A Formal Framework for Comparing Linked Data Fragments

Olaf Hartig[1(✉)], Ian Letter[2], and Jorge Pérez[3(✉)]

[1] Department of Computer and Information Science (IDA), Linköping University, Linköping, Sweden
olaf.hartig@liu.se

[2] Departamento de Ingeniería Matemática, Universidad de Chile, Santiago, Chile
iletter@dim.uchile.cl

[3] Department of Computer Science, Universidad de Chile, Santiago, Chile
jperez@dcc.uchile.cl

Abstract. The Linked Data Fragment (LDF) framework has been proposed as a uniform view to explore the trade-offs of consuming Linked Data when servers provide (possibly many) different interfaces to access their data. Every such interface has its own particular properties regarding performance, bandwidth needs, caching, etc. Several practical challenges arise. For example, before exposing a new type of LDFs in some server, can we formally say something about how this new LDF interface compares to other interfaces previously implemented in the same server? From the client side, given a client with some restricted capabilities in terms of time constraints, network connection, or computational power, which is the best type of LDFs to complete a given task? Today there are only a few formal theoretical tools to help answer these and other practical questions, and researchers have embarked in solving them mainly by experimentation.

In this paper we propose the *Linked Data Fragment Machine* (LDFM) which is the first formalization to model LDF scenarios. LDFMs work as classical Turing Machines with extra features that model the server and client capabilities. By proving formal results based on LDFMs, we draw a fairly complete *expressiveness lattice* that shows the interplay between several combinations of client and server capabilities. We also show the usefulness of our model to formally analyze the fine-grain interplay between several metrics such as the number of requests sent to the server, and the bandwidth of communication between client and server.

1 Introduction

The idea behind Linked Data Fragments (LDFs) is that different Semantic Web servers may provide (possibly many) different interfaces to access their datasets allowing clients to decide which interface better satisfies a particular need. Every such interface provides a particular type of so-called "fragments" of the underlying dataset [13]. Moreover, every interface has its own particular properties regarding performance, bandwidth needs, cache effectiveness, etc. Clients can

C. d'Amato et al. (Eds.): ISWC 2017, Part I, LNCS 10587, pp. 364–382, 2017.
DOI: 10.1007/978-3-319-68288-4_22

analyze the trade-offs when using one of these interfaces (or a combination of them) for completing a specific task. There are a myriad of possible interfaces in between SPARQL endpoints and RDF data dumps. Some interfaces that have already been proposed in the literature include Linked Data Documents [4,5], Triple Pattern Fragments (TPF) [13], and Bindings-Restricted Triple Pattern Fragments (brTPF) [7]. Different options for LDF interfaces are shown in Fig. 1.

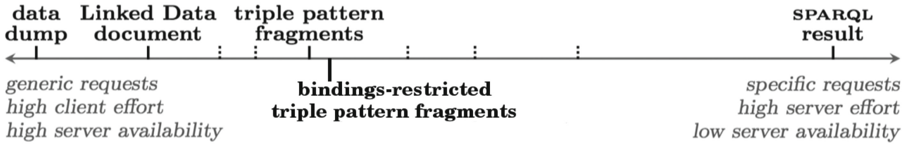


Fig. 1. Unidimensional view of Linked Data Fragments (figure taken from [7,13])

LDFs have already had a considerable practical impact. For instance, since the proposal of the TPF interface, the LOD Laundromat Website has published more than 650,000 datasets on the Web with this interface [3]. Moreover, DBpedia has also published a TPF interface which had an uptime of 99.99% during its first nine months [13]. Up to now, the research and development of LDFs has produced interesting practical results, but the studied interfaces are definitely not the final answer to querying semantic data on the Web, and one may expect that many new interfaces with different trade-offs can be made available by Semantic Web data servers in the near future.

Several practical challenges arise. On the server side, developers need to construct LDF interfaces that ensure a good cost/performance trade-off. Before implementing a new interface in some server, can we formally say something about the comparison of this new type of LDFs with earlier-proposed types? If the new interface is somehow subsumed in capabilities and cost by previously implemented interfaces (or by a simple combination of them), then there might be no reason to implement it. Answering this question requires an answer to the more general question on how to formally compare the properties of two different LDF interfaces given only their specifications.

On the client side, developers need to efficiently use and perhaps combine LDF interfaces. Thus, an interesting problem is the following: given a client with some restricted capabilities (in terms of time constraints, small budget, little computational power, restricted local expressiveness, etc.) and a task to be completed, which is the best interface that can be used to complete the task? Or even more drastically, can the task be completed at all given the restrictions on the client and a set of LDF interfaces to choose from? Today there are only a few formal tools to help answer the previously described questions, and researchers have embarked in solving them mainly by experimentation. The main goal of this paper is to help fill this gap by developing solid theoretical foundations for studying and comparing LDF interfaces.

