; however, Fedx returned incomplete results, while our dynamic approach returned all results. For Q3, Fedx failed in the second step because this step returned zero results, while our dynamic join approach successfully finished the query in the third step. For Q5, FedX and our dynamic approach produced the same number of intermediate results and the same join plan. In this case, the dynamic approach needed an additional join order optimization cost and was therefore a little slower. We did not measure the details of the intermediate results for SPLENDID.

[1] A SPARQL compiler error occurs when FedX joins a certain intermediate result with another subquery. The dynamic join avoids this problem because the number of intermediate results is much less than that of FedX.

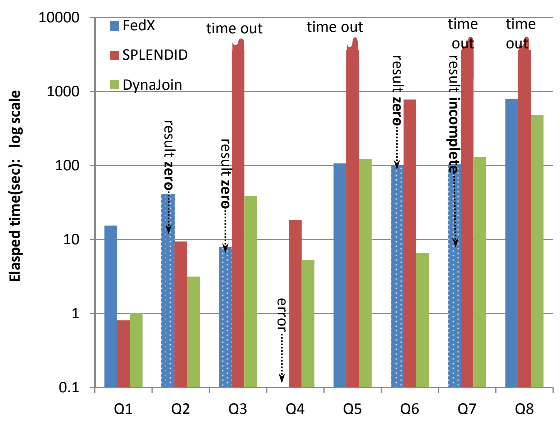


Fig. 3. Bio2RDF results

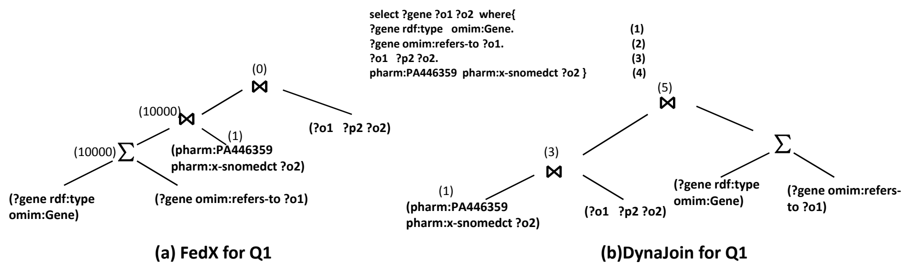
Table 3. Intermediate results of the first two steps

	1st step		2nd step	
	FedX	Dyna	Fedx	Dyna
Q1	14609 (10000)	2	1	1
Q2	95443 (10000)	1	95443 (10000)	3
Q3	91656866 (10000)	1	0	10000
Q4	14609 (10000)	1	14609 (10000)	3
Q5	19	19	9492	9492
Q6	70115 (10000)	2	0	132
Q7	80406 (10000)	4323	>10000[a]	32077
Q8	36	36	60362	27

[a] The exact size could not be measured because its previous step was not completely executed.

We investigated why Fedx returned no results for Q2. Figure 4(a) and 4(b) illustrate the produced join plan of Q2. In the figures, the number inside the bracket shows the intermediate result after executing the operation. The first evaluation produced by FedX was the *exclusive group*. With the limitation of the OMIM server, the evaluation returned 10,000 intermediate results that contributed no final results; here the actually produced intermediate result size was 95,443. The dynamic join approach sends COUNT queries; thus, determining the fourth triple pattern has the highest selectivity, thereby returning only one result. It then binds the results of variable *?o2* to the other triple patterns, constructs the COUNT queries, and sends them to the relevant endpoints. In this case, the dynamic approach judged the third triple pattern to have fewer results and finally joined the exclusive group. The reason why Q3 and Q6 returned no results is similar to that of Q2.

For Q7 and Q8, it was more difficult to make a join order plan. There is a single triple pattern with two bound variables, which seemingly has higher selectivity. For Q7, FedX first produced a larger initial search space of 80,406, which partially contributed to the final results and therefore returned only part of the results. The dynamic join first evaluated the group (i.e., 4323 results from the fourth and sixth triple patterns), with the search space size being far less than that of Fedx. Consequently, FedX returned only part of the results, whereas the dynamic join returned all results.

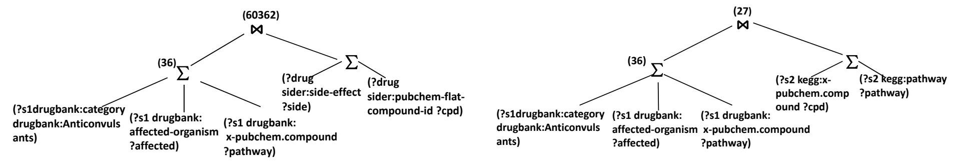


```
select ?gene ?o1 ?o2  where{
?gene rdf:type   omim:Gene.                              (1)
?gene omim:refers-to ?o1.                                (2)
?o1   ?p2 ?o2.                                           (3)
pharm:PA446359  pharm:x-snomedct ?o2 }                   (4)
```

(0)
(10000)
(10000)
(1)
(pharm:PA446359 pharm:x-snomedct ?o2)
(?o1 ?p2 ?o2)
(?gene rdf:type omim:Gene)
(?gene omim:refers-to ?o1)
(a) FedX for Q1
(5)
(3)
(1)
(pharm:PA446359 pharm:x-snomedct ?o2)
(?o1 ?p2 ?o2)
(?gene rdf:type omim:Gene)
(?gene omim:refers-to ?o1)
(b)DynaJoin for Q1

Fig. 4. Join order and intermediate result sizes (inside the brackets) for Q2.

For Q8, 111,962 results were returned-the largest size in this group of queries. The query was evaluated across three endpoints, as shown in Figure 5. Both FedX and our dynamic join first evaluated the exclusive group (i.e., the first three triple patterns) at the Drugbank endpoint. Next, FedX evaluated the second exclusive group (i.e., the fourth and fifth triple patterns); however, the dynamic join approach judged the second exclusive group to have more results than the third exclusive group (i.e., the seventh and eighth triple patterns). Evaluating the third exclusive group earlier substantially reduced the size of the intermediate results, thereby accelerating the query.


(60362)
(36)
(?s1drugbank:category drugbank:Anticonvuls ants)
(?s1 drugbank: affected-organism ?affected)
(?s1 drugbank: x-pubchem.compound ?pathway)
(?drug sider:side-effect ?side)
(?drug sider:pubchem-flat-compound-id ?cpd)
(27)
(36)
(?s1drugbank:category drugbank:Anticonvuls ants)
(?s1 drugbank: affected-organism ?affected)
(?s1 drugbank: x-pubchem.compound ?pathway)
(?s2 kegg:x-pubchem.comp ound ?cpd)
(?s2 kegg:pathway ?pathway)

(a) FedX and SPLENDID for Q8. (b) DynaJoin for Q8.

Fig. 5. Join order and intermediate result sizes (inside the brackets) for Q8.

For Q5, FedX and our dynamic approach produced the same join plan. The dynamic approach needed an additional join order optimization cost; therefore, FedX was slightly faster. Table 4 shows the additional overhead and their corresponding percentages accounting for the total query time in detail. The largest overhead here was 3.74 seconds for Q5. Consequently, the size increased and the query became heavier, thereby causing the optimization cost to no longer seem insignificant.

