OPT^{+}: A Monotonic Alternative to OPTIONAL in SPARQL

Sijin Cheng and Olaf Hartig

Dept. of Computer and Information Science (IDA), Linköping University, Sweden
<firstname>.<lastname>@liu.se

Abstract

Due to the OPTIONAL operator, the core fragment of the SPARQL query language is non-monotonic. That is, some solutions of a query result can be returned to the user only after having consulted all relevant parts of the queried dataset(s). This property presents an obstacle when developing query execution approaches that aim to reduce responses times rather than the overall query execution times. Reducing the response times—i.e., returning as many solutions as early as possible—is important in particular in Web-based client-server query processing scenarios in which network latencies dominate query execution times. Such scenarios are typical in the context of integration of Web data sources where a data integration component executes queries over a decentralized federation of such data sources. In this paper we introduce an alternative operator that is similar in spirit to OPTIONAL but without causing non-monotonicity. We show fundamental properties of this operator and observe that the downside of achieving the desired monotonicity property is a potentially significant increase in query result sizes. We study the extend of this trade-off in practice. Thereafter, we introduce different algorithms to implement the new operator and evaluate them regarding their potential to reduce response times.

1 Introduction

While the SPARQL query language has been designed primarily for queries over a centralized collection of RDF data, it also has become the prevalent language for declarative approaches to query RDF datasets in decentralized settings. In fact, in its latest version the SPARQL specification itself has been extended with a notion of subqueries to be executed over a remote dataset on a different server [15]. Other typical examples of adopting the SPARQL language to query decentralized RDF data are queries over federations of SPARQL endpoints [1, 17], over Linked Data on the Web [8, 18], and over data sources that expose RDF via some Linked Data Fragments interface [19].

A feature of SPARQL that is particularly interesting for these use cases is the OPTIONAL operator which allows users to indicate that specific parts of a query can be ignored if no corresponding data is available. This feature is important for querying and integrating decentralized data because, due to the autonomous nature of the data sources, we cannot always assume that their data is complete.

While the OPTIONAL operator is useful in this context in terms of expressiveness, it is unsuitable in terms of another property that is desirable for a language to query decentralized data; namely, such a language should enable a query execution engine to employ approaches that return as many elements of a query result as early as possible during the query execution process. This property is important because, due to network latencies, the execution times of queries over decentralized data are typically greater than in a centralized setting [2, 9, 10] and, thus, software applications that are based on such queries should be enabled to achieve low user-perceived response times by presenting at least a partial result soon after starting a query execution. Unfortunately, the OPTIONAL operator is an obstacle in this context because the operator makes the core fragment of SPARQL non-monotonic as illustrated by the following example.

Example 1. Assume a query execution engine executes the SPARQL query in Figure 1(a) over a dataset of a remote data source, and during this query ex-

```
PREFIX ex: <http://example.org/>
SELECT ?post ?text ?img WHERE {
              ?post ex:hasText  ?text
   OPTIONAL { ?post ex:hasImage ?img } }
```

(a)

```
ex:post1 ex:hasText "Good ..."
ex:post2 ex:hasText "I can..."
ex:post1 ex:hasImage ex:sun.png
```

(b)

Figure 1 Example query (left) and example data (right).

ecution, the engine receives the sequence of RDF triples listed in Figure 1(b). After having received the first two of these triples, the query engine may produce an intermediate query result that consists of two solution mappings:

$$\mu_1 = \{\, \texttt{?post} \rightarrow \texttt{ex:post1},\ \texttt{?text} \rightarrow \texttt{"Good ..."} \,\},$$
$$\mu_2 = \{\, \texttt{?post} \rightarrow \texttt{ex:post2},\ \texttt{?text} \rightarrow \texttt{"I can..."} \,\}.$$

However, after having received the complete sequence of triples, it turns out that μ_2 is a solution for the query but μ_1 is not; instead, the following new mapping is another solution in the (sound and complete) query result:

$$\mu_3 = \{\, \texttt{?post} \rightarrow \texttt{ex:post1},\ \texttt{?text} \rightarrow \texttt{"Good ..."},\ \texttt{?img} \rightarrow \texttt{ex:sun.png} \,\}.$$

This example shows that there may be solution mappings that are in the result of a query over a subset of data, but the result of the query over the complete dataset does not contain these mappings anymore.[1] We also notice that, due to this non-monotonic nature of the OPTIONAL operator, the query engine in the example cannot output the mapping μ_2—which is indeed a correct solution of the final query result—until the engine has received and processed all of the data that is relevant for the optional pattern of the query.

At this point, one may wonder: If we were aiming to reduce user-perceived response times of applications that query decentralized data, we might permit the query engine in the example to already output the solution mappings μ_1 and μ_2 as soon as these mappings have been produced; if it turns out later that these mappings can be extended based on data that matches the optional pattern (as it is the case for μ_1 in the example), then the engine may output the extended mapping(s) as well. Clearly, the final set of all solution mappings returned in this way may not anymore be a sound query result in terms of the definition of the OPTIONAL operator. However, the advantage of being able to return some solution mappings earlier is worth investigating because it appears to allow applications to query decentralized data using an OPTIONAL-like query feature based on which the user-perceived response times may be reduced.

In this paper we conduct such an investigation. To this end, we define a new operator that we call OPT$^+$ and that provides a formal foundation for the alternative query evaluation outlined above. Like the OPTIONAL operator, OPT$^+$ has two subpatterns, one of which is treated as mandatory and the other as optional. Informally, the result of an OPT$^+$ operator with two such

[1] For some queries that contain multiple OPTIONAL operators we may even observe cases in which the result for a subset of data contains a solution mapping that is not anymore in the result for a bigger subset but that is contained again in the result for the complete dataset.

subpatterns consists of all the solution mappings that also are in the result of the OPTIONAL operator with the same two subpatterns and, additionally, all the solution mappings that can be obtained from the mandatory subpattern but that are not in the result of the corresponding OPTIONAL operator. For instance, for a version of the query in Figure 1(a) in which the OPTIONAL operator is replaced by OPT$^+$, the query result over the triples in Figure 1(b) consists of all three solution mappings mentioned in Example 1 (i.e., not only μ_2 and μ_3, but also μ_1).

It is not difficult to see that the OPT$^+$ operator is monotonic and the price we have to pay for achieving this monotonicity is a possible increase in the size of query results (when compared to using OPTIONAL). Our aim in this paper is to achieve an understanding of this trade-off, including the potential gain of reduced query response times. Hence, we focus on the following two research questions.

RQ1. How significant is the increase of the size of query results in practice when using the OPT$^+$ operator instead of OPTIONAL?

RQ2. How suitable is the OPT$^+$ operator in terms of its potential for query executions that return as many solutions of query results as early as possible?

To address these questions we make the following contributions.

1. We define and analyze the OPT$^+$ operator formally. Our analysis shows properties (monotonicity, expressive power) that the core fragment of SPARQL has if the OPTIONAL operator is replaced by OPT$^+$ (Section 3).
2. We provide an empirical analysis based on 10 real-world query logs (with an overall of ca. 34M OPTIONAL queries) that shows how OPTIONAL is used in practice (Section 4), and we compare the result sizes obtained by queries in these logs when using either OPTIONAL or OPT$^+$ (Section 5).
3. We introduce two different approaches to implement the OPT$^+$ operator natively in a physical query execution plan (Section 6) and evaluate them experimentally. In addition to showing their respective potential for returning as many solution mappings of query results as early as possible, this evaluation shows—to our surprise—that none of these approaches can achieve a significant advantage over an approach to implement the OPTIONAL operator (Section 7).

Before focusing on these contributions, we introduce existing relevant definitions and results (Section 2). The source code and the data used for the work in this paper is available at https://github.com/hartig/OptPlusExperiments.

2 Preliminaries

This section defines the relevant concepts of RDF and SPARQL formally.

We assume four pairwise disjoint, countably infinite sets: $\mathcal{U}$ (URIs), $\mathcal{B}$ (blank nodes), $\mathcal{L}$ (literals), and $\mathcal{V}$ (variables). An *RDF triple* is a tuple $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. An *RDF graph* is a set of such triples.

For SPARQL we focus on the core fragment of the language and adopt the formalization approach of this fragment as introduced by Pérez et al. [13]; that is, we use the algebraic syntax and the multiset query semantics defined by Pérez et al. [13]. We emphasize that this formalization and the official specification of SPARQL [6] are equivalent in terms of expressive power [3]. Hence, the foundations of OPT$^+$ as presented in this paper can be easily carried over to the syntax and semantics of SPARQL as found in the specification. Moreover, focusing on the core fragment of SPARQL is not a limitation either because all language features are built on top of this core fragment [6].

The algebraic syntax of SPARQL defines *SPARQL expressions* recursively: i) A tuple $(s, p, o) \in (\mathcal{V} \cup \mathcal{U}) \times (\mathcal{V} \cup \mathcal{U}) \times (\mathcal{V} \cup \mathcal{U} \cup \mathcal{L})$ is a SPARQL expression called a *triple pattern*.[2] ii) If P_1 and P_2 are SPARQL expressions, then so are $(P_1 \text{ AND } P_2)$, $(P_1 \text{ UNION } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ FILTER } R)$, where R is a filter condition [13].[3] To denote the set of all variables in all triple patterns of a SPARQL expression P we write $\mathrm{vars}(P)$.