It is not difficult to see that one can compare LDF interfaces in different ways. For instance, in Fig. 1 (taken from [7,13]) three criteria are considered: (1) general vs. specific requests, (2) client vs. server effort, and (3) high vs. low availability. We note however that this figure is not meant to provide an accurate account of the trade-offs of the included interfaces but to highlight the existence of such trade-offs. To this end, the figure has been kept deliberately simplistic by organizing the criteria and the interfaces along a single axis. While serving the intended purpose, this deliberate simplification has the disadvantage of suggesting that the given three criteria are correlated and, for example, the Linked Data Documents interface is always in between data dumps and SPARQL endpoints. A counterexample to the latter can be shown if we consider expressiveness as another criterion; more specifically, lets consider the type of queries that can be answered if we allow the client to use full computational power (Turing complete) to process data after making as many requests to the server as it needs. Assume that we have a server that provides data dumps, Linked Data Documents, and a SPARQL endpoint. Then, one can formally prove that the client is strictly less expressive when accessing the Linked Data Documents instead of the data dump or the SPARQL endpoint. To see this, consider a query of the following form:

$Q1$: "Give me all the subjects and objects of RDF triples whose predicate is rdf:type."

This query cannot be answered completely over a dataset by using the Linked Data Document interface no matter how many requests the client sends to the server [6]. On the other hand, it is not difficult to show that both, data dumps and SPARQL endpoints, can answer the query completely. Thus, when considering the expressiveness dimension, Linked Data Documents are not longer in between data dumps and SPARQL endpoints.

Consider another scenario in which one wants to measure only the number of requests that the client sends to the server in order to answer a specific query. Lets assume this time that the server provides a data dump, a SPARQL endpoint, and a TPF interface, and consider the following query.

$Q2$: "Give me all the persons reachable from Peter by following two foaf:knows links."

It is straightforward to see that a client using either the data dump or the SPARQL endpoint can answer this query by using a single request to the server, while a TPF client needs at least two requests. Thus, in this case, data dumps are more efficient than TPFs in terms of number of server requests. On the other hand it is clear that in terms of the amount of data transferred, TPFs are more desirable for $Q2$ than data dumps.

Although the two examples described above are very simple, they already show that the comparison of LDF interfaces is not always one-dimensional. Moreover, the comparison can quickly become more complex as we want to analyze and compare more involved scenarios. For instance, in both cases above we just analyzed a single query. In general, one would like to compare LDF interfaces in terms of classes of queries. Another interesting dimension is client-side computational power. In both cases above we assumed that the client is Turing complete, and thus the client is able to apply any computable function to the fragments obtained from an LDF interface. However, one would like to consider also clients with restricted capabilities (e.g., in terms of computational power or storage). Moreover, other dimensions such as bandwidth from client to server, bandwidth from server to client, time complexity on the server, cacheability of results, and so on, can substantially add difficulty to the formal analysis. In this paper we embark on the formal study of Linked Data Fragments by proposing a framework in which several of the aforementioned issues can be formally analyzed.

Main contributions and organization of the paper: As our main conceptual contribution we propose the *Linked Data Fragment Machine* (LDFM). LDFMs work as classical Turing Machines with some extra features that model the server and client capabilities in an LDF scenario. Our machine model is designed to clearly separate three of the main tasks done when accessing a Linked Data Fragment server: (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. These design decisions allow us to have a model that is powerful enough to capture several different scenarios while simple enough to allow us to formally prove properties about it. The LDFM model is presented in Sect. 2.

As one of our main technical contributions, we use our machine to formalize the notion of *expressiveness* of an LDF scenario and we draw a fairly complete *lattice* that shows the interplay between several combinations of client and server capabilities. While expressiveness is studied in Sect. 3, in Sect. 4 we analyze LDF scenarios in terms of classical computational complexity. Moreover, our machine model also allows us to formally analyze LDFs in terms of two additional important metrics, namely, the number of requests sent to the server, and the bandwidth of communication between the server and the client. Both notions are formalized as specific computational-complexity measures over LDFMs. We present formal results comparing different scenarios and demonstrate the suitability of our proposed framework to also analyze the fine-grain interplay between complexity metrics. These results are presented in Sect. 5.

For the sake of space most of the details on the proofs have been omitted but can be found in the appendix at http://dcc.uchile.cl/~jperez/ldfm-ext.pdf.

2 Linked Data Fragment Machine

This section introduces our abstract machine model that captures possible client-server systems that execute user queries, issued at the client side, over a server-side dataset.