In addition, our evaluation shows that SPLENDID cannot produce a better join plan than our dynamic approach despite using pre-computed statistical information. More specifically, we checked the join plan produced by SPLENDID. For Q7 and Q8, SPLENDID produced the same join order as FedX, which generated far larger intermediate results than our dynamic approach. For other queries, SPLENDID produced the same join order plan as our dynamic join

Table 4. Additional overhead of our dynamic join approach

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
time(sec)	0.4	2.33	2.96	3.49	3.74	1.29	1.09	2.24
%	40.5	73.9	7.7	65.8	3.0	19.5	0.8	0.4

approach. The additional cost of the dynamic approach for Q1 resulted from the two COUNT queries, while SPLENDID used pre-computed information to evaluate the selectivity of the two triple patterns.

We also checked the difference when using an index cache; however, we do not provide details here because the cache was not used in the dynamic join order procedure. Here source selection was implemented in the same way as that in case of FedX, which does not impact performance; therefore, the aforementioned conclusions still hold.

4.2 Fedbench benchmark

As mentioned in the above section, the FedBench benchmark cannot measure the performance of join optimization in the federated query well because of its simplicity in producing a join plan; however, in this section, we still provide evaluation results with the Fedbench benchmark as a reference.

Our experiments were conducted on the AWS platform, with five m3.2xlarge instances for the Cross Domain dataset and four instances for life science data. These instances were configured with Intel(R) Xeon(R) CPU E5-2670 v2 2.50 GHz 4 Core CPU with 30 GB RAM and high network performance property (AWS standards) with a 64-bit GNU/Linux operating system and the 64-bit Java VM 1.7.0_75. All datasets were stored with an 8 GiB general purpose SSD EBS, except for the Geonames dataset, which used a 100 GiB one. Endpoints used open-source Virtuoso 07.00.3203.

Table 5 summarizes the FedBench dataset, while Table 6 presents query characteristics. #Tp., #Src, and #Res represent the number of triple patterns, data sources, and results, respectively. Figure 6 presents our experimental results.

Except for query LS6, FedX was slightly faster than our approach, with a maximum difference of less than 0.5 seconds. Our proposed dynamic join eventually generated the same join plan as FedX. Therefore, the cost difference mainly came from the additional optimization cost of our proposed dynamic optimization algorithm. Overall, the additional cost is not substantial. Further, as the queries in the life science field become heavier than queries in the cross domain, the additional cost will decrease.

CD1 shows an extreme case in which only two triple patterns were joined. In this query, Fedx simply identified the triple pattern with higher selectivity. Our dynamic join approach seemed to experience a large cost (0.5 seconds) for optimization; however, the evaluation of Q1 in our designed benchmark, which also joined two triple patterns, showed our dynamic join approach to be much faster than FedX. In such cases, they applied different join order plans. LS6 illustrated a special case in which our dynamic join outperformed FedX. The results of this query are different from what was described in the FedX paper; the FedX team has confirmed these results with our current dataset and settings. We are jointly investigating the reasons why these inconsistencies exist.

Table 5. FedBench datasets

Dataset	#Triples(M)	#Pred	#Types
DBpedia subset	43.6M	1063	248
GeoNames	108M	26	1
LinkedMDB	6.15M	222	53
Jamendo	1.05M	26	11
New York Times	335k	36	2
KEGG	1.09M	21	4
ChEBI	7.33M	28	1
Drugbank	767k	119	8

Table 6. Query characteristics

Cross Domain(CD)				Life Science (LS)		
Query	#Tp.	#Src	#Res	#Tp	#Src	#Res
1	3	2	90	2	2	1159
2	3	2	1	3	4	333
3	5	5	2	5	3	9054
4	5	5	1	7	2	3
5	4	5	2	6	3	393
6	4	4	11	5	3	28
7	4	5	1	5	3	144

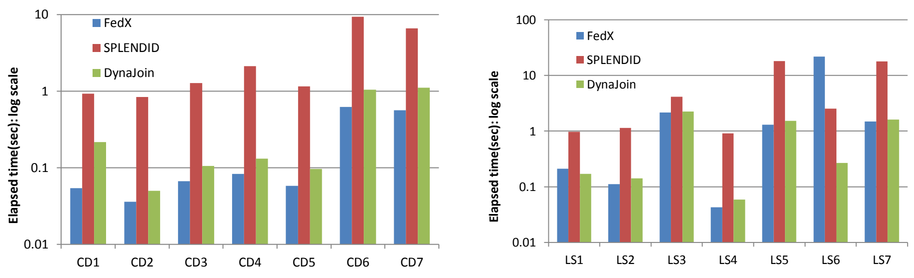


Fig. 6. FedBench results

5 Conclusions

In this paper, we proposed a novel dynamic join order optimization technique. Because the search space of SPARQL queries is always changing, we believe that the join order should be dynamically optimized during query execution, considering the frequency and importance of the join operation in such SPARQL queries. We perform a SPARQL query by executing a group of subqueries in which we optimize the join order by binding the variable values from previous subqueries to the remaining subqueries and then evaluating the next intermediate result size and selecting the plan with the minimum intermediate result size. Both the Fedbench benchmark and our heavier federated biological benchmark proved that in comparison with the static optimization approach, our proposed dynamic approach engine can stably present an optimal join plan and therefore improve the performance of a federated query, with the degree of improvement becoming clearer as the query becomes more complex. Our dynamic approach does introduce additional overhead with its multiple updates of the join plan, with the overhead being significant in queries that return a small number of results and therefore have join orders that are not complex; however, as queries become more complex and result sizes increase, the optimization cost becomes increasingly insignificant.

Note that the overhead of the COUNT queries could be further controlled by parallelizing the COUNT queries and setting timeout limitations. For the first returned COUNT query, we could assume that it has less of a join cost because the amount of data, the scale of server computational ability, or the degree of network cost is better than others. We plan to implement this in the future to gain a better understanding here. In addition, although we implemented selectivity estimation via COUNT queries in this paper, other approaches are available. With fine-grained metadata, selectivity estimation could be estimated with less cost, although previous results provide concrete instances.

Acknowledgements

This work was supported by the National Bioscience Database Center (NBDC) of the Japan Science and Technology Agency (JST). We also thank the continued support from the FedX team for evaluating FedBench.

References

1. Bio2rdf, `http://bio2rdf.org/`
2. Basca, C., Bernstein, A.: Avalanche: Putting the spirit of the web back into semantic web querying. In: 9th International Semantic Web Conference (ISWC2010) (November 2010), `http://data.semanticweb.org/conference/iswc/2010/paper/527`
3. Grlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: In Proceedings of the 2nd International Workshop on Consuming Linked Data (2011)
4. Haas, P.J., Naughton, J.F., Seshadri, S., Swami, A.N.: Selectivity and cost estimation for joins based on random sampling. Journal of Computer and System Sciences 52(3), 550 – 569 (1996), `http://www.sciencedirect.com/science/article/pii/S0022000096900410`
5. Lynden, S.J., Kojima, I., Matono, A., Tanimura, Y.: Aderis: Adaptively integrating rdf data from sparql endpoints. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA (2). Lecture Notes in Computer Science, vol. 5982, pp. 400–403. Springer (2010), `http://dblp.uni-trier.de/db/conf/dasfaa/dasfaa2010-2.html#LyndenKMT10`
6. Quilitz, B., Leser, U.: Querying distributed rdf data sources with sparql. In: Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications. pp. 524–538. ESWC'08, Springer-Verlag, Berlin, Heidelberg (2008), `http://dl.acm.org/citation.cfm?id=1789394.1789443`

7. Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., Ngonga Ngomo, A.C.: A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web Journal (2014), http://svn.aksw.org/papers/2014/fedeval-swj/public.pdf
8. Schmidt, M., Grlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: Fedbench: A benchmark suite for federated semantic data query processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) International Semantic Web Conference (1). Lecture Notes in Computer Science, vol. 7031, pp. 585–600. Springer (2011), http://dblp.uni-trier.de/db/conf/semweb/iswc2011-1.html#SchmidtGHLST11
9. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: Sp2bench: A sparql performance benchmark. CoRR abs/0806.4627 (2008)
10. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: A federation layer for distributed query processing on linked open data. In: The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II. pp. 481–486 (2011), http://dx.doi.org/10.1007/978-3-642-21064-8_39
11. Steinbrunn, M., Moerkotte, G., Kemper, A.: Optimizing join orders. Citeseer (1993)
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web. pp. 595–604. WWW '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1367497.1367578
13. Swami, A., Schiefer, K.: On the estimation of join result sizes. In: Jarke, M., Bubenko, J., Jeffery, K. (eds.) Advances in Database Technology EDBT '94, Lecture Notes in Computer Science, vol. 779, pp. 287–300. Springer Berlin Heidelberg (1994), http://dx.doi.org/10.1007/3-540-57818-8_58

Parallel Data Loading during Querying Deep Web and Linked Open Data with SPARQL

Pauline Folz[1,2], Gabriela Montoya[1,3], Hala Skaf-Molli[1], Pascal Molli[1], and Maria-Esther Vidal[4]

[1] LINA– Nantes University, France
{pauline.folz,gabriela.montoya,hala.skaf,pascal.molli}@univ-nantes.fr
[2] Nantes Métropole - Direction Recherche, Innovation et Enseignement Supérieur, France
[3] Unit UMR6241 of the Centre National de la Recherche Scientifique (CNRS), France
[4] Universidad Simón Bolívar, Venezuela
{mvidal}@ldc.usb.ve

Abstract. Web integration systems are able to provide transparent and uniform access to heterogeneous Web data sources by integrating views of Linked Data, Web Service results, or data extracted from the Deep Web. However, given the potential large number of views, query engines of Web integration systems have to implement execution techniques able to scale up to real-world scenarios and efficiently execute queries. We tackle the problem of SPARQL query processing against RDF views, and propose a non-blocking query execution strategy that incrementally accesses and merges the views relevant to a SPARQL query in a parallel fashion. The proposed strategy is implemented on top of Jena 2.7.4, and empirically compared with SemLAV, a sequential SPARQL query engine on RDF views. Results suggest that our approach outperforms SemLAV in terms of the number of answers produced per unit of time.

1 Introduction

Linked Open Data initiatives have motivated the integration of a large number of RDF datasets into the Linking Open Data (LOD) cloud [4]. Different Web-based interfaces are available to access these publicly accessible Linked Data sets, *e.g.*, SPARQL endpoints and Linked Data fragments [17]. However, the Deep Web which has around 500 times the size of the Surface Web [11, 10] has not been integrated as part of LOD cloud. Performing SPARQL queries without considering the Deep Web can potentially deliver incomplete results. For example, the execution of the SPARQL query: *Which members of the Semantic Web community are interested in* Dalai Lama*,* Barack Obama*, or* Rihanna*?* (cf. Figure 2) without the integration of the Deep Web will provide no answers [8]. Nevertheless, if data from social networks such as Twitter, Facebook, or LinkedIn were considered, the query execution could return some answers.

Two main approaches exist for data integration: data warehousing, and the virtual mediators [7]. Semantic data-warehouses such as Virtuoso with the Sponger feature [1] allow for the implementation of wrappers able to create RDF data from unsemantified data sources, *e.g.*, Web services, CSV files; but this approach may suffer from the *freshness problem* [2], *i.e.*, data may become stale when data sources are updated.

On the other hand, a mediator relies on a global schema to provide a uniform interface for accessing the data sources. Global-As-View (GAV) and Local-As-View (LAV), are the main paradigms for mapping data sources and the global schema. In GAV mediators, entities of the global schema are described using views over the data sources, but including or updating data sources may require the modification of a large number of views [16]. Whereas, in LAV mediators, the sources are described as views over the global schema, and adding new data sources can be easily done [16]. Despite of its expressiveness and flexibility, LAV query re-writting is in general intractable, *i.e.*, NP-complete for conjunctive queries [3]. State-of-the-art LAV query rewriters efficiently solve some families of the query rewriting problem [3, 12]; nevertheless, they may not equally perform on SPARQL queries [13]. Recently, SemLAV [13], the first scalable LAV-based approach for SPARQL query processing, was proposed. Instead of enumerating the query rewritings of a SPARQL query, SemLAV selects the most relevant LAV views, accesses the selected views according to their relevance, and materializes the downloaded data into an *integrated RDF graph*. Then, the SPARQL query is executed against the integrated RDF graph.

SemLAV provides a new paradigm to execute SPARQL queries against LAV views, but because relevant views are loaded sequentially, SemLAV may get blocked loading large views. In the worst case, if the first loaded view is huge and it does not provide relevant data for the query answer, SemLAV will be blocked without producing any answer. Following a sequential view loading strategy may reduce the number of answer produced per unit of time, *i.e.*, throughput, and the time for first answer. Loading several views in parallel may overcome these limitations. However, a parallel view loading strategy will introduce the problem of concurrent writing on the integrated RDF graph. In this paper, we propose a non-blocking query execution strategy to integrate the data from the relevant views into the integrated RDF graph in a parallel fashion. We implement the proposed non-blocking strategy on the top of Jena 2.7.4; we name this new SPARQL query engine *parallel SemLAV*. Further, an empirical evaluation is conducted to study the new parallel strategy with respect to SemLAV. The Berlin Benchmark [5] and queries and views designed by Castillo-Espinola [6] are used to evaluate both query engines. Results suggest that the parallel SemLAV outperforms SemLAV with respect to answers produced per time unit.

The paper is organized as follows. Section 2 describes background and motivation. Section 3 presents strategies for integrating relevant views into the integrated RDF graph in a parallel fashion. Section 4 reports our experimental results. Finally, conclusions and future work are outlined in Section 5.

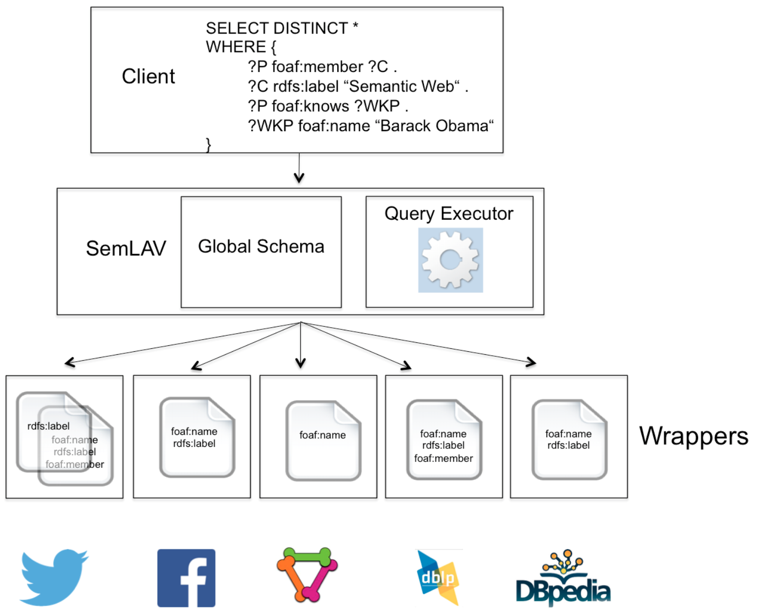


Fig. 1: SemLAV a mediator and wrapper architecture

2 Background and Motivation

SemLAV follows a mediator and wrapper architecture [18] where data from the sources are virtually integrated by SemLAV in a global schema composed by several RDF vocabularies, as shown in Figure 1. Sources are described by LAV views and can be heterogeneous, *e.g.*, from the Deep Web, RDF data sets, or relational tables. SPARQL queries are expressed in terms of the global schema and posed against the SemLAV mediator. A wrapper is specific for a data source, and retrieves data on demand; the retrieved data are transformed to match the global schema. Wrappers can be generated by tools like Karma [15] or OPAL [9]. The global schema is the interface between users and the data sources.

