A Model of Distributed Query Computation in Client-Server Scenarios on the Semantic Web *

Olaf Hartig[1], **Ian Letter**[2], **Jorge Pérez**[3,4]
[1] Dept. of Computer and Information Science (IDA), Linköping University, Sweden
[2] Departamento de Ingeniería Matemática, Universidad de Chile
[3] Department of Computer Science, Universidad de Chile
[4] Millennium Institute for Foundational Research on Data, Chile
olaf.hartig@liu.se, iletter@dim.uchile.cl, jperez@dcc.uchile.cl

Abstract

This paper provides an overview of a model for capturing properties of client-server-based query computation setups. This model can be used to formally analyze different combinations of client and server capabilities, and compare them in terms of various fine-grain complexity measures. While the motivations and the focus of the presented work are related to querying the Semantic Web, the main concepts of the model are general enough to be applied in other contexts as well.

1 Introduction

Client-server-based query processing is concerned with an interaction between a client and a server with the goal of computing queries—posed on the client-side—over a server-side dataset. The queries that can be computed in such a setting depend on server-side and on client-side query processing capabilities, and so does the space of possible execution plans for such queries (where different plans may result in different query execution times and induce different resource requirements on the client, on the server, and on the network).

The trade-offs that arise from relying on different client/server capabilities have recently become a topic of research in the Semantic Web community. This development can be attributed to the realization that publicly accessible servers that provide query interfaces with a highly-expressive query language may easily become overloaded and, as a result, have availability issues [Buil-Aranda *et al.*, 2013].

To identify alternatives that are more reliable for building a Semantic Web in which data sources can be queried by client software, new types of query-based data access interfaces have become of interest [Verborgh *et al.*, 2016]. These interfaces are deliberately limited in the types of queries they support, which enforces that the computation of more expressive queries has to be performed partially on the client side.

While existing research in this context has focused on experimental results regarding the trade-offs imposed by these interfaces, we have recently embarked on a formal study to achieve a more fundamental understanding of possible interfaces and their trade-offs. To this end, in a recent paper [Hartig *et al.*, 2017] we have proposed an abstract machine model that can be used to capture formally the client-side capabilities and the server-side capabilities in a client-server system. Based on this model we have shown results that organize several combinations of such client and server capabilities into a lattice related to the classes of queries that can be computed by the corresponding types of client-server systems. Moreover, we have studied the interplay between several metrics such as the number of requests sent to the server, and the bandwidth of communication between client and server.

In this invited summary paper we provide an overview of this work. More specifically, we first provide some more background on client-server approaches to query processing in the Semantic Web context. Thereafter, we introduce our formal model and highlight the key results that we have shown based on the model. We emphasize that even if some notions of our model are defined to capture Semantic Web technologies, the model can easily be adapted to other technologies (e.g., relational databases).

2 Client-Server Approaches to Query Data on the Semantic Web

We begin this section by reminding the reader of the main concepts of the Resource Description Framework (RDF) [Cyganiak *et al.*, 2014] and its query language SPARQL [Harris and Seaborne, 2013], which are fundamental building blocks of all approaches to query the Semantic Web (including client-server approaches, which we describe below).

The RDF data model represents data in the form of *triples* of RDF terms where such *RDF terms* include Uniform Resource Identifiers (URIs) and literals (literals may appear only in the third position of RDF triples). Such an RDF triple may be understood as a binary predicate where the URI that denotes the predicate is in the second position of the triple. A set of RDF triples is called an *RDF graph* because it can be

*This paper is an abridged version of a paper titled "A Formal Framework for Comparing Linked Data Fragments" [Hartig *et al.*, 2017] which won the best research paper award at the 16th International Semantic Web Conference (ISWC 2017).

visualized as a graph (with directed, labeled edges).

The RDF query language SPARQL is based on the idea of graph pattern matching. Hence, the core component of each SPARQL query is a so-called query pattern. In the simplest case, such a query pattern is a *triple pattern*, that is, an RDF triple that may have variables in any of its three positions. Multiple triple patterns can be combined into a set, which is called a *basic graph pattern* (*BGP*) and resembles the notion of a conjunctive query. In addition to such BGPs, several other features are possible in SPARQL query patterns (e.g., UNION, FILTER). The subgraphs of a queried RDF graph that match such query patterns are used to produce the query results. Formally, the expected result of any given SPARQL query is defined based on an evaluation function $[\![\cdot]\!]_G$ that, given an RDF graph G, takes a SPARQL query pattern and returns a set (or multiset) of *solution mappings*, that is, partial functions that associate variables with RDF terms [Pérez *et al.*, 2009]. For instance, for a triple pattern tp, the result $[\![tp]\!]_G$ contains every solution mapping whose domain is the set of all variables in tp and replacing the variables in tp according to the mapping produces an RDF triple that is in G.