Informally, the machine in our model captures the whole of a client-server system (i.e., both, the server and the client). However, the program of the machine can be considered to be executed on the client side. To communicate with the server the machine uses a *server language*, $\mathcal{L}_S$, which essentially represents the type of requests that the server interface is able to answer. Additionally, the machine is also in charge of producing the result of the given user query by combining the responses from the server. The corresponding result-construction capability is captured by a *response-combination language*, $\mathcal{L}_C$, which is an algebra over the server responses. To answer a user query the machine performs the following general process: The machine begins by creating requests for the server in the form of $\mathcal{L}_S$ queries. After issuing such a request, the corresponding response becomes available in an internal *result container*. Then, the machine can decide to continue with this process by issuing another request. Every response from the server is stored in a different result container, and moreover, a result container cannot be modified after it is filled with a server response (i.e., it can only be read by the machine). In the final step, the machine uses the response-combination language $\mathcal{L}_C$ to create a query over the result containers. The execution of this $\mathcal{L}_C$-query produces the final output of the process (that is, the result of the user query). In the following, we define the machine formally. We first formally capture the different types of query languages involved and next we provide the formal definition of the machine; thereafter, we describe the rationale of the different parts of the machine and we introduce notions of computability and expressiveness based on the machine.

2.1 Preliminaries

Our model assumes the following three types of queries.

User queries are queries that are issued at the client side and that the client-server system (captured by our machine) executes over the server-side dataset. We assume that this dataset is represented as an RDF graph without blank nodes. Then, a possible class of user queries could be SPARQL queries. However, to make our model more general we allow user queries to be expressed also in other query languages. To this end, for our model we introduce the abstract notion of an *RDF query*. Formally, an RDF query is an expression q for which there exists an evaluation function that is defined for every RDF graph G and that returns a set of SPARQL solution mappings, denoted by $[\![q]\!]_G$.

Requests are queries that the client sends to the server during the execution of a user query. The form of these requests depends on the type of interface provided by the server. We capture such interface types (and, thus, the possible requests) by introducing the notion of a *server language*; that is, a language $\mathcal{L}_S$ that is associated with an evaluation function that, for every query $q_R \in \mathcal{L}_S$ and every RDF graph G, returns a set of SPARQL solution mappings, which we denote by $[\![q_R]\!]_G$. Examples of server languages considered in this paper are given as follows:

– coreSparql is the core fragment of SPARQL that considers triple patterns, And, Opt, Union, Filter, and Select. Due to space limitations, we refer to [2,10] for a formal definition of this fragment and its evaluation function.

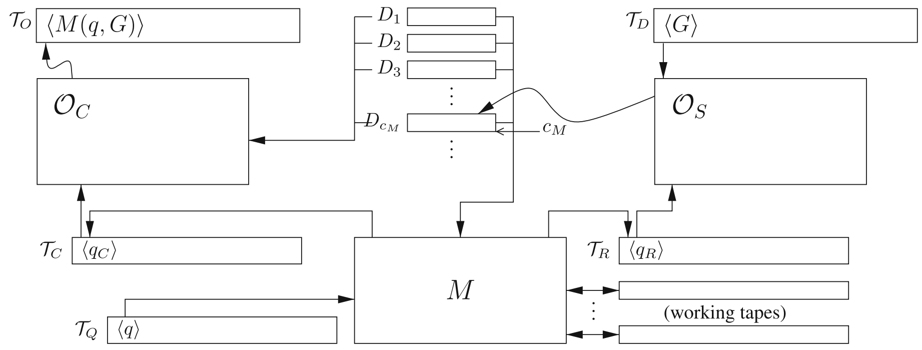
- Bgp is the basic graph pattern fragment of SPARQL (i.e., triple patterns and And).
- Tpf is the language composed of queries that are a single triple pattern. Hence, this language captures servers that support the triple pattern fragments interface [13].
- Tpf+Filter is the language composed of queries of the form (tp Filter θ) where tp is a triple pattern and θ is a SPARQL built-in condition as defined in [10].
- 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. This language captures the bindings-restricted triple pattern interface [7]. The evaluation function is defined such that for every RDF graph G it holds that $[\![(tp, \Omega)]\!]_G = \pi_{\text{vars}(tp)}([\![tp]\!]_G \bowtie \Omega)$ where π is the projection operator [11], vars(tp) is the set of variables in tp, $[\![tp]\!]_G$ is the evaluation of tp over G [10], and $\bowtie$ is the join operator [10]. For simplicity we assume that a triple pattern tp is also a brTpf query in which case the evaluation function is simply $[\![tp]\!]_G$.
- Dump is the language that has a single expression only, namely the triple pattern $(?s, ?p, ?o)$ where $?s$, $?p$, and $?o$ are different variables. This language captures interfaces for downloading the complete server-side dataset.

For any two server languages $\mathcal{L}_S$ and $\mathcal{L}'_S$ we write $\mathcal{L}_S \subseteq \mathcal{L}'_S$ if every query in $\mathcal{L}_S$ is also in $\mathcal{L}'_S$. For instance, Dump $\subseteq$ Tpf $\subseteq$ Bgp $\subseteq$ coreSparql.