2.1 SemLAV Overview

Given a query and a set of views, SemLAV computes a ranked set of relevant views for answering the query, no *statistics* are used to rank the views. Relevant views are ranked based on the number of triple patterns of the original query that each view covers [13]. Views are materialized by calling the wrappers, and each time a new view is fully materialized, the original query is executed.

The benefits of SemLAV are illustrated in the following example [8]. Suppose SemLAV global schema comprises different RDF vocabularies, *e.g.*, foaf [5] and

[5] `http://xmlns.com/foaf/0.1/`

```
prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#>
prefix foaf:    <http://xmlns.com/foaf/0.1/>

SELECT DISTINCT *
WHERE {
  ?P foaf:member ?C .
  ?C rdfs:label "Semantic Web" .
  ?P foaf:knows ?WKP .
  ?WKP foaf:name ?N .
   FILTER (?N="Dalai Lama" || ?N="Barack Obama" || ?N="Rihanna")
}
```

Fig. 2: A SPARQL query over Deep Web and Linked Data

rdfs [6]. Figure 2 presents a SPARQL query expressed using the global schema. Views are expressed as conjunctive queries, where RDF predicates are represented by binary predicates, *e.g.*, label(C,L) corresponds to *?C rdf:label ?L* and *?P foaf:name ?N* is expressed as name(P,N). Listing 1 defines five LAV views. Triple patterns in the query are also seen as binary predicates and BGPs are represented as conjunctive queries; the running SPARQL query is composed of four subgoals on the predicates: member(P,C), label(C, "Semantic Web"), knows(P,WKP), and name(WKP,N). The filter expression is modeled as a disjunction of atomic expressions on the equality comparison operator.

Listing 1: Views s1-s5 for Query Q

```
v1(P,A,I,C,L):-made(P,A),affiliation(P,I),member(P,C),label(C,L)
v2(A,T,P,N,C):-title(A,T),made(P,A),name(P,N),member(P,C)
v3(P,N,R,M):-name(P,N),name(R,M),knows(P,R)
v4(P,N,G,R,C):-name(P,N),gender(P,G),knows(P,R),member(P,C)
v5(P,N,R,C,L):-name(P,N),knows(P,R),member(P,C),label(C,L)
```

Given a subgoal *sg* of a conjunctive query, *e.g.*, label(C,"Semantic Web"), a view *v* is relevant for *sg*, if *sg* is part of the body of *v*, *e.g.*, v1(P,A,I,C,L) and v5(P,N,R,C,L) are relevant for label(C,"Semantic Web"). Table 1a presents the set of relevant views for each query subgoal of query in Figure 2.

SemLAV sorts relevant views according to the number of the subgoals of the query that the view defines, *e.g.*, view *v*5 is sorted first since it defines all the subgoals. Table 1b represents the sorted relevant views for query in Figure 2.

SemLAV identifies and ranks the relevant views of a query, and executes the query over the data collected from the relevant views. Different strategies can be followed to contact the views and load the data. For example, following a blocking strategy, views are contacted one by one in order, and a view is not contacted until all the data from the previous contacted view have been downloaded completely. This is the strategy followed by SemLAV, which is illustrated in the Figure 3a, we can see that this strategy can be blocking if the first view is huge. While the view v5 is loading we are not able to perform the query. This blocking issue can have a negative impact on the performance of the query en-

[6] "http://www.w3.org/2000/01/rdf-schema

Table 1: Relevant views of query Q (cf. Figure 2), and views from Listing 1.

(a) Unsorted relevant views

member(P, C)	label(C, L)	knows(P, WKP)	name(WKP, N)
v1(P,A,I,C,L)	v1(P,A,I,C,L)	v3(P,N,R,M)	v2(A,T,P,N,C)
v2(A,T,P,N,C)	v5(P,N,R,C,L)	v4(P,N,G,R,C)	v3(P,N,R,M)
v4(P,N,G,R,C)		v5(P,N,R,C,L	v4(P,N,G,R,C)
v5(P,N,R,C,L)			v5(P,N,R,C,L)

(b) Sorted relevant views

member(P, C)	label(C, L)	knows(P, WKP)	name(WKP, N)
v5(P,N,R,C,L)	v5(P,N,R,C,L)	v5(P,N,R,C,L)	v5(P,N,R,C,L)
v4(P,N,G,R,C)	v1(P,A,I,C,L)	v4(P,N,G,R,C)	v4(P,N,G,R,C)
v1(P,A,I,C,L)		v3(P,N,R,M)	v2(A,T,P,N,C)
v2(A,T,P,N,C)			v3(P,N,R,M)

gine if the performance is measured in terms of the number of answers produced per unit of time, *i.e.*, throughput.

To illustrate this problem, consider Figure 3a, where v5 is loaded first. Even if v5 covers all the query subgoals, loading v5 first reduces the throughput, because v5 is the biggest view and does not contribute to the result. On the other hand, loading both v1 and v4, which together cover all the subgoals takes less time and may produce query answers. If relevant views were loaded in parallel following a non-blocking strategy, this situation would not affect the query engine performance. This solution is illustrated in Figure 3b, where there are five threads and each of them loads one of the first five top ranked views at the time; views are allocated in *different* threads. Time to load v5 is greater than the time required to load v4 and v1 in parallel. Additionally, v4 and v1 cover all the subgoals of our running query; thus, answers are produced before loading v5 completely.

We propose a non-blocking strategy for executing SPARQL queries against views. Like SemLAV, this approach does not rely on statistics to rank and select the relevant views. The proposed strategy prevents the query engine from getting blocked until all the data are retrieved from the relevant views.