Given these fundamental concepts, we now describe client-server approaches that enable clients to query a server-side RDF graph using SPARQL queries. While all approaches rely on the HTTP protocol for the communication between the client and the server, the major technical difference is in the actual type of HTTP interface that the server exposes.

A prevalent (and standardized) type of such interfaces is the interface provided by so-called *SPARQL endpoints*; such endpoints are Web services that can be called to obtain the result of executing *any given SPARQL query* over the server-side RDF data [Feigenbaum *et al.*, 2013]. While this approach represents the most simple option from the client perspective (because the complete work of processing queries is performed solely by the server), a SPARQL endpoint server may easily become overloaded if many client applications start to access such a server concurrently. In fact, Buil-Aranda *et al.* [2013] have shown empirically that many public SPARQL endpoints on the Web have availability issues.

The other extreme are servers that simply offer a *dump* of their dataset for clients to download. In this case, the amount of query processing work on the server is reduced to a minimum. On the other hand, the drawbacks are that query processing on the client can start only after a potentially extensive load phase and the client-side copy of the data may have to be maintained and kept up-to-date. Moreover, the approach may cause a lot of unnecessary data transfer if only a fraction of the data is relevant to the queries processed by the client.

Based on the observation that SPARQL endpoints and data dumps are two opposite extremes in terms of how the query processing work is distributed between clients and servers, Verborgh *et al.* [2014; 2016] have challenged the Semantic Web community to study other options that may exist in between these extremes. To this end, Verborgh *et al.* have introduced the notion of *Linked Data Fragments*; this notion captures the types of query results that can be requested via a possible data access interface and that can be understood to represent "fragments" of the server-side dataset.

In addition to their conceptual framework, Verborgh *et al.* have also proposed a new type of interface called *Triple Pattern Fragments* (*TPF*) interface. Requests that clients can send to a TPF interface can be arbitrary triple patterns, and the response to such a request are all the triples in the server-side RDF graph that match the given triple pattern. Hence, in contrast to SPARQL endpoints, TPF-based servers are deliberately limited in the expressiveness of the requests they support. Then, executing an arbitrary SPARQL query using such a TPF server requires a client-side query engine that decomposes the query into a series of TPF requests and performs the other operators of the query based on the data retrieved by the requests. In comparison to querying a SPARQL endpoint, Verborgh *et al.* [2014; 2016] show experimentally that when using the TPF approach, the server load is reduced, the server response times become more stable, and so does the overall query throughput for increasing numbers of concurrent clients; on the other hand, SPARQL endpoints achieve a better performance in terms of query execution times (in particular, for a limited number of concurrent clients).

The proposal of TPF has sparked numerous follow-up research. For instance, aiming to improve the performance of TPF-based client-server systems, some authors have proposed other TPF-based query execution approaches to be used by the client-side query engine [Van Herwegen *et al.*, 2015a; Acosta and Vidal, 2015]. Other authors have started to propose other new types of interfaces. Most of these proposals extend TPF with additional features based on which specific operations of the query processing work can be pushed back to the server (e.g., [Van Herwegen *et al.*, 2015b; Vander Sande *et al.*, 2015]). A notable example of such an extension is the *Bindings-Restricted TPF* (*brTPF*) interface [Hartig and Buil-Aranda, 2016]. In addition to pure TPF requests, this interface allows clients to attach a set of SPARQL solution mappings to any TPF request. The response to such a brTPF request is expected to contain RDF triples from the underlying dataset that do not only match the given triple pattern (as in the case of TPF), but that are guaranteed to contribute in a join with the given solution mappings. By using this extension, the execution of joins can be distributed between client and server by using the well-known bind join strategy in which the client sends intermediate results to the server [Haas *et al.*, 1997]. By using this strategy, experiments have shown that a brTPF-based client-server system can achieve both a significant reduction of network load and a much higher query throughput when compared to a TPF-based setup [Hartig and Buil-Aranda, 2016].