The result of evaluating a SPARQL expression takes the form of a multiset of *solution mappings*; that is, partial functions $\mu\colon \mathcal{V} \to \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$. The subset of $\mathcal{V}$ for which such a mapping μ is defined is denoted by $\mathrm{dom}(\mu)$. Two solution mappings μ and μ' are *compatible* if for every variable $?v$ in $\mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$ we have that $\mu(?v) = \mu'(?v)$; in this case, the combination of μ and μ', denoted by $\mu \cup \mu'$, is also a solution mapping. Given a triple pattern tp and a solution mapping μ, we write $\mu[tp]$ to denote the triple pattern that we obtain by replacing the variables in tp according to μ. Notice that $\mu[tp]$ is an RDF triple if $\mathrm{vars}(tp) \subseteq \mathrm{dom}(\mu)$.

[2] For the sake of simplicity we do not permit blank nodes in triple patterns. In practice, each blank node in a SPARQL query can be replaced by a new variable.

[3] We do not define filter conditions in this paper because they are not relevant for our work. For a formal definition of their syntax and semantics refer to Pérez et al. [13].

The semantics of SPARQL expressions is defined based on a set of operators *over multisets* of solution mappings. For the sake of conciseness, we introduce only the set-specific versions of these operators and refer to Pérez et al.'s work for the multiset versions [13]. Given two sets of solution mappings, Ω and Ω', and a filter condition R, the operators *join* ($\bowtie$), *union* ($\cup$), *difference* ($\setminus$), and *selection* (σ) are defined as follows.

$$\begin{aligned}
\Omega \bowtie \Omega' &= \{\mu \cup \mu' \mid \mu \in \Omega,\ \mu' \in \Omega', \text{ and } \mu \text{ and } \mu' \text{ are compatible}\}\\
\Omega \cup \Omega' &= \{\mu \mid \mu \in \Omega \text{ or } \mu \in \Omega'\}\\
\Omega \setminus \Omega' &= \{\mu \in \Omega \mid \text{there exists no } \mu' \in \Omega' \text{ such that } \mu \text{ and } \mu' \text{ are compatible}\}\\
\sigma_R(\Omega) &= \{\mu \in \Omega \mid \mu \text{ satisfies } R \text{ [13]}\}
\end{aligned}$$

Now we are ready to define the semantics of SPARQL expressions:

Definition 1. Given a SPARQL expression P and an RDF graph G, the *evaluation of P over G*, denoted by $[\![P]\!]_G$, is defined recursively as follows:

1. If P is a triple pattern tp, then $[\![P]\!]_G$ is a multiset of solution mappings that consists of the solution mapping from the following set Ω and contains each of these mappings exactly once.

$$\Omega = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(tp) \text{ and } \mu[tp] \in G\}$$

2. If P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
3. If P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.
4. If P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = ([\![P_1]\!]_G \bowtie [\![P_2]\!]_G) \cup ([\![P_1]\!]_G \setminus [\![P_2]\!]_G)$.
5. If P is $(P' \text{ FILTER } R)$, then $[\![P]\!]_G = \sigma_R\big([\![P']\!]_G\big)$.

A SPARQL expression P is *monotonic* if for every pair G_1, G_2 of RDF graphs such that $G_1 \subseteq G_2$, it holds that $[\![P]\!]_{G_1}$ is a sub-multiset of $[\![P]\!]_{G_2}$. A SPARQL expression P is *satisfiable* if there exists an RDF graph G such that $[\![P]\!]_G$ contains at least one solution mapping. It is trivial to show that every SPARQL expression that is not satisfiable is monotonic, and every SPARQL expression that is not monotonic is satisfiable. Furthermore, it is well known that both satisfiability and monotonicity of SPARQL are undecidable, and that SPARQL expressions without OPT are monotonic (see, e.g., [7]).

3 Formal Foundation

To define the OPT$^+$ operator formally we extend the notion of a SPARQL expression by adding the following case to the recursive definition: iii) If P_1 and P_2 are SPARQL expressions, then $(P_1 \text{ OPT}^+ P_2)$ is a SPARQL expression.

Now, we need to define the semantics of SPARQL expressions with OPT$^+$. To this end, we extend the recursive definition of the SPARQL evaluation function (cf. Definition 1) with an additional case for OPT$^+$.

Definition 2. For every RDF graph G and every SPARQL expression P of the form $(P_1 \text{ OPT}^+ P_2)$, we define that $[\![P]\!]_G = ([\![P_1]\!]_G \bowtie [\![P_2]\!]_G) \cup [\![P_1]\!]_G$.

Given these definitions, we can now show some fundamental properties of SPARQL expressions with OPT$^+$. We begin with a simple rewriting rule.

Proposition 1. *For every two SPARQL expressions P_1 and P_2, the following equivalence holds: $(P_1 \text{ OPT}^+ P_2) \equiv ((P_1 \text{ AND } P_2) \text{ UNION } P_1)$.*

Proof. The equivalence is a trivial consequence of Definitions 1 and 2. □

As a corollary of this equivalence we can show that adding the OPT$^+$ operator to SPARQL does not change the expressive power of the language.

Corollary 1. *For every SPARQL expression P, there exists a SPARQL expression P' such that $P \equiv P'$ and P' does not contain OPT$^+$.*

Proof (Sketch). The corollary can be shown by using the rewriting rule of Proposition 1. That is, given a SPARQL expression with OPT$^+$, the rule can be applied repeatedly until all OPT$^+$ operators have been replaced using AND and UNION. □

A natural question at this point is: Given that we can capture the idea of OPT$^+$ by using AND and UNION, why do we need the OPT$^+$ operator at all? There are two reasons: First, having an explicit OPT$^+$ operator enables a query engine to use specific algorithms that implement this operator in a more efficient way than a generic combination of algorithms that implement AND and UNION, respectively. Our experiments in Section 7 verify this benefit. The second reason is that expressions that capture the notion of OPT$^+$ by using AND and UNION may become unmanageably large. More specifically, there exist expressions with OPT$^+$ for which the size of the equivalent expressions with AND and UNION are exponential in the size of the OPT$^+$ expressions. To show this formally in the following result we define the *size* of a SPARQL expression P, denoted by $|P|$, to be the number of triple patterns in P.

Proposition 2. *For every $n \geq 2$, there exists a SPARQL expression P that contains OPT$^+$ such that $|P| = n$ and for every equivalent expression P' without OPT$^+$ (i.e., $P \equiv P'$), it holds that $|P'| \geq 2^n - 1$.*

Proof. Consider a SPARQL expression P_n of the form

$$((\ldots((tp_1 \,\text{OPT}^+\, tp_2)\,\text{OPT}^+\, tp_3)\ldots)\,\text{OPT}^+\, tp_n).$$

That is, P_n contains a sequence of $n-1$ OPT$^+$ operators with an overall of n triple patterns that are all different from one another. Given P_n, we prove the proposition by induction on n.

In the *base case* ($n = 2$), P_2 is of the form $(tp_1\,\text{OPT}^+ tp_2)$. We rewrite P_2 into $P_2' = ((tp_1\,\text{AND}\, tp_2)\,\text{UNION}\, tp_1)$. Then, we have that $P_2' \equiv P_2$ (cf. Proposition 1) and $|P_2'| = 3 \geq 2^n - 1$. Similarly, every other expression P_2'' obtained by rewriting P_2' without using OPT$^+$ must contain tp_1 twice and tp_2 once (assuming $P_2'' \equiv P_2'$ and, thus, $P_2'' \equiv P_2$).

For the *induction step* ($n > 2$), P_n is of the form $(P_{n-1}\,\text{OPT}^+ tp_n)$ where P_{n-1} is $((\ldots((tp_1\,\text{OPT}^+\, tp_2)\,\text{OPT}^+\, tp_3)\ldots)\,\text{OPT}^+\, tp_{n-1})$. By the induction hypothesis, there exists a P_{n-1}' without OPT$^+$ such that $P_{n-1}' \equiv P_{n-1}$ and $\left|P_{n-1}'\right| \geq 2^{n-1} - 1$. Then, by using P_{n-1}' instead of P_{n-1}, we rewrite P_n into $P_n' = ((P_{n-1}'\,\text{AND}\, tp_n)\,\text{UNION}\, P_{n-1}')$, for which we know by Proposition 1 that $P_n \equiv P_n'$. Now, it remains to show that $|P_n'| \geq 2^n - 1$.

$$\begin{aligned} \left|P_n'\right| &= 2 \cdot \left|P_{n-1}'\right| + 1 \\ &\geq 2 \cdot (2^{n-1} - 1) + 1 \\ &= 2 \cdot 2^{n-1} - 2 + 1 \\ &= 2^n - 1. \end{aligned}$$ □

Proposition 2 shows that representing OPT$^+$ using AND and UNION may increase the size of the resulting expressions exponentially. We emphasize that this exponential increase is specific to expressions that contain sequences of OPT$^+$ operators. For instance, for expressions in which OPT$^+$ operators are nested, the increase is linear as shown by the following result.

Proposition 3. *For every SPARQL expression P of the form*

$$(P_n\, \mathit{OPT}^+\, (P_{n\text{-}1}\, \mathit{OPT}^+\, (\ldots(P_2\, \mathit{OPT}^+\, P_1)\ldots)))$$

in which no P_i contains OPT$^+$ (for all $i \in \{1, \ldots, n\}$), there exists a SPARQL expression P' without OPT$^+$ such that $P \equiv P'$ and $|P'| = 2 \cdot |P| - |P_1|$.