Response-combination queries are queries that describe how the result of a user query can be produced from the server responses. Since each server response in our model is a set of solution mappings, and so is the result of any user query, we assume that response-combination queries can be expressed using languages that resemble an algebra over sets of solution mappings. We call such a language a *response-combination language*. In this paper we denote such response-combination languages by the set of algebra operators that they implement. For instance, the response-combination language denoted by the set $\{\bowtie, \pi\}$ can be used to combine multiple sets $\Omega_1, \ldots, \Omega_n$ of solution mappings by applying the aforementioned join and projection operators in an arbitrary manner. Other algebra operators that we consider in this paper are the union and the left outer join [2], denoted by $\cup$ and $⟕$, respectively. Note that based on our notation, the empty operator set ($\emptyset$) also denotes a response-combination language. This language can be used only to simply select one Ω_i out of multiple given sets $\Omega_1, \ldots, \Omega_n$ of solution mappings (i.e., without being able to modify Ω_i).

2.2 Formalization

A *Linked Data Fragment Machine* (LDFM) M is a multi-tape Turing Machine with the following special features. In addition to several 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 tapes, while $\mathcal{T}_R$, $\mathcal{T}_C$, and $\mathcal{T}_O$ are write-only tapes. As another special component,



$\mathcal{T}_O$ $\langle M(q,G)\rangle$
D_1
D_2
D_3
D_{c_M}
c_M
$\mathcal{T}_D$ $\langle G\rangle$
$\mathcal{O}_C$
$\mathcal{O}_S$
$\mathcal{T}_C$ $\langle q_C\rangle$
M
$\mathcal{T}_R$ $\langle q_R\rangle$
$\mathcal{T}_Q$ $\langle q\rangle$
(working tapes)

Fig. 2. $(\mathcal{L}_C, \mathcal{L}_S)$-LDFM M

the machine has an unbounded sequence $D_1, D_2, \ldots, D_k, \ldots$ of *result containers* (which can also be considered as read-only tapes), and a counter c_M, called the *result counter*, that defines the last used result container. M also has four different *modes*: *computing the next server request* (R), *waiting for response* (W), *computing client query* (C), and *done* (F). In all these modes the machine may use the full power of a standard Turing Machine. Additionally, 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 $\mathcal{L}_C$.

An LDFM M receives as input an RDF query q and an RDF graph G. Before the computation begins, q is assumed to be in tape $\mathcal{T}_Q$ and G is assumed to be in tape $\mathcal{T}_D$. All other tapes as well as the result containers are initially empty, the counter c_M is 0, and the machine is in mode R. Then, during the computation, the machine can use its ordinary working tapes arbitrarily (read/write). However, the access to the special tapes is restricted. That is, tape $\mathcal{T}_R$ can be used by the machine only when it is in mode R, and tape $\mathcal{T}_C$ can be used only in mode C. Moreover, the machine does not have direct access to the tapes $\mathcal{T}_D$ and $\mathcal{T}_O$; instead, the read-only tape $\mathcal{T}_D$ can be accessed only by the oracle $\mathcal{O}_S$, and the write-only tape $\mathcal{T}_O$ can be accessed only by oracle $\mathcal{O}_C$. Regarding the result containers, M is only able to read from them, and only oracle $\mathcal{O}_S$ can write in them. Figure 2 illustrates an LDFM and Fig. 3 shows the possible state transitions.

The computation of an LDFM M works as follows. While in mode R, the machine can construct a query $q_R \in \mathcal{L}_S$ and write it in tape $\mathcal{T}_R$. When the machine is finished writing q_R, it may change to mode W, which is a call to oracle $\mathcal{O}_S$. The oracle then increments the counter c_M, deletes the content of tape $\mathcal{T}_R$, and writes the set of mappings $[\![q_R]\!]_G$ in the container D_{c_M}. Next, the computation continues, M changes back to mode R, and the previous process may be repeated. Alternatively, at any point when in mode R, the machine may decide to change to mode C. In this mode, M constructs a query $q_C \in \mathcal{L}_C$, writes it in tape $\mathcal{T}_C$, and changes to mode F, which is a call to oracle $\mathcal{O}_C$. Then,

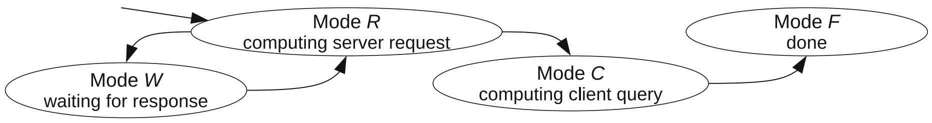


Fig. 3. Possible state transitions of an LDFM

oracle $\mathcal{O}_C$ evaluates q_C over data $D_1, \ldots, D_{c_M}$, and writes the result of this evaluation in tape $\mathcal{T}_O$ which is the final output of M. Hence, at this point the computation terminates. We denote the final output as $M(q, G)$.