In general, any possible data access interface—and its resulting client-server systems—have their own particular properties and trade-offs regarding performance, bandwidth needs, cache effectiveness, etc. The aforementioned research in this context has produced interesting practical results, but there have been only few formal tools to show fundamental results regarding such properties and trade-offs. The main goal of our recent work was to help fill this gap by developing solid theoretical foundations for studying and comparing Linked Data Fragment interfaces [Hartig *et al.*, 2017]. The remainder of this paper provides a summary of this work.

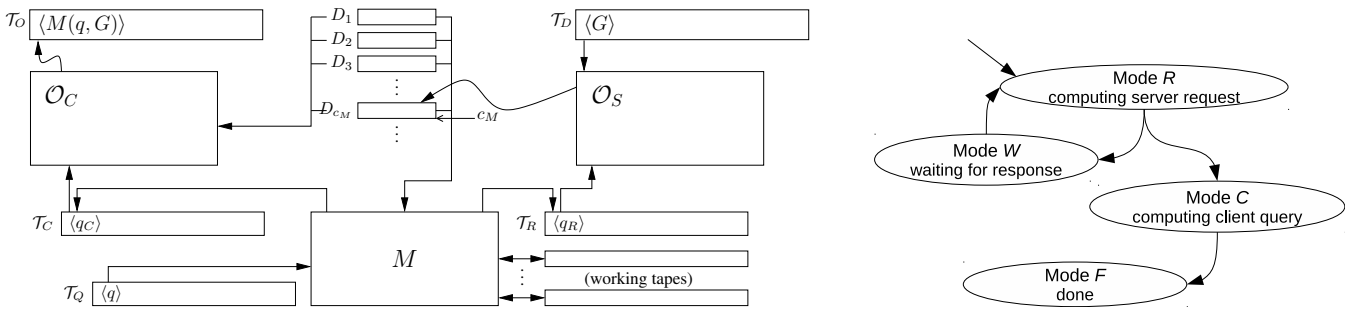

Figure 1: Illustration of an LDFM (left), and possible state transitions of an LDFM (right)

3 Linked Data Fragment Machine (LDFM)

This section introduces the abstract machine model—called *Linked Data Fragment Machine* (LDFM)—that we developed to capture possible client-server systems that execute user queries, issued at the client side, over a server-side dataset.

As a basis for defining LDFMs, we consider three types of languages: *query languages* to express the user queries, *server languages* to capture the types of requests that the client may send to the server, and *response-combination languages* to express how the responses from the server may be combined to produce the result of the given user query.

Then, an LDFM M is a multi-tape Turing Machine with several special features as illustrated in Figure 1 (left). That is, in addition to multiple ordinary working tapes, M has five special tapes: a *query tape* $\mathcal{T}_Q$, a *data tape* $\mathcal{T}_D$, a *server-request tape* $\mathcal{T}_R$, a *client tape* $\mathcal{T}_C$, and an *output tape* $\mathcal{T}_O$. Tapes $\mathcal{T}_Q$ and $\mathcal{T}_D$ are read-only, and $\mathcal{T}_R$, $\mathcal{T}_C$, and $\mathcal{T}_O$ are write-only. Additionally, M has an unbounded sequence $D_1, D_2, \ldots, D_k, \ldots$ of read-only tapes called *response containers*, and a counter c_M, called the *response counter*, that defines the last used response container. Finally, M has access to two oracle machines: a *server oracle* $\mathcal{O}_S$, which is associated with a server language $\mathcal{L}_S$, and a *client oracle* $\mathcal{O}_C$, associated with a response-combination language $\mathcal{L}_C$.

The input to an LDFM is a query, expressed in a query language and given in tape $\mathcal{T}_Q$, and the queried dataset, given in the read-only tape $\mathcal{T}_D$. All other tapes as well as the response containers are initially empty, and the counter c_M is 0.

Computations of LDFMs are based on four modes: *computing the next server request* (R), *waiting for response* (W), *computing client query* (C), and *done* (F). In all these modes, an LDFM M works as a standard Turing Machine that can use its ordinary working tapes arbitrarily (read/write). However, special tape $\mathcal{T}_R$ can be used only when the machine is in mode R, and tape $\mathcal{T}_C$ can be used only in mode C. Moreover, tapes $\mathcal{T}_D$ and $\mathcal{T}_O$ can be accessed only by the oracles $\mathcal{O}_S$ and $\mathcal{O}_C$, respectively (hence, M cannot access the queried dataset directly). Regarding the response containers, M is only able to read from them, and only oracle $\mathcal{O}_S$ is able to write in them.