3 Our Approach

A non-blocking strategy to access the views in a parallel fashion is defined. Although this strategy improves the performance of a query engine, loading the retrieved data into the integrated RDF graph in parallel, may generate concurrency problems, *i.e.*, many processes may simultaneously add data to the integrated RDF graph. So, we define a new concurrent model for RDF, and we propose a non-blocking query execution strategy able to adapt query execution to different criteria, *e.g.*, a query is executed after a certain number of triples

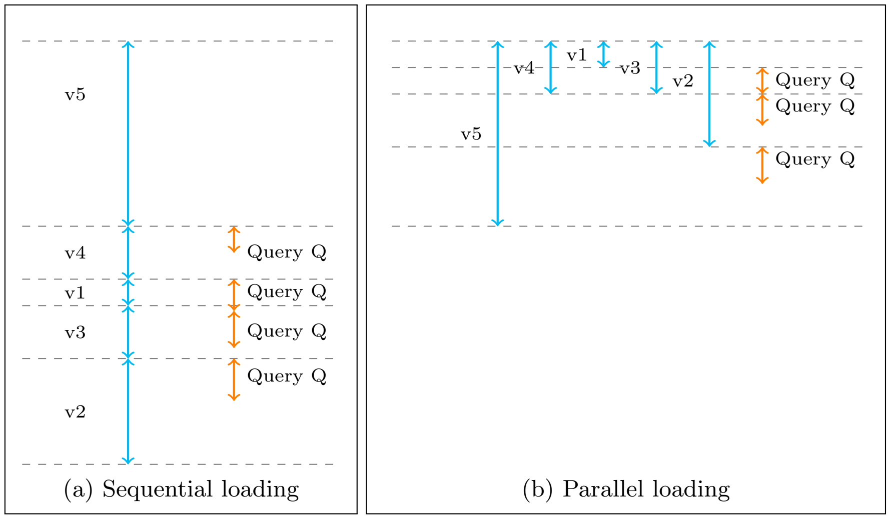


Fig. 3: Views loading and Query execution. For sequential loading just one thread is used, while for parallel loading five threads are used

are loaded into the integrated RDF graph. We implement the concurrency model and the non-blocking query execution strategy on top of Jena 2.7.4 [7] .

3.1 A Concurrency Model for the Integrated RDF Graph

Regarding our approach, we need a model that can handle concurrent insertions. However, RDF stores like Jena do not handle concurrent insertions, they are only able to favor one type of operation, *e.g.*, reads or insertions. This strategy is implemented thanks to locks, but read and insert locks are mutually exclusive, *i.e.*, they cannot be simultaneously activated. Existing RDF stores assume that there are more readers than writers and follow the multiple-readers/single-writer strategy (MRSW)[8]. According to MRSW, many readers may read simultaneously, while a writer must have exclusive access. MRSW assumes writers have the priority to keep data up-to-date. Nevertheless, in our proposed approach, data insertions are going to be more frequent than data reads. A reader is the query engine that accesses the integrated RDF graph during query execution, while writers are the wrappers of the relevant views which load the data into the integrated RDF graph. The query engine cannot execute the query more often than loading views into the integrated RDF graph, because executing the query is expensive, and doing so too often may lead to performance degradation.

In other words, our proposed approach prioritizes read operations over insertions, *i.e.*, a single-reader/multiple-writers strategy (SRMW) [14] is followed to

[7] http://jena.apache.org/
[8] https://jena.apache.org/documentation/notes/concurrency-howto.html

manage concurrency on the integrated RDF graph. So the reader, *e.g.*, a query execution engine, will have a higher priority rather than a writer, *e.g.*, a wrapper loading a view. Additionally, two insert locks cannot be activated at the same time due to the specification of the integrated RDF model. However, the query engine divides each view into blocks of n triples to allow for the loading of portions of several views at the same time. A lock is requested before starting a block loading, and it is released after n triples have been loaded completely. In our example, the first block of v5 is loaded, then the first block of v4, and to load the second block of v5, it may be necessary to wait until all the first blocks of the currently loading views are already loaded. However, this order may fluctuate depending on the system time allocation among the threads.

3.2 A Non-Blocking Strategy for SPARQL Query Execution

We implement a non-blocking strategy that is able to execute a query according to the following criteria; the selection of the criteria can be either configured or provided by the user during query execution.

- View dependent: the reader is woken up after a new view is loaded; thus, if v is a new loaded view, then the query engine will re-execute the query against the integrated RDF graph. If enough data is loaded into the integrated RDF graph from v, then the query engine will be able to generate new results when it is executed. This criterion is also implemented by SemLAV.
- Time dependent: the reader is woken up after a period of time t, *i.e.*, if t is n milliseconds, the query engine will re-execute the query against the RDF graph every n milliseconds. If enough data is loaded into the integrated RDF graph during the period t, the query engine will be able to generate new results. But, the concurrency model prioritizes the reader over writers; thus, if the writers are stopped and not able to load enough data into the integrated RDF graph, the query will be inefficiently executed.
- Data dependent: the reader is woken up after a certain number n of triples are inserted into the integrated RDF graph by the writers; thus, the query engine will re-execute the query against the RDF graph whenever n new triples are integrated. If the n new triples contribute to the results, then the query engine will be able to generate new answers when it is executed.
- Two-phase execution: the reader is woken up either after a period of time t or a certain number n of triples are inserted into the integrated RDF graph by the writers. In the first phase, the reader performs ASK queries to check if new results can be produced, if the answer is `true`, the second phase is launched. The second phase strategy will directly execute the query, then the reader will be woken up either after a period of time t or a certain number n of new triples have been inserted into the integrated RDF graph.

4 Experimental Evaluation

The Berlin SPARQL Benchmark (BSBM) [5], and queries and views proposed by Espinola-Castillo [6] are used to compare the performance of parallel SemLAV with respect to SemLAV. Our goal is to reproduce the experiments reported by Montoya et al. [13]; therefore, we used the Berlin Benchmark dataset composed of 10,000,736 triples using a scale factor of 28,211 products, 16 out of 18 queries, and nine out of the ten defined views proposed by Espinola-Castillo [6]. In SemLAV experiments, some queries and views were not considered because they included constants and some of the evaluated rewriters only process queries with variables. Five additional views were defined to cover all the predicates in the evaluated queries, *i.e.*, 14 views were evaluated. Furthermore, 476 views were produced by horizontally partitioning each original view into 34 parts, such that each part produces 1/34 of the answers given by the original view.

Queries and views are described in Tables 2a and 2b. The size of the complete answer is computed by including all the views into the Jena RDF triple store and by executing the queries against this centralized RDF dataset. The Jena 2.7.4 library with main memory setup is used to store and query the integrated RDF graphs. We executed parallel SemLAV with a timeout of 10 minutes.

Table 2: Queries and their answer size, number of subgoals, and views size, source [13]

(a) Query information

Query	Answer Size	# Subgoals
Q1	6.68E+07	5
Q2	5.99E+05	12
Q4	2.87E+02	2
Q5	5.64E+05	4
Q6	1.97E+05	3
Q8	5.64E+05	3
Q9	2.82E+04	1
Q10	2.99E+06	3
Q11	2.99E+06	2
Q12	5.99E+05	4
Q13	5.99E+05	2
Q14	5.64E+05	3
Q15	2.82E+05	5
Q16	2.82E+05	3
Q17	1.97E+05	2
Q18	5.64E+05	4

(b) Views size

Views	Size
V1-V34	201,250
V35-V68	153,523
V69-V102	53,370
V103-V136	26,572
V137-V170	5,402
V171-V204	66,047
V205-V238	40,146
V239-V272	113,756
V273-V306	24,891
V307-V340	11,594
V341-V374	5,402
V375-V408	5,402
V409-V442	78,594
V443-V476	99,237
V477-V510	1,087,281

Experiments are also run on the same platform than SemLAV experiments, *i.e.*, on a Linux server with 128 GB of memory, 124 processors where 20 GB of RAM are allocated for the experiments. Wrappers are implemented for each view and to load data from RDF files, *i.e.*, 476 wrappers are available.