Example 1. A typical SPARQL endpoint based client-server scenario may be captured by an LDFM M whose server language $\mathcal{L}_S$ is coreSparql and the response-combination language $\mathcal{L}_C$ is $\emptyset$. For any user query q, assuming q is in coreSparql, the machine simply copies q into tape $\mathcal{T}_R$ and enters mode W. After obtaining $[\![q]\!]_G$ from oracle $\mathcal{O}_S$ in result container D_1, the machine enters mode C, writes D_1 (as an expression in $\mathcal{L}_C = \emptyset$) in tape $\mathcal{T}_C$, and changes to mode F. Then, oracle $\mathcal{O}_C$ writes the query result $[\![q]\!]_G$ from D_1 to the output tape.

Example 2. Let M be an LDFM such that $\mathcal{L}_S$ = brTpf and $\mathcal{L}_C = \{\bowtie, \cup\}$. Hence, M has access to a server capable of handling brTpf requests, and M can do joins and unions to construct the final output. Assume now that a user wants to answer a SPARQL query q of the form $((?X, a, ?Y)$ And $(?X, b, ?Y))$, which is initially in tape $\mathcal{T}_Q$. Then, to evaluate q over a graph G (in tape $\mathcal{T}_D$), M may work as follows: First, M writes query $(?X, a, ?Y)$ in tape $\mathcal{T}_R$ and calls $\mathcal{O}_S$ by entering mode W. After this call we have that $D_1 = [\![(?X, a, ?Y)]\!]_G$. Now, M can write $(?X, b, ?Y)$ in tape $\mathcal{T}_R$, which is another call to $\mathcal{O}_S$ that produces $D_2 = [\![(?X, b, ?Y)]\!]_G$. Finally, M writes query $D_1 \bowtie D_2$ in tape $\mathcal{T}_C$ and calls $\mathcal{O}_C$, which produces the output, $M(q, G)$, in tape $\mathcal{T}_O$. It is not difficult to see that $M(q, G) = [\![q]\!]_G$. We may have an alternative LDFM M' that computes q as follows. Initially, M' calls the server oracle $\mathcal{O}_S$ with query $(?X, a, ?Y)$ to obtain $D_1 = [\![(?X, a, ?Y)]\!]_G$. Next, M' performs the following iteration: for every mapping $\mu \in D_1$ it writes the brTpf query $((?X, b, ?Y), \{\mu\})$ in $\mathcal{T}_R$ and calls the oracle $\mathcal{O}_S$ to produce $[\![((?X, b, ?Y), \{\mu\})]\!]_G = [\![(?X, b, ?Y))]\!]_G \bowtie \{\mu\}$ in one of its result containers. After all these calls, M' writes query $(D_2 \cup D_3 \cup \cdots \cup D_k)$ in tape $\mathcal{T}_C$, where $k = c_{M'}$ is the index of the last used result container. The oracle $\mathcal{O}_C$ then produces the final output $M'(q, G)$. In this case we also have that $M'(q, G) = [\![q]\!]_G$.

2.3 Rationale and Limitations of LDFMs

Machine models to formalize Web querying have been previously proposed in the literature [1,6,9]. Most of the early work in this context is based on an understanding of the Web as a distributed hypertext system consisting of Web pages that are interconnected by hyperlinks. These machines then formalized the notion of navigation and of data retrieval while navigating, and their focus was on classical computability issues (what can, and what cannot be computed in a distributed Web scenario). Though similar in motivation, our machine model in contrast formalizes a different approach to access and to query Web data. In this section we explain the rationale behind our design.

The perhaps most important characteristic of our model is that it separates the computation that creates the final output (as done by the client oracle $\mathcal{O}_C$) from the computation that plans and drives the overall query execution process (as done by the LDFM itself). Hence, the expressive power of the response-combination language $\mathcal{L}_C$ only determines how the query result to be returned to the user can be computed by using the result containers, but it does not have any impact whatsoever on the computations that the machine can do when it generates any of the server requests (in mode R) or when it generates the final $\mathcal{L}_C$-query (in mode C). This separation allows us to precisely pinpoint the computational power needed for the latter without mixing it up with the power needed for constructing the output (and vice versa). Of course, in practice the two tasks do not need to be separated into two consecutive phases as suggested by our model. In fact, an alternative version of our model could allow the machine to use oracle $\mathcal{O}_C$ multiple times to produce the first elements of the complete output as early as possible.

Another separation, which is perhaps more natural because it also exists in practice, is the delegation of the computation of the server responses to the server oracle $\mathcal{O}_S$. Besides also avoiding a mix-up when analyzing required computational power, this separation additionally allows us to prevent the LDFM from accessing the data tape $\mathcal{T}_D$ directly. This features captures the fact that, in practice, a client also has to use the server interface instead of being able to directly access the server-side dataset.

The result containers $(D_1, D_2, \ldots)$, with their corresponding result counter (c_M), provide us with an abstraction based on which notions of network cost of different pairs of client/server capabilities can be quantified. We shall use this abstraction to define network-related complexity measures in Sect. 5.