Any computation of an LDFM M starts in mode R and progresses as follows (see Figure 1, right). In mode R, the machine can construct a server request $q_R \in \mathcal{L}_S$ and write it in tape $\mathcal{T}_R$. When done, the machine may change to mode W, which is a call to oracle $\mathcal{O}_S$. The oracle increments the counter c_M, writes into the container D_{c_M} the result of evaluating q_R over the dataset in $\mathcal{T}_D$, and deletes the content of tape $\mathcal{T}_R$. Next, M changes back to mode R, and the process may be repeated as described. Alternatively, at any point when in mode R, the machine may decide to change to mode C. In this mode, M constructs a response-combination query $q_C \in \mathcal{L}_C$, writes it in tape $\mathcal{T}_C$, and calls oracle $\mathcal{O}_C$ by changing to mode F. Then, oracle $\mathcal{O}_C$ evaluates q_C over data $D_1, \ldots, D_{c_M}$, and writes the result of this evaluation in tape $\mathcal{T}_O$. At this point the computation terminates. For concrete examples of LDFM-based query computations refer to the original paper [Hartig *et al.*, 2017].

We emphasize that this machine model clearly separates three of the main tasks done during client-server-based query processing: (1) the computation that plans and drives the overall query execution process by making requests to the server, (2) the computation that the server needs to do in order to answer requests issued by the client, and (3) the computation that the client needs to do to create the final output from the server responses. In our model the latter two tasks are separated into the oracles $\mathcal{O}_S$ and $\mathcal{O}_C$, respectively, and the languages $\mathcal{L}_S$ and $\mathcal{L}_C$ capture the related capabilities. An LDFM that is defined to use a particular pair of these languages, $\mathcal{L}_S$ and $\mathcal{L}_C$, is called an *($\mathcal{L}_C$,$\mathcal{L}_S$)-LDFM*.

Note that the definition of LDFMs is independent of specific technologies! Hereafter, to focus on Semantic Web technologies, we assume that the server-side dataset (given in tape $\mathcal{T}_D$) is an RDF graph, and for any query language $\mathcal{L}_Q$ (e.g., SPARQL) and any server language $\mathcal{L}_S$, the result of every query in $\mathcal{L}_Q$ or in $\mathcal{L}_S$ is a set of SPARQL solution mappings. Due to this assumption, any response-combination language $\mathcal{L}_C$ is an algebra over such sets of solution mappings.

4 Expressiveness Lattice

This section provides an overview of our results that show the relationships between different pairs of client and server capabilities in terms of expressiveness. We begin by defining the notions of computability and expressiveness of LDFMs.

Definition 1. *Let q be a user query, $\mathcal{L}_C$ be a response-combination language, and $\mathcal{L}_S$ be a server language. We say that q* is computable under $(\mathcal{L}_C, \mathcal{L}_S)$ *if there exists an $(\mathcal{L}_C, \mathcal{L}_S)$-LDFM M such that for every RDF graph G, the computation of M, with q in tape $\mathcal{T}_Q$ and G in $\mathcal{T}_D$, terminates with $[\![q]\!]_G$ in the output tape $\mathcal{T}_O$.*