Proof. We show the proposition by induction. The base case ($n = 2$) follows trivially from Proposition 1. For the induction step ($n > 2$) we let P be (P_n OPT$^+$ Q) where Q is ($P_{n\text{–}1}$ OPT$^+$ (...(P_2 OPT$^+$ P_1)...)). By induction, there exists a SPARQL expression Q' without OPT$^+$ such that $Q \equiv Q'$ and $|Q'| = 2 \cdot |Q| - |P_1|$. By using $Q \equiv Q'$, we have $P \equiv (P_n$ OPT$^+$ $Q')$, and by Proposition 1, we have $P \equiv P'$ where P' is ((P_n AND Q') UNION P_n) with:

$$\begin{aligned} \left|P'\right| &= 2 \cdot |P_n| + \left|Q'\right| \\ &= 2 \cdot |P_n| + (2 \cdot |Q| - |P_1|) \\ &= 2 \cdot (|P_n| + |Q|) - |P_1| \\ &= 2 \cdot |P| - |P_1| \,. \end{aligned}$$

□

We complete our formal analysis of SPARQL expressions with OPT$^+$ by another corollary which follows from Proposition 1, and which shows that, by using OPT$^+$ instead of OPT, we achieve the desired monotonicity.

Corollary 2. *Every SPARQL expression without OPT (but possibly with OPT$^+$) is monotonic.*

Proof. It is not difficult to verify that for a SPARQL expression with OPT$^+$ but without OPT, there exists an equivalent expression without OPT$^+$ (Corollary 1) that does not contain OPT either. Then, the monotonicity follows from the monotonicity of SPARQL expressions that use only AND, UNION, and FILTER [7] (cf. Section 2). □

As a final remark, we remind the reader of Pérez et al.'s results that show that the complexity of the evaluation problem of expressions with AND, UNION, and FILTER is NP-complete and it becomes PSPACE-complete if we add OPT [14]. Hence, by Corollary 1, the complexity of the evaluation problem drops to NP if we use OPT$^+$ instead of OPT.

4 Usage of OPTIONAL in Practice

To understand the potential consequences of replacing OPTIONAL by the OPT$^+$ operator in queries over decentralized data it is important at first to understand how OPTIONAL is used in practice. To achieve such an understanding we have analyzed 10 real-world query logs with an overall of more than 34M SPARQL queries with OPTIONAL. This section describes our analysis and the results.

Table 1 Information about the query logs used for our analysis.

Name of log	Endpoint/Dataset	Source	Period of time
$DBP_{3.3}$	DBpedia v.3.3	[11]	2009-07-01–2009-07-13
$DBP_{3.4}$	DBpedia v.3.4	[11]	2009-11-18–2010-02-01
$DBP_{3.5.1}$	DBpedia v.3.5.1	[11, 16]	2010-04-30–2010-07-20 [16], 2010-05-28–2010-07-20 [11]
$DBP_{3.6}$	DBpedia v.3.6	[11]	2011-01-23–2011-06-10
$DBP_{3.8}$	DBpedia v.3.8	[11]	2012-07-26–2012-11-01 and 2013-06-30–2013-08-07
SWDF	Sem.Web Dog Food	[11, 16]	2008-11-01–2013-01-22 [11], 2014-04-16–2014-11-12 [16]
LGD	LinkedGeoData	[11, 16]	2010-11-24–2011-07-06 [16], 2011-05-23–2011-11-24 and 2012-10-02–2014-01-12 [11]
BM	British Museum	[16]	2014-11-08–2014-12-01
WD_{all}	Wikidata	[12]	2017-06-12 – 2017-09-03
WD_{org}	Wikidata	[12]	2017-06-12 – 2017-09-03

4.1 Query Logs

The query logs that we use are from three different sources. That is, we use logs from the USEWOD datasets [11], from the LSQ dataset [16], and from a dataset made available as part of a study of Wikidata [12]. Each of these logs is from a different public SPARQL endpoint that provides (or provided) SPARQL-based query access to a respective RDF dataset. Hence, each log contains SPARQL queries that have been sent to the corresponding SPARQL endpoint during the period of time covered by the log. There are three logs in the USEWOD datasets for which the LSQ dataset contains another log from the same endpoint, respectively. For our analysis we have combined the corresponding logs into one. Table 1 provides provenance information about the logs that we use, including the respective time periods covered by each of the logs. The datasets related to these logs are the following:

- DBpedia contains data extracted from structured information in the Wikipedia. Our analysis covers logs for five versions of DBpedia.
- The Semantic Web Dog Food dataset describes conferences and workshops in the Semantic Web field, including data about corresponding publications and authors.
- LinkedGeoData is a large spatial knowledge base consisting of data collected by the OpenStreetMap effort.
- The British Museum dataset is a collection of data provided by the British Museum.
- The Wikidata dataset is the result of collecting a large amount of structured knowledge across all Wikimedia projects and languages. While the WD_{all} log contains all queries that were issued to the Wikidata SPARQL endpoint during the specified period of time, the WD_{org} log is a subset

Table 2 Statistics about the number of queries with OPTIONAL in the logs and the subsets of these queries that could be parsed and, thus, can be analyzed. The percentages in the table are calculated w.r.t. the total number of queries in the respective log.

log	number of all queries	number of queries with OPTIONAL		distinct queries w/ OPTIONAL		parsed queries with OPTIONAL		distinct parsed w/ OPTIONAL	
DBP$_{3.3}$	2,937,357	438,844	14.9%	325,957	11.1%	430,164	14.6%	322,053	11.0%
DBP$_{3.4}$	2,640,253	472,295	17.9%	136,022	5.2%	461,556	17.5%	130,692	4.9%
DBP$_{3.5.1}$	6,036,916	1,740,941	28.8%	630,004	10.4%	1,599,087	26.5%	574,720	9.5%
DBP$_{3.6}$	8,384,677	2,308,730	27.5%	878,262	10.5%	1,429,332	17.0%	688,058	8.2%
DBP$_{3.8}$	11,909,344	2,114,092	17.8%	960,060	8.1%	1,541,483	16.6%	608,090	5.1%
WD$_{\text{all}}$	173,091,565	24,545,693	14.2%	3,049,023	1.8%	3,171,721	1.8%	358,318	0.2%
WD$_{\text{org}}$	661,505	326,662	49.4%	99,722	15.1%	88,046	13.3%	29,113	4.4%
LGD	12,719,055	1,315,085	10.3%	307,954	2.4%	1,135,630	8.9%	100,406	0.8%
SWDF	99,165	34,267	34.6%	34,267	34.6%	6,433	6.5%	6,433	6.5%
BM	1,589,840	1,106,750	69.6%	101,103	6.4%	1,106,710	69.6%	101,064	6.4%
total:	220,069,677	34,403,359		6,522,374		10,970,162		2,918,941	

from which all queries have been removed that are assumed to be sent by bots [12]. It has been shown that the queries in WD$_{\text{org}}$ are structurally more diverse, whereas WD$_{\text{all}}$ contains many trivial queries [12].

4.2 Statistics Collection

To process the logs and to collect relevant statistics from them we have developed a program that uses the Apache Jena framework[4] to parse and to analyze SPARQL queries. As a first processing step, this program simply extracts all the query strings from each of the logs. Thereafter, for each query string, the program records whether the string contains the keyword OPTIONAL and whether the exact same string has been processed before. We use the latter as a simple approach to identify duplicates. Next, if the query string contains the keyword OPTIONAL, the program tries to parse the string into an object representation of a SPARQL query. If the string can be parsed successfully, the resulting query object is analyzed to record statistics about the use of OPTIONAL in the given query. Hereafter, we call these queries *analyzed queries*.

4.3 Basic Statistics

Before going into the details of how exactly OPTIONAL is used in the queries, we refer to Table 2 which shows how many queries in the different logs use OPTIONAL and how many of them are analyzed queries. While the

[4] https://jena.apache.org/

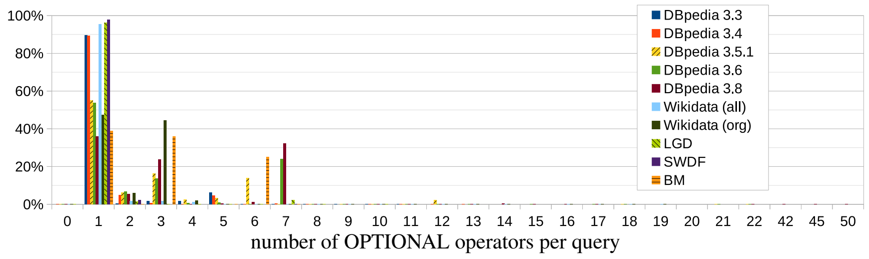


Figure 2 Percentage of all the analyzed queries with a given number of OPTIONALs.

percentage of queries with OPTIONAL differs significantly for the different logs (ranging from 10.3% to 69.6%), we observe that each log contains a non-negligible portion of them. Hence, the OPTIONAL operator is indeed used in practice. Another noteworthy observation is that for some logs (e.g., $\text{DBP}_{3.3}$, SWDF) almost all the queries with OPTIONAL are distinct whereas for other logs (e.g., WD_{all}, BM) there are many duplicates among these queries.