While our notion of the LDFM provides us with a powerful model to formally study many phenomena of LDF-based client-server settings, there are a few additional factors in practice that are not captured by the model in its current form. In particular, the model does not capture the option for the server to (i) decide to split responses into pages (that have to be requested separately) and (ii) send metadata with its responses that clients can use to adapt their query execution plans. Additionally, in practice there may be a cache located between the server and the client, which might have to be captured to study metrics related to server load (given that such a cache is not equally effective for different LDF interfaces [7,13]). We deliberately ignored these options to keep our model sufficiently simple. However, corresponding features may be added to our notion of an LDFM if useful for future analyses.

2.4 Computability and Expressiveness for LDFMs

We conclude the introduction of our machine model by defining notions of computability and expressiveness based on LDFMs.

The most basic notion of computability for LDFMs is that of a computable query. We say that an RDF query q *is computable under an LDFM* M if for every RDF graph G it holds that $M(q, G) = [\![q]\!]_G$. That is, q is computable under M if, with (q, G) as input, M produces $[\![q]\!]_G$ as output, for every possible graph G. We can also extend this notion to classes of queries. Formally, the *class of queries computed by* an LDFM M, denoted by $\mathcal{C}(M)$, is the set of all RDF queries that are computable under M.

Notice that every LDFM comes with a response-combination language and a server language, and thus we can also define classes of LDFMs in terms of the languages that they use. In particular, we say that an LDFM M is an $(\mathcal{L}_C, \mathcal{L}_S)$-LDFM if the response-combination language of M is $\mathcal{L}_C$ and the server language of M is $\mathcal{L}_S$. Now, we can define our main notion of computability.

Definition 1. *Let $\mathcal{L}_C$ be a response-combination language and $\mathcal{L}_S$ be a server language. A class $\mathcal{C}$ of RDF queries is* computable under $(\mathcal{L}_C, \mathcal{L}_S)$ *if there exists an $(\mathcal{L}_C, \mathcal{L}_S)$-LDFM M such that every query q in $\mathcal{C}$ is computable under M.*

Definition 1 is our main building block to compare different combinations of client and server languages independent of the possible LDFMs that use these languages. The following definition formalizes our main comparison notion.

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. Then, $(\mathcal{L}_1', \mathcal{L}_2')$ 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 class of queries that is computable under $(\mathcal{L}_1, \mathcal{L}_2)$ is also computable under $(\mathcal{L}_1', \mathcal{L}_2')$.*

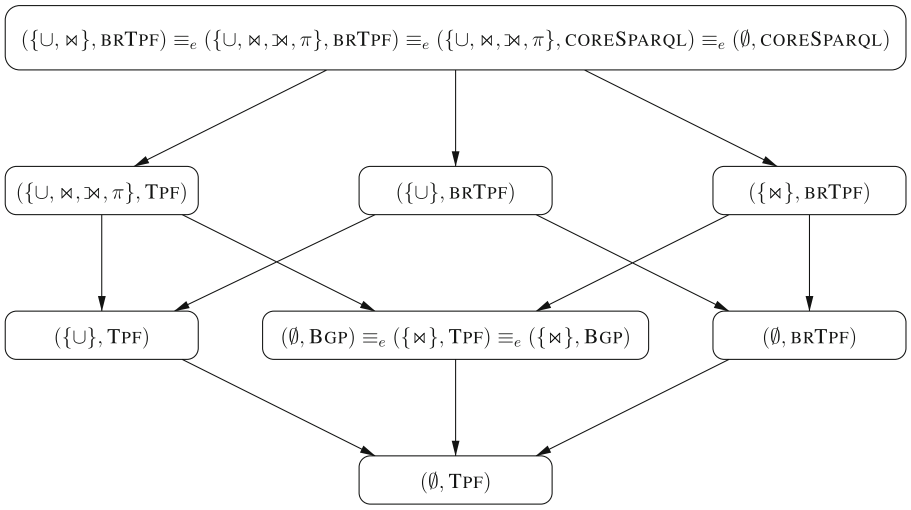
We use $(\mathcal{L}_1, \mathcal{L}_2) \equiv_e (\mathcal{L}_1', \mathcal{L}_2')$ to denote that $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}_1', \mathcal{L}_2')$ are *equally expressive*, that is, $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}_1', \mathcal{L}_2')$ and $(\mathcal{L}_1', \mathcal{L}_2') \preceq_e (\mathcal{L}_1, \mathcal{L}_2)$. As usual, we write $(\mathcal{L}_1, \mathcal{L}_2) \prec_e (\mathcal{L}_1', \mathcal{L}_2')$ to denote that $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}_1', \mathcal{L}_2')$ and $(\mathcal{L}_1', \mathcal{L}_2') \not\preceq_e (\mathcal{L}_1, \mathcal{L}_2)$.