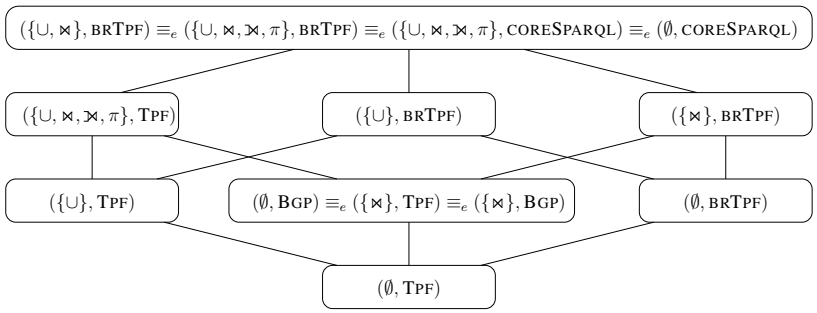

Figure 2: Expressiveness lattice for LDFMs

Definition 2. *Let $\mathcal{L}_1$ and $\mathcal{L}'_1$ be response-combination languages, and $\mathcal{L}_2$ and $\mathcal{L}'_2$ be server languages. We say that $(\mathcal{L}'_1, \mathcal{L}'_2)$ is* at least as expressive as *$(\mathcal{L}_1, \mathcal{L}_2)$, denoted by $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$, if every query that is computable under $(\mathcal{L}_1, \mathcal{L}_2)$ is also computable under $(\mathcal{L}'_1, \mathcal{L}'_2)$.*

We write $(\mathcal{L}_1, \mathcal{L}_2) \prec_e (\mathcal{L}'_1, \mathcal{L}'_2)$ if $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2) \not\preceq_e (\mathcal{L}_1, \mathcal{L}_2)$, and $(\mathcal{L}_1, \mathcal{L}_2) \equiv_e (\mathcal{L}'_1, \mathcal{L}'_2)$ denotes that $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2) \preceq_e (\mathcal{L}_1, \mathcal{L}_2)$.

Given these notions, we have shown results that allow us to establish the expressiveness-related lattice illustrated in Figure 2. For the server languages and the response-combination languages described in the following, this lattice provides a full picture of how most combinations of these languages compare in terms of expressiveness (the higher up in the lattice a language combination is, the more expressive it is).

The response-combination languages we have focused on are almost all the algebras (over sets of solution mappings) that can be constructed by using some of the following operators: join ($\bowtie$), union ($\cup$), left join ($⟕$), and projection (π) [Pérez *et al.*, 2009]. Note that the empty operator set ($\emptyset$) also denotes a response-combination language. This language can be used only to simply select the set of solution mappings from one of the response containers.

The server languages considered are the following:

- CORESPARQL is the core fragment of SPARQL with triple patterns, AND, OPT, UNION, FILTER, and SELECT.
- BGP is the basic graph pattern fragment of SPARQL.
- TPF is the language composed of queries that are a single triple pattern. Hence, this captures TPF servers.
- BRTPF is the language composed of queries of the form (tp, Ω), where tp is a triple pattern and Ω is a set of solution mappings; additionally, a triple pattern is also a BRTPF query. This captures the brTPF interface.

We emphasize that some of the equivalences and separations in the expressiveness lattice (Figure 2) do not necessarily follow from standard expressiveness results in the query language literature. In particular, the lattice highlights the high expressive power of using the brTPF interface.

5 Additional Complexity Measures

While the expressiveness results are a necessary starting point to formally compare different combinations of client/server capabilities, from a practical point of view one would also want to compare the resources that have to be payed when using one data access interface or another. More specifically, if two combinations of client/server capabilities are equally expressive, a question is: are we paying an additional cost when using one or the other? Or more interestingly, is any of the two strictly better than the other in terms of some of the resources needed to answer queries? In the remainder of this paper, we highlight that our proposed framework is suitable to also analyze such aspects of client-server systems.

First, another classical complexity measure is the (computational) complexity of query evaluation. In the original paper [Hartig *et al.*, 2017] we have adopted this measure (and corresponding results [Schmidt *et al.*, 2010]) to derive two more lattices that provide a comparison of LDFM settings in terms of the complexity of the query evaluation problem for the server and response-combination languages, respectively.

Perhaps more interestingly, we also have proposed new complexity measures; that is, we have defined *request complexity* and *transfer complexity* of queries under a given pair $(\mathcal{L}_C, \mathcal{L}_S)$ based on the idea that the *number of requests* of an LDFM computation is the final value of counter c_M and the *amount of data transferred* is the value $|D_1| + |D_2| + \cdots + |D_{c_M}|$. Based on these complexity measures, we define the following notions to compare combinations of client/server capabilities w.r.t. the resources needed to compute queries.