4.4 Number of OPTIONALs per Query

We now focus on the analyzed queries. First, we consider the number of OPTIONAL operators in these queries. Figure 2 illustrates, for each of the query logs, the percentage of all the analyzed queries that contain a given number of OPTIONAL operators. For all logs together, we observe that the majority of queries contains a single OPTIONAL operator only (namely, 69% of all the 10.9M analyzed queries; respectively, 53% of the 2.9M distinct queries).

There are some logs ($\text{DBP}_{3.3}$, $\text{DBP}_{3.4}$, WD_{all}, LGD, SWDF) in which almost all of the analyzed queries contain only one OPTIONAL. On the other hand, there also are logs ($\text{DBP}_{3.5.1}$, $\text{DBP}_{3.6}$, $\text{DBP}_{3.8}$, WD_{org}, BM) that contain a sizable fraction of analyzed queries with more than one OPTIONAL. For instance, for both $\text{DBP}_{3.8}$ and BM, more than 60% of the analyzed queries have more than one OPTIONAL, respectively. Among the 101K distinct analyzed queries in BM, 35% of them have 3 OPTIONALs and 28.8% have 6, and among the 608K distinct analyzed queries in $\text{DBP}_{3.8}$, 35% have 3 OPTIONALs and 47.8% have 7. The maximum are 50 OPTIONALs per query, which is the case for 16 of all 1.5M analyzed queries in $\text{DBP}_{3.8}$.

4.5 Sequences and Nesting of OPTIONALs

For each of the subsets of queries with multiple OPTIONAL operators, we now report how these operators are combined into sequences or by nesting (including combinations thereof).

Regarding sequences, only 3032 of all multi-OPTIONAL queries (across all the logs) do *not* contain a sequence! 99.9% contain one sequence, and 0.01% contain two separate sequences (no query contains more than two). The logs with the highest number of queries with sequences are DBP$_{3.8}$ (ca. 1.0M, or 64%, of the 1.5M queries in the log), DBP$_{3.5.1}$ (ca. 716K, 44.8%), BM (ca. 675K, 61.0%), and DBP$_{3.6}$ (ca. 661K, 46.2%). When it comes to the lengths of such sequences, considering all logs, we observe that the lengths of the longest sequence per query range from 1 (i.e., two OPTIONALs) to 49 (!), where most of these (longest) sequences are short (e.g., 98% have a length smaller than 7 and 50% have a length smaller than 3). Queries with long sequences (length > 10) are in WD$_{\text{all}}$, WD$_{\text{org}}$, DBP$_{3.5.1}$, DBP$_{3.6}$, and DBP$_{3.8}$. When considering only distinct queries, the observations regarding sequences are essentially the same.

In contrast to the very high number of multi-OPTIONAL queries with sequences, only very few queries contain nested OPTIONALs (namely, only 3803 of all 10.9M analyzed queries; respectively, only 1350 of all 2.9M distinct queries).

5 Result Size Increase in Practice

We now are ready to focus on research question RQ1 about the increase of query result sizes when using the OPT$^+$ operator instead of OPTIONAL. We answer this question based on an empirical analysis, which we describe in this section.

5.1 Method

Our approach to conduct this analysis has been to use queries obtained from some of the aforementioned query logs to create pairs of queries consisting of an OPTIONAL version and an OPT$^+$ version; then, we execute these queries over the corresponding dataset and compare the sizes of the query results.

We have created such a pair of queries for *every* distinct analyzed query in the selected logs (recall that these queries use the OPTIONAL operator). Given the WHERE clause of such a query, the first query for the corresponding new pair of queries is created by simply combining the WHERE clause with a SELECT clause of the form "`SELECT *`". This query becomes *the OPTIONAL query of the pair*. We use only the WHERE clause (i.e., the query pattern) of the original query from the log because additional query features such as DISTINCT and LIMIT are irrelevant for our analysis and may even introduce bias. The other query of the pair, called *the* OPT^+*-like query*, is created as follows. We copy the OPTIONAL query of the pair, replace every occurrence of OPTIONAL by OPT^+, and apply the rewriting rule of Proposition 1 repeatedly; then, we obtain a query that does not anymore contain any OPT^+ operator (nor OPTIONAL) but that is equivalent to the OPT^+-version of the OPTIONAL query. This rewriting is necessary because the systems that we use for executing the test queries are standard SPARQL systems and, thus, not aware of the OPT^+ operator. A downside of rewriting is that some of the resulting OPT^+-like queries are rather large (cf. Proposition 2).

For the analysis we have selected both Wikidata logs, $DBP_{3.5.1}$, and LGD. Hence, we have ca. 358K pairs of test queries from WD_{all}, ca. 29K pairs from WD_{org}, etc. To execute these test queries we either use the SPARQL endpoint of the corresponding dataset (Wikidata and LGD) or a local triple store loaded with the dataset (DBpedia v.3.5.1). If the execution of any of the two queries of a pair fails (e.g., a timeout error from the SPARQL endpoint), we ignore this pair. Otherwise, we record the respective size of the results of both queries in the pair and calculate both the *difference* between these two sizes (i.e., the number of additional solutions in the result of the OPT^+-like query) and the *increase factor* (i.e., the factor of how much greater the result size of the OPT^+-like query is).

5.2 Results

Table 3 summarizes the statistics that we have calculated from our measurements. We first focus on statistics that consider all pairs of test queries for which there were no errors (i.e., the first block of statistics in the table).

We observe that for a large fraction of query pairs, the two queries have the same result size. For instance, for ca. 66% of the pairs for $DBP_{3.5.1}$, the OPTIONAL query and the OPT^+-like query in the pair have a result of the same size (which, by the definition of OPT^+, means the two query results are equivalent). For WD_{org} and LGD it is even more than 98%, respectively. For the pairs of queries from WD_{org} for which there is a result size increase, the charts in Figure 3 illustrate the respective differences and the respective increase factors (both ordered from smaller to greater). We notice that

Table 3 Statistics about the increase of query result sizes.

log	% of pairs with same result size	at most 2x increase	at most 10x increase	at most 100x increase	at most 1000x increase	greatest increase	greatest difference
All pairs of test queries (for which there were no errors)							
$DBP_{3.5.1}$	65,74%	71,23%	80,86%	91,46%	99,57%	6144x	35.768
WD_{all}	67,19%	98,90%	99,88%	99,89%	99,90%	4806x	718.290
WD_{org}	98,73%	99,73%	99,95%	99,95%	99,95%	9.62x	455.588
LGD	98,46%	99,88%	100,00%	100,00%	100,00%	22.52x	19.682
Only the pairs whose OPTIONAL query did *not* contain sequences of OPTIONALs							
$DBP_{3.5.1}$	93,09%	100,00%	100,00%	100,00%	100,00%	2.00x	1.104
WD_{all}	66,04%	99.99%	100,00%	100,00%	100,00%	2.94x	263.919
WD_{org}	99,08%	99.99%	100,00%	100,00%	100,00%	2.12x	159.622
LGD	98,57%	100,00%	100,00%	100,00%	100,00%	2.00x	19.682
Only the pairs whose OPTIONAL query did contain sequences of OPTIONALs							
$DBP_{3.5.1}$	65,41%	70,88%	80,63%	91,36%	99,56%	6144x	35.768
WD_{all}	75,12%	91,35%	99,06%	99,17%	99,17%	4806x	718.290
WD_{org}	97,61%	98,88%	99,81%	99,81%	99,81%	9.62x	455.588
LGD	92,69%	93,67%	99,95%	100,00%	100,00%	22.52x	19.199

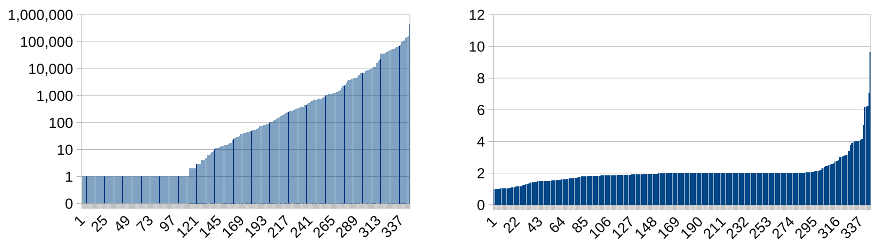


Figure 3 Query result size differences (left) and increase factors (right), both ordered from smaller to greater, for the test queries of WD_{org} for which the result sizes differ.

there is a non-negligible number of cases for which the result sizes increase substantially. The same holds for the queries of the other logs (cf. Table 3).

After looking at all pairs of test queries as a whole, we have tried to isolate the queries for which there is a notable result size increase. It turns out that the differentiating characteristic is whether the queries contain sequences of OPTIONALs or not. As shown by the statistics in the second and the third block of Table 3, almost all of the queries for which we observe the more significant results size increases are queries with sequences of OPTIONALs.

6 Approaches to Implement OPT$^+$

We now turn to research question RQ2 which is concerned with the potential of reducing the response times when using OPT$^+$ instead of OPTIONAL. Since OPT$^+$ is a logical operator, the crux of the question is whether OPT$^+$ enables a query execution engine to employ a specific algorithm that implements OPT$^+$ and is designed to return as many solution mappings of query results as early as possible. If that is the case, an additional aspect of research question RQ2 is whether this algorithm allows the engine to return the first solutions for an OPT$^+$ query earlier than the same number of solutions that the engine would return for the corresponding OPTIONAL query (naturally, for the latter, the engine has to employ an algorithm that implements the OPTIONAL operator).