4.1 Implementation

We use critical section and lock to implement the single-reader/multiple-writers SRMW concurrency model in Jena 2.7.4. The number of threads impacts the SPARQL engine performance; thus, we consider this number as one of the independent parameters of our study.

Table 3: Result of SemLAV and parallel SemLAV on BSBM using the View Dependent Criterion with 20 threads (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	604,254	30,481	**88.7036**	7
2	600,656	260,333	0.9823	66	605,729	**72,515**	**0.9883**	16
4	660,938	104,501	0.0004	47	359,635	288,558	**0.0008**	20
5	632,809	116,037	0.8916	28	457,269	257,097	**1.2339**	14
6	625,173	43,306	0.1892	24	273,662	211,313	**0.7203**	9
8	627,612	5,393	0.8990	42	318,475	24,877	**1.7716**	7
9	5,107	1,235	5.5240	18	2,453	1,839	**11.5006**	3
10	607,841	9,810	4.9243	44	439,562	32,438	**6.8094**	15
11	601,042	8,352	4.9800	43	105,684	31,660	**28.3219**	6
12	609,509	5,784	0.9822	121	372,481	15,542	**1.6072**	16
13	671,893	183,844	0.8910	124	392,147	**41,799**	**1.5266**	20
14	636,387	29,201	0.5419	24	333,754	201,864	**1.6905**	14
15	645,172	2,911	0.4373	37	388,061	20,016	**0.7270**	18
16	648,826	2,531	0.4348	46	306,694	15,390	**0.9198**	7
17	644,090	1,504	0.3060	32	278,330	5,894	**0.7082**	7
18	651,094	> 600,000	0.0000	12	509,646	**259,598**	**1.1071**	13

4.2 Impact of the Non-Blocking Query Execution Criteria

The goal of the experiment is to study the impact of the non-blocking query execution criteria on the query engine performance. We hypothesize that parallel SemLAV will outperform SemLAV in terms of throughput and time for the first answer. We measure the following metrics: *i*) total time (TT) in milliseconds; *ii*) time for first answer (TFA) in milliseconds; *iii*) throughput (answer/millisecond); and *iv*) number of times the original query is executed (#EQ).

We evaluate parallel SemLAV for the non-blocking query execution criteria defined in Section 3 with different number of threads, *i.e.*, the number of writers and the configuration of the non-blocking query execution strategy. We use setups with different number of threads 5, 10, and 20. Results suggest that 20 threads is the best number for writers. All the results are available at the project web site `https://sites.google.com/site/semanticlav`.

Table 4: Result of SemLAV and parallel SemLAV on BSBM using the View Dependent Criterion with 5 threads (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	601,221	13,235	35.3143	8
2	600,656	260,333	0.9823	66	646,416	**87,166**	0.9261	25
4	660,938	104,501	0.0004	47	406,008	**91,383**	**0.0007**	50
5	632,809	116,037	0.8916	28	601,055	**88,752**	**0.9387**	29
6	625,173	43,306	0.1892	24	317,213	61,451	**0.6214**	25
8	627,612	5,393	0.8990	42	410,306	7,202	**1.3751**	12
9	5,107	1,235	5.5240	18	2,687	**987**	**10.4991**	4
10	607,841	9,810	4.9243	44	631,503	11,438	4.7398	31
11	601,042	8,352	4.9800	43	300,244	9,879	**9.9691**	13
12	609,509	5,784	0.9822	121	508,837	9,048	**1.1765**	37
13	671,893	183,844	0.8910	124	532,783	**54,758**	**1.1236**	40
14	636,387	29,201	0.5419	24	463,967	62,251	**1.2161**	28
15	645,172	2,911	0.4373	37	600,885	8,390	**0.4695**	36
16	648,826	2,531	0.4348	46	462,310	4,820	**0.6102**	12
17	644,090	1,504	0.3060	32	311,895	2,533	**0.6320**	17
18	651,094	$>$ 600,000	0.0000	12	600,102	**264,917**	**0.9402**	37

Table 5: Result of BSBM over SemLAV and parallel SemLAV using the View Dependent Criterion with 10 threads (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	602,508	17,819	**41.3346**	8
2	600,656	260,333	0.9823	66	608,174	**70,504**	**0.9843**	25
4	660,938	104,501	0.0004	47	332,060	127,329	**0.0009**	50
5	632,809	116,037	0.8916	28	505,404	128,097	**1.1164**	29
6	625,173	43,306	0.1892	24	272,134	98,736	**0.7243**	25
8	627,612	5,393	0.8990	42	323,938	11,994	**1.7418**	12
9	5,107	1,235	5.5240	18	2,479	1,489	**11.3800**	4
10	607,841	9,810	4.9243	44	601,192	17,710	**4.9787**	31
11	601,042	8,352	4.9800	43	168,108	16,997	**17.8051**	13
12	609,509	5,784	0.9822	121	390,470	11,081	**1.5331**	37
13	671,893	183,844	0.8910	124	409,106	**39,892**	**1.4633**	40
14	636,387	29,201	0.5419	24	326,745	91,049	**1.7268**	28
15	645,172	2,911	0.4373	37	496,533	11,419	**0.5682**	36
16	648,826	2,531	0.4348	46	321,641	9,723	**0.8771**	12
17	644,090	1,504	0.3060	32	252,595	3,643	**0.7803**	17
18	651,094	$>$ 600,000	0.0000	12	600,785	**221,434**	**0.9391**	37

The View Dependent Criterion: The thread which executes the query is woken up when a new view is loaded. Table 3 shows the result of SemLAV and parallel SemLAV using the view strategy, *i.e.*, re-execute the query after a new view is loaded. Parallel SemLAV outperforms SemLAV in terms of throughput and total execution time. But surprisingly, the time for first answer is increased, for all queries except queries 2, 13, and 18; for these queries the time for the first answer is at most half of the SemLAV time. In most queries the time for first answer is increased because the number of times the original query is executed (#EQ) in parallel SemLAV is less than in SemLAV; furthermore, parallel SemLAV breaks the views ranking established by SemLAV, *i.e.*, SemLAV starts by loading the view ranked in first place and executes the query. However, parallel SemLAV loads views in parallel, and the query is re-executed when a new view is loaded, which is not necessarily the first ranked view by SemLAV. In setups with 5 and 10 threads, the time for first answer is better than for 20 threads, but the throughput is lower as shown in Tables 4 and 5.

Table 6: Result of BSBM over SemLAV and parallel SemLAV using the Time Dependent Criterion with 20 threads; queries are executed every 500 msecs (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	604,465	28,033	**67.3762**	16
2	600,656	260,333	0.9823	66	602,164	**73,074**	**0.9941**	17
4	660,938	104,501	0.0004	47	370,372	262,367	**0.0008**	102
5	632,809	116,037	0.8916	28	465,548	254,253	**1.2119**	27
6	625,173	43,306	0.1892	24	266,556	184,145	**0.7395**	83
8	627,612	5,393	0.8990	42	334,311	18,176	**1.6877**	17
9	5,107	1,235	5.5240	18	2,343	1,772	**12.0405**	4
10	607,841	9,810	4.9243	44	460,109	31,589	**6.5054**	28
11	601,042	8,352	4.9800	43	114,680	23,886	**26.1002**	19
12	609,509	5,784	0.9822	121	357,470	15,481	**1.6746**	22
13	671,893	183,844	0.8910	124	363,735	**41,237**	**1.6458**	24
14	636,387	29,201	0.5419	24	305,013	161,527	**1.8498**	94
15	645,172	2,911	0.4373	37	412,315	20,019	**0.6842**	23
16	648,826	2,531	0.4348	46	302,547	12,336	**0.9325**	14
17	644,090	1,504	0.3060	32	235,062	5,910	**0.8386**	21
18	651,094	> 600,000	0.0000	12	509,085	**276,665**	**1.1083**	99

The Time Dependent Criterion: The thread which executes the query is woken up each 500 milliseconds. Table 6 shows the result of SemLAV and parallel SemLAV using the time dependent strategy for 20 threads. The results also show that parallel SemLAV outperforms SemLAV in terms of throughput and total execution time; however, the time for first results is increased as when the view dependent criterion is executed.