Definition 3. *Let $\mathcal{L}_1, \mathcal{L}'_1$ be response-combination languages and $\mathcal{L}_2, \mathcal{L}'_2$ be server languages. Then, we say that $(\mathcal{L}_1, \mathcal{L}_2)$* is at most as request demanding as *$(\mathcal{L}'_1, \mathcal{L}'_2)$, denoted by $(\mathcal{L}_1, \mathcal{L}_2) \preceq_r (\mathcal{L}'_1, \mathcal{L}'_2)$, if the following condition holds: For every function f and every query q that is computable under both $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$, if q has request complexity at most f under $(\mathcal{L}'_1, \mathcal{L}'_2)$, then q has request complexity at most f under $(\mathcal{L}_1, \mathcal{L}_2)$. We similarly define the notion of being* at most as data-transfer demanding *and denote it using $\preceq_t$.*

For $c \in \{r, t\}$ we have defined $\prec_c$ and $\equiv_c$ from $\preceq_c$ as usual. Then, for $(\{\bowtie\}, \text{TPF})$, $(\{\bowtie\}, \text{BGP})$, and $(\emptyset, \text{BGP})$ (which all have the same expressive power; cf. Figure 2) we have shown that $(\{\bowtie\}, \text{BGP}) \equiv_r (\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{TPF})$ and $(\{\bowtie\}, \text{BGP}) \prec_t (\{\bowtie\}, \text{TPF})$. Moreover, $(\{\bowtie\}, \text{TPF})$ and $(\emptyset, \text{BGP})$ are not comparable in terms of $\preceq_t$.

As another result, for $c \in \{r, t\}$ we also have shown that $(\{\cup, \bowtie, ⟕, \pi\}, \text{BRTPF}) \prec_c (\{\cup, \bowtie\}, \text{BRTPF})$. This result indicates that even though $(\{\cup, \bowtie\}, \text{BRTPF})$ is very expressive

(cf. Figure 2), one may need to pay an extra overhead in terms of both the transfer and the request complexity compared to a setting with a richer response-combination language.

These results illustrate the usefulness of our framework to formally compare different combinations of client and server capabilities in terms of fine-grain complexity measures. As future work, we plan to cover more such combinations, as well as additional complexity measures.

Acknowledgments

Hartig's work has been funded by the CENIIT program at Linköping University (prj. no. 17.05). Pérez is funded by the Millennium Institute for Foundational Research on Data.

References

[Acosta and Vidal, 2015] Maribel Acosta and Maria-Esther Vidal. Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data. In *Proceedings of the 14th International Semantic Web Conference (ISWC)*, 2015.

[Buil-Aranda *et al.*, 2013] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *Proceedings of the 12th International Semantic Web Conference (ISWC)*, 2013.

[Cyganiak *et al.*, 2014] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll, and Brian McBride. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014.

[Feigenbaum *et al.*, 2013] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. SPARQL 1.1 protocol. W3C Recommendation, March 2013.

[Haas *et al.*, 1997] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, 1997.

[Harris and Seaborne, 2013] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, March 2013.

[Hartig and Buil-Aranda, 2016] Olaf Hartig and Carlos Buil-Aranda. Bindings-Restricted Triple Pattern Fragments. In *Proceedings of the 15th International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE)*, 2016.

[Hartig *et al.*, 2017] Olaf Hartig, Ian Letter, and Jorge Pérez. A Formal Framework for Comparing Linked Data Fragments. In *Proceedings of the 16th International Semantic Web Conference (ISWC)*, 2017.

[Pérez *et al.*, 2009] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3), 2009.

[Schmidt *et al.*, 2010] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT)*, 2010.

[Van Herwegen *et al.*, 2015a] Joachim Van Herwegen, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. Query Execution Optimization for Clients of Triple Pattern Fragments. In *Proceedings of the 12th European Semantic Web Conference (ESWC)*, 2015.

[Van Herwegen *et al.*, 2015b] Joachim Van Herwegen, Laurens De Vocht, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. Substring Filtering for Low-Cost Linked Data Interfaces. In *Proceedings of the 14th International Semantic Web Conference (ISWC)*, 2015.

[Vander Sande *et al.*, 2015] Miel Vander Sande, Ruben Verborgh, Joachim Van Herwegen, Erik Mannens, and Rik Van de Walle. Opportunistic Linked Data Querying Through Approximate Membership Metadata. In *Proceedings of the 14th International Semantic Web Conference (ISWC)*, 2015.

[Verborgh *et al.*, 2014] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. Querying Datasets on the Web with High Availability. In *Proceedings of the 13th International Semantic Web Conference (ISWC)*, 2014.

[Verborgh *et al.*, 2016] 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. *Journal of Web Semantics*, 37-38:184–206, 2016.