Our approach to answer RQ2 has been to use an existing SPARQL query execution engine and extend it with algorithms for the OPT$^+$ operator. We have selected the query engine of Apache Jena for this purpose. That is, we have developed an OPT$^+$-aware extension for Jena. This extension consists of two different algorithms to implement the OPT$^+$ operator natively in query execution plans (i.e., in contrast to implementing OPT$^+$ by using AND and UNION as done for the aforementioned OPT$^+$-like queries).

In this section we describe the algorithms, including the algorithm that Jena uses for OPTIONAL and that is the basis of two of the four OPT$^+$-specific algorithms. In the next section we compare these algorithms experimentally.

6.1 Execution of OPTIONAL queries in Apache Jena

Query execution in Jena is based on the well-known iterator model [5]. That is, every query execution plan is a sequence (or a tree) of iterators that are connected as a pipeline. Each iterator in the pipeline produces solution mappings by executing an algorithm that consumes input solution mappings from the predecessor iterator. The approach is synchronous; that is, the solution mappings are passed through the pipeline in a pull-based manner.

While the query engine of Jena contains various types of iterators for the different logical operators of SPARQL, we are concerned only with the iterator used for SPARQL expressions of the form $(P_1 \text{ OPT } P_2)$. The algorithm implemented by this iterator is a variation of a nested loops join (NLJ) where the outer loop consumes an input iterator I_{L} that is created to produce the result of P_1. Every solution mapping μ obtained from I_{L} is used to initialize an iterator I_{R}^{μ} for a version of P_2 in which variables have been replaced according to μ. Next, the algorithm uses I_{R}^{μ} as the inner loop; during this loop, every solution mapping μ' that can be consumed from I_{R}^{μ} is used to produce an output mapping $\mu \cup \mu'$. On the other hand, if I_{R}^{μ} does not return any solution mappings (i.e., μ does not have join partners in the result of P_2), then μ itself is an output mapping (as per the semantics of the OPT operator).

Example 2. Let us revisit the example query and example data in Figure 1. To execute the query, Jena would first create an iterator I_{L} that shall produce the result of the non-optional first triple pattern in the query. Hence, for the example data, this iterator would return the following two solution mappings:

$$\mu_1 = \{\, \texttt{?post} \to \texttt{ex:post1},\ \ \texttt{?text} \to \texttt{"Good ..."} \,\},$$
$$\mu_2 = \{\, \texttt{?post} \to \texttt{ex:post2},\ \ \texttt{?text} \to \texttt{"I can..."} \,\}.$$

Next, Jena would create an iterator I_{OPT} for the OPTIONAL operator in the query and connect it to I_{L} as its input iterator. Then, during query execution, I_{OPT} requests the mappings produced by I_{L} one after another. Suppose I_{L} returns μ_1 first. Based on μ_1, I_{OPT} substitutes the variable `?post` in the optional, second triple pattern of the query, which results in the more specific triple pattern $(\texttt{ex:post1}, \texttt{ex:hasImage}, \texttt{?img})$. Now, I_{OPT} initializes a new iterator $I_{\mathrm{R}}^{\mu_1}$ to obtain the result of this pattern. For the example data, $I_{\mathrm{R}}^{\mu_1}$ returns a single solution mapping, $\mu' = \{\, \texttt{?img} \to \texttt{ex:sun.png} \,\}$, with which I_{OPT} produces its first output mapping $\mu_3 = \mu_1 \cup \mu'$, i.e.,

$$\mu_3 = \{\, \texttt{?post} \to \texttt{ex:post1},\ \ \texttt{?text} \to \texttt{"Good ..."},\ \ \texttt{?img} \to \texttt{ex:sun.png} \,\}.$$

Since $I_{\mathrm{R}}^{\mu_1}$ does not return any more mappings, I_{OPT} closes $I_{\mathrm{R}}^{\mu_1}$ and requests the next mapping from I_{L}, which is μ_2. When processing this mapping, I_{OPT} uses another new iterator, $I_{\mathrm{R}}^{\mu_2}$, to obtain the result of the triple pattern $(\texttt{ex:post2}, \texttt{ex:hasImage}, \texttt{?img})$. However, this result is empty for the example data. Therefore, after unsuccessfully trying to obtain any solution mapping from $I_{\mathrm{R}}^{\mu_2}$, it becomes clear that μ_2 is another output mapping of I_{OPT}. Moreover, since I_{L} has also been exhausted at this point, μ_2 is the last output.

6.2 Algorithm NLJ$^+$

Our first algorithm for the OPT$^+$ operator, which we call NLJ$^+$, is a simple adaptation of the aforementioned NLJ-based algorithm for OPTIONAL. The only difference is the following: Whenever NLJ$^+$ obtains a solution mapping μ from the input iterator I_{L}, this mapping is returned as an output mapping immediately (which is a correct behavior for the OPT$^+$ operator). Only after this step does NLJ$^+$ initialize the iterator I_{R}^{μ} for the inner loop, which then proceeds as described above (cf. Section 6.1). That is, during this inner loop, μ is joined with all mappings returned by I_{R}^{μ}. Then, after exhausting I_{R}^{μ}, NLJ^{+} directly continues with the next solution mapping from I_{L} (i.e., independent of whether I_{R}^{μ} has returned solution mappings or not).

Example 3. Assume that the keyword OPTIONAL in the example query (Figure 1) was replaced by a new keyword that denotes an OPT^{+} operator. Then, for the evaluation of this operator we may use an iterator $I_{\mathrm{NLJ}^{+}}$ that implements the NLJ^{+} algorithm. The input to this iterator would be the same iterator I_{L} as used in the previous example (cf. Example 2). During the query execution, immediately after obtaining the solution mapping μ_1 from I_{L}, $I_{\mathrm{NLJ}^{+}}$ returns μ_1 as an output mapping. Thereafter, exactly as done by the iterator I_{OPT} in Example 2, $I_{\mathrm{NLJ}^{+}}$ creates iterator $I_{\mathrm{R}}^{\mu_1}$, consumes the mapping μ' returned by $I_{\mathrm{R}}^{\mu_1}$, and produces mapping $\mu_3 = \mu_1 \cup \mu'$ as the second output mapping. Next, $I_{\mathrm{NLJ}^{+}}$ closes $I_{\mathrm{R}}^{\mu_1}$, receives μ_2 from I_{L}, and directly passes μ_2 on as the next output mapping. Finally, $I_{\mathrm{NLJ}^{+}}$ creates iterator $I_{\mathrm{R}}^{\mu_2}$, closes it again after unsuccessfully trying to obtain any solution mapping from it, and indicates that there are no more output mappings. Hence, by using the algorithm NLJ^{+}, the three solution mappings that make up the complete query result are returned in the following order: μ_1, μ_3, μ_2.

6.3 Algorithm mNLJ^{+}

The second algorithm for OPT^{+}, which we call mNLJ^{+}, is a variation of NLJ^{+} in which the mappings from the input iterator I_{L} are materialized into a list. In addition to appending each such mapping to this list, the mapping is returned immediately as an output mapping. After I_{L} has been exhausted and, thus, all its mappings are in the list, the list is now used for the outer loop and the NLJ-style processing begins. Of course, in this case, solution mappings from the list are not returned again as output mappings (but as join partners to be merged with mappings from the inner-loop iterators).

Example 4. As in Example 3, assume that we aim to execute an OPT^{+} version of the example query in Figure 1. Now, however, we use an iterator $I_{\mathrm{mNLJ}^{+}}$ that implements the mNLJ^{+} algorithm. As before, the input to this iterator would be the iterator I_{L} of Examples 2 and 3. During the query execution, $I_{\mathrm{mNLJ}^{+}}$ obtains the solution mapping μ_1 from I_{L}, returns it as an output mapping, and adds it to the list of mappings maintained by $I_{\mathrm{mNLJ}^{+}}$. Next, $I_{\mathrm{mNLJ}^{+}}$ immediately obtains the next solution mapping from I_{L}, which is μ_2. As for μ_1, μ_2 is returned as an output mapping and added to the list of mappings. Now, I_{L} has been exhausted and, thus, $I_{\mathrm{mNLJ}^{+}}$ proceeds to the next phase in which each solution mapping μ_x that has been added to the internal list is used to find join partners by creating a corresponding iterator $I_{\mathrm{R}}^{\mu_x}$ as in the previous examples. Hence, the result of this phase is that the third output mapping μ_3 is produced and returned. Therefore, in comparison to the NLJ$^+$-based query execution in Example 3, the mNLJ$^+$-based query execution returns the three solution mappings in a different order: μ_1, μ_2, μ_3.

7 Evaluation

Given our OPT$^+$-aware extension for Apache Jena, we have conducted an experimental evaluation. In this evaluation we compare query executions using the resulting OPT$^+$-aware execution plans to executions of corresponding OPTIONAL queries as well as corresponding OPT$^+$-like queries (see above). The goal of this evaluation is to gain an understanding of the response times that can be achieved by such query executions. In this section we describe the setup of the experiments and the results.

7.1 Experimental Environment

For the experiments we have used HDT [4] as a back-end for storing RDF data. Data in HDT can be accessed in terms of triple patterns. Hence, using HDT as storage back-end bears similarities to accessing data from a remote server that provides a Triple Pattern Fragments interface [19]. The HDT java libraries come with a Jena connector that we have employed to use the query execution engine of Jena—with our OPT$^+$-aware extension—on top of an HDT-stored RDF dataset. We have integrated these components into a driver program that runs the experiments. This program executes a given workload of queries sequentially, one query at a time. For each query, the program records the relevant measurements (see below). Query executions that take longer than 10 seconds are stopped and recorded as timed out.