Table 7: Result of BSBM over SemLAV and parallel SemLAV using the Data Dependent Criterion with 20 threads; queries are executed whenever 500 triples have been inserted in the integrated RDF graph (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	604,668	27,306	**62.8580**	10
2	600,656	260,333	0.9823	66	603,706	**68,132**	**0.9916**	14
4	660,938	104,501	0.0004	47	343,267	234,513	**0.0008**	21
5	632,809	116,037	0.8916	28	431,564	162,773	**1.3074**	16
6	625,173	43,306	0.1892	24	248,937	165,997	**0.7918**	14
8	627,612	5,393	0.8990	42	318,207	17,766	**1.7731**	8
9	5,107	1,235	5.5240	18	2,717	1,731	**10.3831**	4
10	607,841	9,810	4.9243	44	459,995	24,917	**6.5070**	15
11	601,042	8,352	4.9800	43	112,908	25,505	**26.5099**	7
12	609,509	5,784	0.9822	121	377,970	15,762	**1.5838**	15
13	671,893	183,844	0.8910	124	385,730	**42,222**	**1.5520**	24
14	636,387	29,201	0.5419	24	304,364	163,948	**1.8538**	17
15	645,172	2,911	0.4373	37	410,031	13,808	**0.6880**	19
16	648,826	2,531	0.4348	46	315,466	13,349	**0.8943**	8
17	644,090	1,504	0.3060	32	297,911	4,792	**0.6616**	9
18	651,094	> 600,000	0.0000	12	520,845	**302,575**	**1.0833**	13

The Data Dependent Criterion: The query thread is woken up each time the integrated RDF graph grows up to 500 new triples. Table 7 shows the results of SemLAV and parallel SemLAV using data dependent strategy for 20 threads. As in previous experiments, parallel SemLAV outperforms SemLAV in terms of throughput and total execution time for all queries; but the time for the first result is increased for the majority of the queries.

The Two-phase Criterion: The first phase of this strategy performs an `ASK` query and when it returns `true`, the second phase is conducted. First, the second phase executes the original query, then the query engine will be woken up either each n milliseconds or when n triples are inserted into the integrated RDF graph. Table 8 reports on the results for the two-phase strategy when the query is executed whenever 500 triples are inserted into the integrated RDF graph. Parallel SemLAV outperforms SemLAV in terms of throughput for all the queries, but throughput values of parallel SemLAV are lower than in previous experiments.

Table 8: Result of SemLAV and parallel SemLAV on BSBM using the Two-phase Criterion with 20 threads; queries are executed whenever 500 triples have been inserted in the integrated RDF graph (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	SemLAV				parallel SemLAV			
Query	TT	TFA	Throughput	#EQ	TT	TFA	Throughput	#EQ
1	606,697	6,370	37.3501	15	604,693	26,624	**62.4690**	4
2	600,656	260,333	0.9823	66	603,290	**72,463**	**0.9923**	8
4	660,938	104,501	0.0004	47	358,149	261,954	**0.0008**	11
5	632,809	116,037	0.8916	28	441,166	169,437	**1.2789**	13
6	625,173	43,306	0.1892	24	275,440	186,320	**0.7156**	6
8	627,612	5,393	0.8990	42	329,872	24,852	**1.7104**	7
9	5,107	1,235	5.5240	18	2,572	1,966	**10.9685**	3
10	607,841	9,810	4.9243	44	475,523	25,193	**6.2945**	15
11	601,042	8,352	4.9800	43	111,739	25,490	**26.7872**	7
12	609,509	5,784	0.9822	121	396,899	16,209	**1.5083**	14
13	671,893	183,844	0.8910	124	369,586	**44,197**	**1.6197**	10
14	636,387	29,201	0.5419	24	308,277	155,879	**1.8302**	10
15	645,172	2,911	0.4373	37	400,752	14,299	**0.7040**	18
16	648,826	2,531	0.4348	46	330,846	12,741	**0.8527**	8
17	644,090	1,504	0.3060	32	274,087	5,984	**0.7192**	8
18	651,094	> 600,000	0.0000	12	517,814	**285,958**	**1.0896**	13

Table 9: Throughput of SemLAV and parallel SemLAV (PS) using the Data-Dependent Criterion each 500 triples (DDC), Time-Dependent Criterion each 500 milliseconds (TDC), and Two-phase Criterion that combines ASK queries with DDC. With 20 threads for each criterion (**bold** font is used to highlight the values where parallel SemLAV outperforms SemLAV)

	Throughput				
Query	SemLAV	PS	PS+DDC	PS+TDC	PS+DDC+ASK
1	37.3501	**88.7036**	62.8580	67.3762	62.4690
2	0.9823	0.9883	0.9916	**0.9941**	0.9923
4	0.0004	**0.0008**	**0.0008**	**0.0008**	**0.0008**
5	0.8916	1.2339	**1.3074**	1.2119	1.2789
6	0.1892	0.7203	**0.7918**	0.7395	0.7156
8	0.8990	1.7716	**1.7731**	1.6877	1.7104
9	5.5240	11.5006	10.3831	**12.0405**	10.9685
10	4.9243	**6.8094**	6.5070	6.5054	6.2945
11	4.9800	**28.3219**	26.5099	26.1002	26.7872
12	0.9822	1.6072	1.5838	**1.6746**	1.5083
13	0.8910	1.5266	1.5520	**1.6458**	1.6197
14	0.5419	1.6905	**1.8538**	1.8498	1.8302
15	0.4373	**0.7270**	0.6880	0.6842	0.7040
16	0.4348	0.9198	0.8943	**0.9325**	0.8527
17	0.3060	0.7082	0.6616	**0.8386**	0.7192
18	0.0000	1.1071	1.0833	**1.1083**	1.0896

4.3 Discussion

Table 9 summarizes the results of the throughput with 20 threads in the different empirical evaluations. In all experiments, parallel SemLAV outperforms SemLAV in terms of the throughput and total execution time. However, none of the defined execution criterion dominates other criterion. For instance, parallel SemLAV with query execution every 500 milliseconds is the best execution strategy for $query2$; whereas parallel SemLAV with execution strategy whenever 500 triples have been inserted into the integrated RDF graph is the most suitable strategy for $query5$. We repeat the experiments with different number of threads. In setup with 20 threads, parallel SemLAV outperforms SemLAV in terms of throughput and total execution time but it increases time for first answer. Preliminary results suggest that there is a tradeoff between throughput and time for first answer. To confirm these results, in the future, we plan to evaluate parallel SemLAV with different time and data setups.