Example 3. It is easy to show that $(\emptyset, \textsc{Dump}) \prec_e (\emptyset, \textsc{Tpf})$. That is, whenever you have a server that can only provide a Dump of its dataset and you do not have any additional power in the client, then you can accomplish strictly less tasks compared with the case in which you have access to a server that can answer Tpf queries. In the next section we prove more such relationships (including less trivial ones).

3 Expressiveness Lattice

In this section we show the relationships between different pairs of client and server capabilities in terms of expressiveness. In particular, we establish a lattice that provides a full picture of many combinations of the server languages mentioned in Sect. 2.1 with almost every possible response-combination language constructed by using some of the algebra operators in $\{\bowtie, \cup, \rtimes, \pi\}$. Figure 4 illustrates this expressiveness-related lattice. As we will show, some of the equivalences and separations in this lattice do not necessarily follow from standard expressiveness results in the query language literature. In particular, the lattice highlights the expressive power of using the brTpf interface [7]. It should be noticed that several other combinations of response-combination languages and server languages might have been considered. We plan to cover more of them as part of our future work. Before going into the results, we make the following simple observation about the expressiveness of LDFMs.

Note 1. Let $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$ be arbitrary pairs of response-combination/server languages s.t. $\mathcal{L}_1 \subseteq \mathcal{L}'_1$ and $\mathcal{L}_2 \subseteq \mathcal{L}'_2$. Then, it is easy to prove that $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$.



({∪, ⋈}, brTpf) ≡e ({∪, ⋈, ⟖, π}, brTpf) ≡e ({∪, ⋈, ⟖, π}, coreSparql) ≡e (∅, coreSparql)
({∪, ⋈, ⟖, π}, Tpf)
({∪}, brTpf)
({⋈}, brTpf)
({∪}, Tpf)
(∅, Bgp) ≡e ({⋈}, Tpf) ≡e ({⋈}, Bgp)
(∅, brTpf)
(∅, Tpf)

Fig. 4. Expressiveness lattice for LDFMs

3.1 The Expressiveness of Using the brTpf Interface

We begin with a result that shows that brTpf in combination with join and union in the client side is as expressive as server-side coreSparql with $\{\bowtie, \cup, \rtimes, \pi\}$ in the client.

Theorem 1. $(\{\cup, \bowtie\}, \text{brTpf}) \equiv_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{coreSparql})$.

The result, that might seem surprising, follows from two facts: (1) an LDFM can use unbounded computational power to issue server requests, and (2) a brTpf server can accept arbitrary solutions mappings to be joined with triple patterns in the server side. The proof is divided in several parts and exploits a trick that is used in practice to avoid client-side joins when accessing a brTpf interface. We illustrate the main idea with an example. Assume that one wants to compute a SPARQL query P of the form $(t_1 \text{O}{\small\text{PT}} t_2)$ over G where t_1 and t_2 are triple patterns. Since $[\![P]\!]_G = [\![t_1]\!]_G ⟕ [\![t_2]\!]_G$, one can easily evaluate P with a $(\{\bowtie, \cup, ⟕, \pi\}, \text{coreSparql})$-LDFM by just evaluating t_1 and t_2 separately in the server, and then using $⟕$ in the client to construct the final output. On the other hand, one can use the following strategy to evaluate P with a $(\{\cup, \bowtie\}, \text{brTpf})$-LDFM M. Recall that

$$[\![t_1]\!]_G ⟕ [\![t_2]\!]_G = ([\![t_1]\!]_G \bowtie [\![t_2]\!]_G) \cup ([\![t_1]\!]_G \smallsetminus [\![t_2]\!]_G),$$

where $[\![t_1]\!]_G \smallsetminus [\![t_2]\!]_G$ is the set of all mappings in $[\![t_1]\!]_G$ that are not compatible with any mapping in $[\![t_2]\!]_G$ [2]. We can first evaluate t_1 in the server to obtain $[\![t_1]\!]_G$ as one of M's result containers, say D_1. Next, M can use D_1 to construct the brTpf query $(t_2, [\![t_1]\!]_G)$, which can be evaluated in the server and stored in the next container D_2. Notice that D_2 now contains all mappings in $[\![t_2]\!]_G$ that can be joined with some mapping in $[\![t_1]\!]_G$. Now M can use its internal computational power to produce the following set of queries: for every mapping μ in D_1 that is not compatible with any mapping in D_2, M constructs the brTpf query $(t_1, \{\mu\})$, sends it to the server, and stores the result in one of the result containers, starting in container D_3. Notice that M is essentially mimicking the difference operator $\smallsetminus$ using one mapping at a time. After all these requests, M has all the mappings of the set $[\![t_1]\!]_G \smallsetminus [\![t_2]\!]_G$ stored in its containers, every mapping in a different container. Moreover, given that $D_1 \bowtie D_2 = [\![t_1]\!]_G \bowtie [\![t_2]\!]_G$, M can generate the client query $(D_1 \bowtie D_2) \cup D_3 \cup \cdots \cup D_{c_M}$ which will give exactly $[\![t_1]\!]_G ⟕ [\![t_2]\!]_G$. A similar strategy can be used to compute all other operators.