The experiments have been conducted on a computer that is equipped with an Intel Core i7-2620M CPU (2.7GHz) and 8 GB of main memory. This computer runs the Ubuntu 12.04.5 LTS operating system with Oracle Java 1.8.0_92. Our Jena-based experiment system is implemented using Jena 3.7.0 and the latest version of the HDT Java libraries from the HDT github repository.[5] 4 GB of main memory have been assigned to the Java process of the experiment system.

[5] https://github.com/rdfhdt/hdt-java (last commit from May 10, 2018)

7.2 Metrics

We have instrumented our experiment driver program to record a timestamp and the number of triples retrieved from the HDT back-end at any point at which a solution mapping is returned for the executed query. After a query execution finishes, this data is used to produce the following measurements:

RT$X\%$: response time until the first 10% of all solutions (RT10%), until the first 20% (RT20%), ..., until 100% of the solutions (RT100%);

Tr$X\%$: number of triples retrieved from the back-end to produce the first 10% of all solutions (Tr10%), the first 20% (Tr20%), ..., 100% (Tr100%);

RT1st$X$: response time to return the first 10 solutions (RT1st10), the first 20 solutions (RT1st20), ..., the first 100 solutions (RT1st100).

Additionally, we measure the overall query execution time (QET).

7.3 Dataset and Queries

We have selected (uniformly at random) a collection of 60K pairs of queries that we had created from the $\text{DBP}_{3.5.1}$ query log for the result-size analysis presented in Section 5. The reason for selecting the queries from $\text{DBP}_{3.5.1}$ is that the OPTIONAL queries in this log are comparably diverse in terms of how they use OPTIONAL (cf. Section 4) and in terms of result-size increase when replacing OPTIONAL by OPT^+ (cf. Table 3). As a consequence of this choice of queries, we have to use the DBpedia dataset, version 3.5.1. Hence, the HDT back-end for our experiments contains this dataset.

Recall that each pair of queries created for our result-size analysis consists of an OPTIONAL query and a corresponding OPT^+-like query; the latter is a representation of the OPT^+ version of the OPTIONAL query that has been obtained by first replacing every OPT operator by OPT^+ and then rewriting every OPT^+ operator using AND and UNION (cf. Proposition 1). Therefore, in our experiment we can also use these OPT^+-like queries to observe how their execution (using the standard Jena query iterators for AND and UNION) compares to the executions that use the OPT^+-specific algorithms for the OPT^+ version of the OPTIONAL queries in the query pairs.

In addition to the pairs of real-world queries from the $\text{DBP}_{3.5.1}$ log, we have created 4 more query pairs for which we handcrafted new OPTIONAL queries (and then generated the corresponding OPT^+-like queries). These queries are listed in the Appendix. Each of these handcrafted queries contains a single OPTIONAL and represents some form of an extreme case:

Table 4 Statistics about the query result sizes of the handcrafted queries.

Query	Q1	Q2	Q3	Q4
Result size OPT version	21,862	21,862	65,618	809,372
Result size OPT$^+$ version	21,862	43,724	87,480	810,486

Q1 The result of the optional part of this query is empty; thus, none of the solutions for the non-optional part has join partners in the optional part.

Q2 Every solution for the non-optional part of this query has exactly one join partner in the result of the optional part.

Q3 Every solution for the non-optional part of this query has at least one join partner in the result of the optional part; some have a few join partners.

Q4 The non-optional part of the query is more complex, and every solution for this part has several join partners in the result of the optional part.

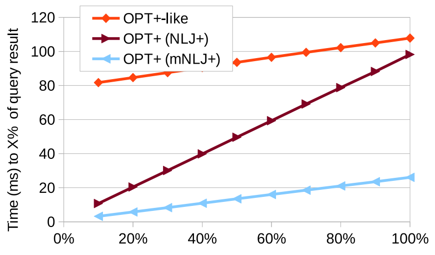
Table 4 presents statistics about the result sizes of the handcrafted queries.

7.4 Comparison of OPT$^+$ Approaches for the Handcrafted Queries

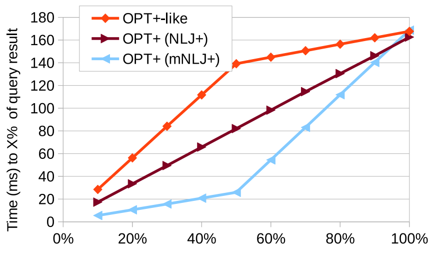
We begin our analysis by taking a detailed look at the response times measured for the executions of each of the handcrafted queries. The charts in Figure 4 illustrate the times required by the different OPT$^+$-specific approaches to produce $X\%$ of the solution mappings contained in the respective query results (i.e., RT$X\%$), and the charts in Figure 6 illustrate the number of triples that the approaches had to retrieve for producing these solution mappings (i.e., Tr$X\%$). Additionally, the charts in Figure 5 illustrate the times required to produce the first X of these solution mappings for Q1 and Q4 (i.e., RT1st$X$). The latter figure does not contain the corresponding charts for Q2 and Q3 because, for these queries, all RT1st$X$ measurements are below 1ms and do not show any significant differences between the approaches.

We first focus on Q1 for which we observe that the execution of the OPT$^+$-like version of this query has the greatest (i.e., worst) RT$X\%$ values, while the mNLJ$^+$-based execution of the OPT$^+$ version of Q1 achieves the best response times. To describe the behavior of the approaches in detail we recall that the OPT$^+$-like version of a single-OPT$^+$ expression $(P\,\text{OPT}^+P')$ is of the form $((P\,\text{AND}\,P')\,\text{UNION}\,P)$. Hence, the non-optional subexpression P of such queries is contained—and, thus, executed—twice in the OPT$^+$-like versions.

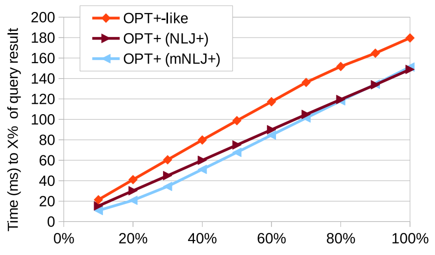
We first focus on the executions of the OPT$^+$-like queries: Except for query Q4, the RT$X\%$ values are the greatest (i.e., worst) for these executions (cf. Figure 4), and the same holds for the RT1st$X$ values in the case



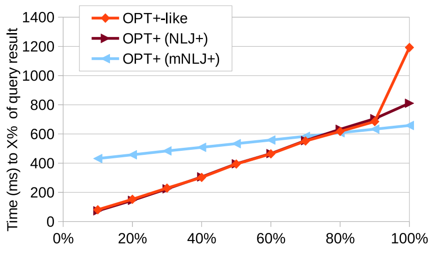
(a) Q1



(b) Q2



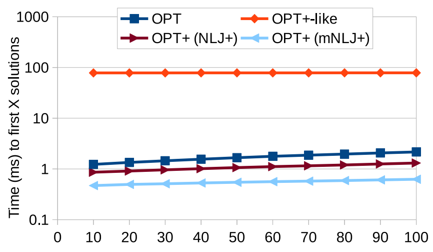
(c) Q3



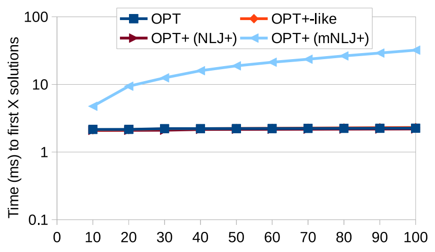
(d) Q4

Figure 4 Response times for the handcrafted queries in terms of time to $X\%$ of all solutions.

of Q1 (cf. Figure 5(a)). To explain this behavior we recall that the OPT$^+$-like version of an expression $(P\,\text{OPT}^+\,P')$ is of the form $((P\text{ AND }P')\text{ UNION }P)$. Hence, the non-optional subexpression P of such queries is contained—and, thus, executed—twice in the OPT$^+$-like versions: first in the $(P\text{ AND }P')$ part and second as the right argument of the UNION operator used in the OPT$^+$-like queries. This double effort does not only lead to typically higher query execution times (as shown in Figure 7) but it also means that the triples for

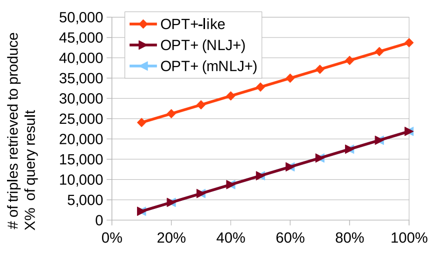


(a) Q1

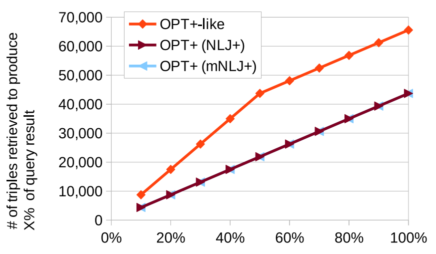


(b) Q4

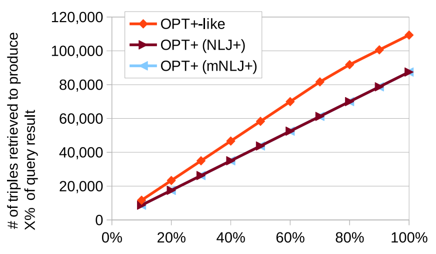
Figure 5 Response times to the first X solutions for the handcrafted queries (log scale).