5 Conclusions and Future Work

We tackle the problem of executing SPARQL queries against LAV views in a parallel fashion. The query execution model relies on an RDF graph that temporally materializes the data retrieved from the relevant views of a SPARQL query. The query engine respects a concurrency model that prioritizes the execution of queries against the integrated RDF graph over loading data from the views. Additionally, a non-blocking query execution strategy allows for the execution of a SPARQL query on an RDF graph depending on different criteria. Similarly than SemLAV, our proposed parallel query execution model, named parallel SemLAV, was implemented on top of Jena. We empirically compared parallel SemLAV and SemLAV in terms of the impact of the non-blocking strategy on the query engine throughput. The observed results suggest that independently of the criterion followed by the non-blocking query engine strategy, parallel SemLAV outperforms SemLAV in terms of throughput. One limitation of our current implementation is inherent from the techniques implemented by Jena to handle concurrent insertions in an RDF graph. To overcome this limitation, we plan to consider a graph database engine as the RDF store backend, in order to provide more robust concurrency management of the RDF graph for incremental query processing.

Acknowledgement

We thank Maxime Pauvert and Nicolas Brondin, both students of the Computer Science Department at the University of Nantes for implementing the non-blocking strategy.

References

1. Virtuoso sponger. White paper, OpenLink Software.

2. S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart. *Web Data Management*. Cambridge University Press, New York, NY, USA, 2011.
3. Y. Arvelo, B. Bonet, and M.-E. Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *AAAI*, pages 225–230. AAAI Press, 2006.
4. 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.
5. C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.
6. R. Castillo-Espinola. *Indexing RDF data using materialized SPARQL queries*. PhD thesis, Humboldt-Universität zu Berlin, 2012.
7. A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.
8. P. Folz, G. Montoya, H. Skaf-Molli, P. Molli, and M. Vidal. Semlav: Querying deep web and linked open data with SPARQL. In *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, pages 332–337, 2014.
9. T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, and C. Schallhart. OPAL: automated form understanding for the deep web. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 829–838, 2012.
10. T. Furche, G. Gottlob, G. Grasso, X. Guo, G. Orsi, C. Schallhart, and C. Wang. DIADEM: thousands of websites to a single database. *PVLDB*, 7(14):1845–1856, 2014.
11. B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the Deep Web. *Commun. ACM*, 50(5):94–101, 2007.
12. G. Konstantinidis and J. L. Ambite. Scalable query rewriting: a graph-based approach. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *SIGMOD Conference*, pages 97–108. ACM, 2011.
13. G. Montoya, L. D. Ibáñez, H. Skaf-Molli, P. Molli, and M.-E. Vidal. SemLAV: Local-As-View Mediation for SPARQL. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XIII, Lecture Notes in Computer Science, Vol. 8420*, pages 33–58, 2014.
14. G. L. Peterson and J. E. Burns. Concurrent reading while writing II: the multi-writer case. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 383–392, 1987.
15. M. Taheriyan, C. A. Knoblock, P. A. Szekely, and J. L. Ambite. Rapidly integrating services into the linked data cloud. In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, J. X. Parreira, J. Hendler, G. Schreiber, A. Bernstein, and E. Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 559–574. Springer, 2012.
16. J. D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.
17. R. Verborgh, O. Hartig, B. D. Meester, G. Haesendonck, L. D. Vocht, M. V. Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. V. de Walle. Querying datasets on the web with high availability. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, pages 180–196, 2014.
18. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

Appendix: Query Set

```
Q1: Find out the gene resource related to "ADRAR".
select ?gene ?p
where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p "ADRAR"^^<http://www.w3.org/2001/XMLSchema#string>
}
Q2: Find out the genes related to diabetes.
select ?gene ?o1 ?o2  where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene <http://bio2rdf.org/omim_vocabulary:refers-to> ?o1.
?o1   ?p2 ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2 }
Q3: Find out the genes related to diabetes.
select ?gene ?o1 ?o2 where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p1 ?o1.
?o1   <http://bio2rdf.org/omim_vocabulary:x-snomed>  ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2 }
Q4: Find the genes related to diabetes.
select ?gene ?o1 ?o2 where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p1 ?o1.
?o1   ?p2 ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2}
Q5: Find out the generic name ,title, side effect for all the anti-allergic agents.
select * where{
?drug <http://bio2rdf.org/sider_vocabulary:generic-name> ?generic.
?drug <http://purl.org/dc/terms/:title> ?drug_name .
?drug <http://bio2rdf.org/sider_vocabulary:side-effect> ?side.
?drug <http://bio2rdf.org/sider_vocabulary:pubchem-flat-compound-id> ?cpd.
?generic <http://purl.org/dc/terms/title> ?generic_name.
?side <http://purl.org/dc/terms/title> ?side_effect.
?drug_drugbank <http://bio2rdf.org/drugbank_vocabulary:category>
<http://bio2rdf.org/drugbank_vocabulary:Anti-Allergic-Agents>.
?drug_drugbank <http://bio2rdf.org/drugbank_vocabulary:x-pubchemcompound> ?cpd        }
Q6: Find out clinical phenotype features, general and specific functions, and omim articles about F8 gene.
select * where {
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/omim_vocabulary:Phenotype> .
?s <http://www.w3.org/2000/01/rdf-schema#label> ?o.
?s <http://bio2rdf.org/omim_vocabulary:clinical-features> ?clinicFeature.
?s <http://bio2rdf.org/omim_vocabulary:article> ?article.
?s <http://bio2rdf.org/omim_vocabulary:x-uniprot> ?protein.
?drug  <http://bio2rdf.org/drugbank_vocabulary:gene-name> "F8"^^<http://www.w3.org/2001/XMLSchema#string>.
?drug <http://bio2rdf.org/drugbank_vocabulary:x-uniprot> ?protein.
?drug  <http://bio2rdf.org/drugbank_vocabulary:general-function> ?genFunction.
?drug  <http://bio2rdf.org/drugbank_vocabulary:specific-function> ?speFunction }
Q7: Find out all the drugs, which are substrate of some enzyme, their category and reaction.
select * where {
?enzyme <http://bio2rdf.org/kegg_vocabulary:substrate> ?cpd.
?enzyme <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/kegg_vocabulary:Enzyme>.
?reaction <http://bio2rdf.org/kegg_vocabulary:enzyme>  ?enzyme.
?drug    <http://bio2rdf.org/drugbank_vocabulary:category> ?category.
?drug <http://purl.org/dc/terms:description> ?desc.
?drug <http://bio2rdf.org/drugbank_vocabulary:x-kegg> ?cpd }
Q8: Find out side effects and pathways of all the anticonvulsants medicine.
select * where{
?s1 <http://bio2rdf.org/drugbank_vocabulary:category>
<http://bio2rdf.org/drugbank_vocabulary:Anticonvulsants>.
?s1 <http://bio2rdf.org/drugbank_vocabulary:affected-organism> ?affected.
?s1 <http://bio2rdf.org/drugbank_vocabulary:x-pubchemcompound> ?cpd.
?drug <http://bio2rdf.org/sider_vocabulary:side-effect> ?side.
?drug <http://bio2rdf.org/sider_vocabulary:pubchem-flat-compound-id> ?cpd.
?side <http://purl.org/dc/terms/title> ?side_effect.
?s2 <http://bio2rdf.org/kegg_vocabulary:x-pubchem.compound> ?cpd.
?s2 <http://bio2rdf.org/kegg_vocabulary:pathway>  ?pathway }
```