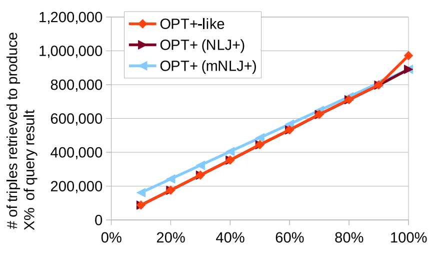
(a) Q1



(b) Q2



(c) Q3

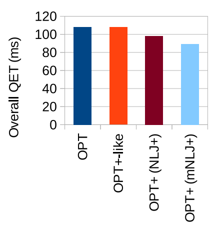


(d) Q4

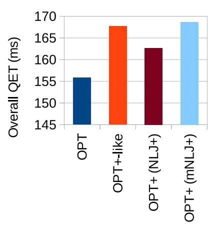
Figure 6 Number of triples retrieved for $X\%$ of all solutions for the handcrafted queries.

executing the non-optional subexpression P are retrieved twice from the storage back-end (as indicated by the comparably higher Tr$X\%$ values shown in Figure 6) and that the response times may be affected negatively.

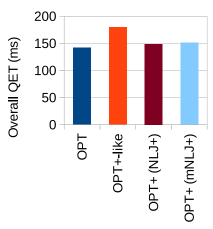
As an example of the latter, consider query Q1. For this query, the result of the optional part P' is empty. Thus, executing the $(P \text{ AND } P')$ part of the OPT$^+$-like version of this query does not produce any solution mappings; yet it requires time (namely, ca. 81ms as can be seen in Figures 4(a) and 5(a)).



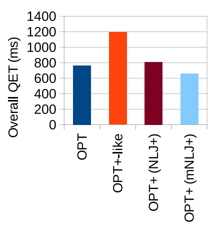
(a) Q1



(b) Q2



(c) Q3



(d) Q4

Figure 7 Overall query execution times for the handcrafted queries.

Only after this time, the query execution proceeds to the second argument of the UNION operator and starts producing the first output.

The execution of the OPT$^+$-like version of Q4 is also interesting. For this query, ca. 90% of the solution mappings of the query result are produced by executing the join between the optional and the non-optional part. For the remaining 10%, the non-optional part has to be executed again in the case of the OPT$^+$-like version of Q4, which explains the sudden rise at the end of the corresponding curves in Figures 4(d) and 6(d). In contrast, by using the dedicated OPT$^+$ algorithms (NLJ$^+$ and mNLJ$^+$) it becomes unnecessary to produce the solution mappings for the non-optional part twice.

When comparing the two native OPT$^+$ approaches, we observe that the mNLJ$^+$ executions achieve better response times for Q1-Q3, whereas, for Q4, the NLJ$^+$ execution is better. In some ways, the behavior of the mNLJ$^+$ algorithm is the opposite of how the OPT$^+$-like versions of the queries are executed; that is, mNLJ$^+$ first returns all solution mappings for the non-optional part before it tries to find corresponding join partners in the optional part (but without having to re-execute the non-optional part). This strategy is beneficial in cases in which most (or even all) of the solution mappings for the non-optional part have a few join partners only (like in Q2 and Q3) or no join partner at all (as in Q1). If, in contrast, most of the solution mappings for the non-optional part have many join partners, the strategy becomes less suitable. In such cases (with Q4 being one of them), the idea of NLJ$^+$ is more effective in terms of achieving small response times.

7.5 Comparison of OPT$^+$ versus OPT for the Handcrafted Queries

We now compare the executions of the OPT$^+$ versions versus the OPT versions of the handcrafted queries. Since the query results for the two versions may have a different size (cf. Section 5 and Table 4), comparing the query executions in terms of their response times to return a particular percentage of the respective query results is an apples-to-oranges comparison. Therefore, we focus on the response times to return a fixed number of solution mappings (i.e, RT1st$X$). Figure 5 illustrates the corresponding measurements.

For Q1, the executions of the OPT$^+$ version achieve slightly better response times than the execution of the OPT version. For the NLJ$^+$ execution this observation may be surprising, given that the NLJ$^+$ algorithm is very similar to the algorithm used for the OPT operator (cf. Sections 6.1 and 6.2). However, recall that the none of the solution mappings for the non-optional part of Q1 has a join partner in the optional part. Then, the NLJ$^+$ algorithm returns each of these solution mappings before the unsuccessful attempt to find join partners, whereas the algorithm for the OPT operator first tries to find join partners and, thereafter, checks whether it has found some or not. Although this check presents only a very small overhead, it adds up when iterating over the solution mappings obtained for the non-optional part.

For Q4, this small advantage of NLJ$^+$ over the OPT algorithm becomes insignificant for the following reason. Each solution mapping obtained for the non-optional part of Q4 does have join partners in the optional part. As a consequence, processing each of these solution mappings takes more time overall, which makes the overhead of the additional check negligible.

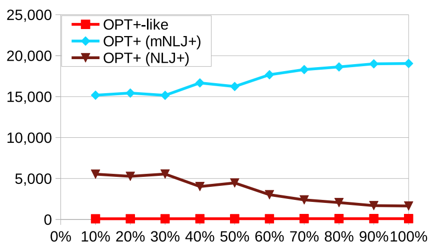
In comparison to the NLJ$^+$ executions, the mNLJ$^+$ executions achieve response times that are more different to the response times of the OPT executions, and these differences may be either positive (as in the case of Q1) or negative (Q4). The reasons for these differences are the same as the aforementioned reasons for the differences between mNLJ$^+$ and NLJ$^+$ (because of the similarity of the algorithm for the OPT operator and the NLJ$^+$ algorithm).

In summary, this experiment with the handcrafted queries shows that there are cases in which replacing the OPT operator by OPT$^+$ is beneficial in terms of response times (assuming a native implementation of OPT$^+$). This benefit may be more substantial if the mNLJ$^+$ algorithm is chosen to implement OPT$^+$ (instead of the NLJ$^+$ algorithm). However, there are also cases in which this choice can have the opposite of the desired effect (see Q4).

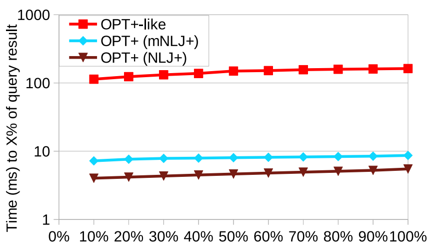
7.6 Comparison of OPT$^+$ Approaches for the Real-World Queries

While the handcrafted queries allow us to reason in detail about how the different approaches behave in specific cases, we now discuss our observations for the large workload of real-world queries obtained from the DBP$_{3.5.1}$ query log. We begin with a comparison of the OPT$^+$ approaches.

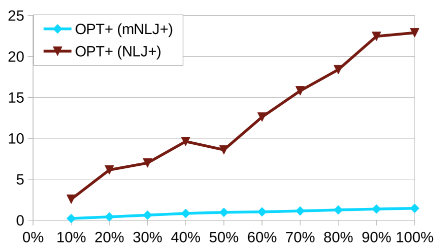
First, we notice that a number of query executions have hit our timeout threshold of 10 seconds (for details refer to the table in Figure 8(d)). Additionally, many of the queries whose executions did not time out have the empty query result. For these cases it is impossible to compare the approaches in terms of our response time metrics (RT1st$X$ and RT$X\%$). As a consequence, out of the 60K queries there are only 20,797 for which we have relevant measurements. For these queries, Figure 8(a) illustrates the number of cases in which each approach has achieved the smallest (i.e., best) RT$X\%$ value among the three approaches, and Figure 8(b) illustrates the average RT$X\%$ values that each of the three approaches has achieved for the 20,797 queries.



(a) # of cases of having the best RT$X\%$



(b) avg. RT$X\%$ for 20,797 queries (log scale)



(c) avg. differences between RT$X\%$ values

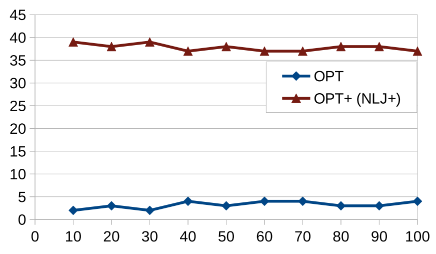
approach	# of timeouts
OPT$^+$-like	359
OPT$^+$ (NLJ$^+$)	167
OPT$^+$ (mNLJ$^+$)	166
OPT	144

(d) number of timeouts

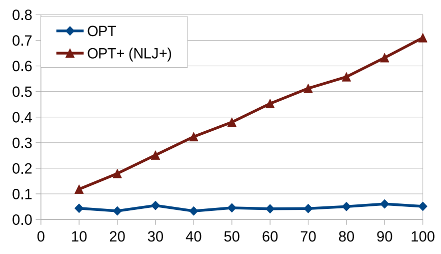
Figure 8 Comparison of the OPT$^+$ approaches for the real-world queries of the DBP$_{3.5.1}$ log.

We observe that the OPT$^+$-like approach cannot compete with the other two approaches. That is, its average response times are two orders of magnitudes greater (i.e., worse) and, for every $X \in \{10, 20, ..., 100\}$, there are less than 100 queries for which the approach has achieved the best RT1st$X$ value. The reasons for this behavior have already been mentioned above.

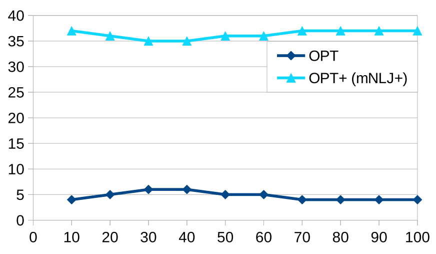
Regarding the NLJ$^+$ approach versus the mNLJ$^+$ approach, there is a significantly higher number of queries for which mNLJ$^+$ achieves better response times; on the other hand, however, NLJ$^+$ is slightly better in terms of average response times. To analyze these observations further we refer to Figure 8(c) which illustrates the average differences (in ms) between the RT$X\%$ values achieved by both approaches for the cases in which mNLJ$^+$ was better than NLJ$^+$ (light blue curve) and for the cases in which NLJ$^+$ was better than mNLJ$^+$ (brown curve). Our conclusion of this comparison is two-fold: First, while there are more cases in which mNLJ$^+$ achieves better response times than NLJ$^+$, the differences in these cases typically are not particularly significant. Second, there are a notable number of cases in which NLJ$^+$ achieves better response times than mNLJ$^+$, and in these cases the response times are significantly different.



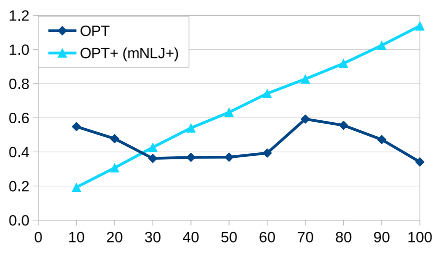
(a) # of cases of having the best RT1st$X$



(b) avg. differences between RT1st$X$ values



(c) # of cases of having the best RT1st$X$



(d) avg. differences between RT1st$X$ values

Figure 9 Comparison of OPT$^+$ versus OPT for the real-world queries of the DBP$_{3.5.1}$ log.

7.7 Comparison of OPT$^+$ versus OPT for the Real-World Queries

To compare the OPT-based executions versus OPT$^+$-based executions of the real-world query workload we, again, focus on the RT1st$X$ measurements only (as the RT$X\%$ measurements are not comparable due to the differences in the query result sizes; cf. Section 5). Then, from the 20,797 queries that have a nonempty result, there are only 41 for which the query result of the OPT version contains at least 100 solution mappings. Since this number of solution mappings is needed to obtain RT1st$X$ measurements with an X of up to 100, we use these 41 queries for the comparison of OPT$^+$ versus OPT.

We separately consider both of the two OPT$^+$-specific approaches for this comparison because none of them turned out to be a clear winner over the other one—neither for the handcrafted queries (cf. Section 7.4) nor for the real-world queries (cf. Section 7.6). For the NLJ$^+$ approach, Figure 9(a) illustrates the number of cases in which the RT1st$X$ values of the OPT$^+$-based executions are better than for the OPT-based executions and vice versa; Figure 9(c) provides the same type of chart for the mNLJ$^+$ approach.

For both, NLJ$^+$ and mNLJ$^+$, we make very similar observations. That is, there are significantly more cases in which the respective OPT$^+$-based executions of the 41 queries achieved better RT1st$X$ response times (cf. Figures 9(a) and 9(c)). However, the average differences between the response times in all of these cases are below 1ms (cf. Figures 9(b) and 9(d)).

8 Conclusions

In this paper we have analyzed a monotonic alternative to the OPTIONAL feature of SPARQL, which we have formalized as a new query operator called OPT$^+$. The main use case for this alternative are Web data integration components that execute queries over RDF-based data sources on the Web.

The trade-off of using the OPT$^+$ operator instead of OPTIONAL is that it may increase the size of query results. Regarding the question of how significant this increase can be in practice (i.e., research question RQ1), we conclude that:

C1.1 For a large fraction of the real-world queries we analyzed, there would be no result size increase at all if these queries were using OPT$^+$ instead of OPTIONAL.

C1.2 However, there is also a sizable fraction of queries for which the result size would increase.

C1.3 For many of these queries, the result sizes would increase no more than 2x, but there is also a non-negligible number of queries for which the result sizes would increase substantially. Almost exclusively, the latter are queries with sequences of OPTIONALs.

The motivation for introducing this alternative operator was to enable dedicated implementations that can reduce the response times of query executions in comparison to implementations of the OPTIONAL feature (i.e., returning a first fraction of the respective query results earlier). Regarding the question of how suitable the OPT$^+$ operator actually is for achieving this goal (research question RQ2), our main conclusion is that:

C2.1 Surprisingly, for the dedicated OPT$^+$ implementation approaches that we have considered we *cannot* confirm any significant advantages in terms of response times; although there are some differences, they are all below 1ms.

Other, more specific conclusions that we draw from our evaluation are:

C2.2 Implementing the idea of the OPT$^+$ operator by simply using the semantically equivalent expressions with AND and UNION is not an efficient approach; typically, it results in worse response times and also increases the amounts of data (triples) that have to be retrieved from the storage back-end (or from the remote server in case of a client-server architecture or in a decentralized query processing context).

C2.3 For each of the two OPT$^+$-specific algorithms, NLJ$^+$ and mNLJ$^+$, there exist cases in which it is better than the other in terms of achieving smaller response times. More specifically, our results show that mNLJ$^+$ achieves better response times than NLJ$^+$ in a greater number of cases, but the differences in these cases typically are not particularly significant. On the other hand, in the cases in which NLJ$^+$ achieves better response times than mNLJ$^+$, which are notably many, the response times are significantly different.

Acknowledgements This work has been funded by the CENIIT program at Linköping University (project no. 17.05).

Appendix

A1 Handcrafted Benchmark Queries

Q1
```
SELECT ?x ?o2 WHERE {
  ?x a <http://dbpedia.org/ontology/Band> .
  OPTIONAL { ?x <http://example.org/thisPropertyDoesNotExist> ?o2 }
}
```

Q2
```
SELECT ?x WHERE {
  ?x a <http://dbpedia.org/ontology/Band> .
  OPTIONAL { ?x a <http://dbpedia.org/ontology/Band> }
}
```

Q3
```
SELECT ?x ?o2 WHERE {
  ?x a <http://dbpedia.org/ontology/Band> .
  OPTIONAL { ?x a ?o2 }
}
```

Q4
```
SELECT ?x WHERE {
  ?x a <http://dbpedia.org/ontology/Band> ;
       <http://dbpedia.org/ontology/recordLabel> ?rl ;
       <http://www.w3.org/2000/01/rdf-schema#label> ?l .
  ?rl <http://www.w3.org/2000/01/rdf-schema#label> ?lrl
  FILTER ( strStarts(?lrl, "A") )
  OPTIONAL { ?b2 <http://dbpedia.org/ontology/recordLabel> ?rl }
}
```

References

[1] Maribel Acosta, Olaf Hartig, and Juan Sequeda. Federated RDF Query Processing. In Sherif Sakr and Albert Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2018.

[2] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 18–34, 2011.

[3] Renzo Angles and Claudio Gutiérrez. The expressive power of SPARQL. In *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings*, pages 114–129, 2008.

[4] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *J. Web Sem.*, 19:22–41, 2013.

[5] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[6] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, Online at http://www.w3.org/TR/sparql11-query/, March 2013.

[7] Olaf Hartig. *Querying a Web of Linked Data: Foundations and Query Execution*. PhD thesis, Humboldt-Universität zu Berlin, Germany, 2014.

[8] Olaf Hartig, Christian Bizer, and Johann Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, pages 293–309, 2009.

[9] Olaf Hartig and M. Tamer Özsu. Walking without a map: Ranking-based traversal for querying linked data. In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, pages 305–324, 2016.

[10] Lars Heling, Maribel Acosta, Maria Maleshkova, and York Sure-Vetter. Querying large knowledge graphs over triple pattern fragments: An empirical study. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, pages 86–102, 2018.

[11] Markus Luczak-Roesch, Saud Aljaloud, Bettina Berendt, and Laura Hollink. USEWOD 2016 Research Dataset. University of Southampton, 10.5258/SOTON/385344, 2016.

[12] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, pages 376–394, 2018.

[13] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics of SPARQL. Technical Report TR/DCC-2006-17, Department of Computer Science, University of Chile, 2006.

[14] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, 2009.

[15] Eric Prud'hommeaux and Carlos Buil-Aranda. SPARQL 1.1 Federated Query. W3C Recommendation, Online at http://www.w3.org/TR/sparql11-federated-query/, March 2013.

[16] Muhammad Saleem, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. LSQ: The Linked SPARQL Queries Dataset. In *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, pages 261–269, 2015.

[17] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems. *Semantic Web*, 7(5):493–518, 2016.

[18] Jürgen Umbrich, Katja Hose, Marcel Karnstedt, Andreas Harth, and Axel Polleres. Comparing Data Summaries for Processing Live Queries over Linked Data. *World Wide Web*, 14(5-6):495–544, 2011.

[19] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple Pattern Fragments: A Low-Cost Knowledge Graph Interface for the Web. *J. Web Sem.*, 37-38:184–206, 2016.