It is not difficult to prove that when having coreSparql for server requests, the operators $\{\bowtie, \cup, ⟕, \pi\}$ on the client do not add any expressiveness. Moreover, from proving Theorem 1 it is easy to also obtain that $(\{\cup, \bowtie\}, \text{brTpf}) \equiv_e (\{\bowtie, \cup, ⟕, \pi\}, \text{brTpf})$. Thus, we have that all the following four settings are equivalent in expressiveness:

$$\begin{aligned}(\{\cup, \bowtie\}, \text{brTpf}) &\equiv_e (\{\cup, \bowtie, ⟕, \pi\}, \text{brTpf}) \\ &\equiv_e (\{\bowtie, \cup, ⟕, \pi\}, \text{coreSparql}) \equiv_e (\emptyset, \text{coreSparql}).\end{aligned}$$

These equivalences are shown at the top of the lattice in Fig. 4.

Theorem 1 has several practical implications. One way to read this result is that whenever a brTpf interface is available, a machine having operators $\{\bowtie, \cup, ⟕, \pi\}$ in the client has plenty of options to produce *query execution plans* to answer user queries. In particular, for user queries needing $⟕$ or π, the machine may decide if some of these operators are evaluated in the client or part of them are evaluated in the server. What Theorem 1 does not state is an estimation of the cost of executing these different plans. In Sect. 5 we shed some light on this issue, in particular, we study the additional cost payed when using different server interfaces in terms of the number of requests sent to the server and the size of the data transferred between server and client.

The following result shows that union in the client is essential to obtain Theorem 1.

Theorem 2. $(\{\bowtie\}, \textsc{brTpf}) \prec_e (\{\cup, \bowtie\}, \textsc{brTpf})$.

It should be noticed that this result does *not* directly follow from the fact that $\cup$ cannot be expressed using $\bowtie$ since, as we have shown, a brTpf interface is very expressive when queried with unbounded computational power. Towards proving Theorem 2, it is clear that $(\{\bowtie\}, \textsc{brTpf}) \preceq_e (\{\cup, \bowtie\}, \textsc{brTpf})$. Thus, to prove the theorem it only remains to show that $(\{\cup, \bowtie\}, \textsc{brTpf}) \not\preceq_e (\{\bowtie\}, \textsc{brTpf})$. The following lemma proves something that, by Note 1 above, is actually stronger.

Lemma 1. $(\{\cup\}, \textsc{Tpf}) \not\preceq_e (\{\bowtie, ⟕, \pi\}, \textsc{brTpf})$.

Consider the coreSparql query $q = ((?X, a, 2)\textsc{Union}\,(3, b, 4))$. It is clear that q is computable by a $(\{\cup\}, \textsc{Tpf})$-LDFM. It can be proved that q is not computable by a $(\{\bowtie, ⟕, \pi\}, \textsc{brTpf})$-LDFM.

The following result proves that join is also needed to obtain Theorem 1.

Theorem 3. $(\{\cup\}, \textsc{brTpf}) \prec_e (\{\cup, \bowtie\}, \textsc{brTpf})$.

As for Theorem 2, we only need to prove that $(\{\cup, \bowtie\}, \textsc{brTpf}) \not\preceq_e (\{\cup\}, \textsc{brTpf})$ which follows from the next, stronger result.

Lemma 2. $(\{\bowtie\}, \textsc{Tpf}) \not\preceq_e (\{\cup, \pi\}, \textsc{brTpf})$.

The lemma follows from the fact that a $(\{\bowtie\}, \textsc{Tpf})$-LDFM can produce solution mappings with an unbounded number of variables in its domain while, given the restrictions of the brTpf interface, every solution mapping in the output of a $(\{\cup, \pi\}, \textsc{brTpf})$-LDFM has at most three variables in its domain.

3.2 The Expressiveness of Using the Tpf Interface

One interesting point is the comparison between Tpf and brTpf. The first important question is whether Theorem 1 can be obtained by considering Tpf instead of brTpf. Our next result provides a negative answer.

Theorem 4. $(\{\bowtie, \cup, ⟕, \pi\}, \textsc{Tpf}) \prec_e (\{\cup, \bowtie\}, \textsc{brTpf})$.

We have that $(\{\bowtie, \cup, ⟕, \pi\}, \textsc{Tpf}) \preceq_e (\{\bowtie, \cup, ⟕, \pi\}, \textsc{coreSparql})$ because it holds that $\textsc{Tpf} \subseteq \textsc{coreSparql}$. By combining this with Theorem 1 we obtain that $(\{\bowtie, \cup, ⟕, \pi\}, \textsc{Tpf}) \preceq_e (\{\cup, \bowtie\}, \textsc{brTpf})$. Thus, to prove Theorem 4 it remains to show that $(\{\cup, \bowtie\}, \textsc{brTpf}) \not\preceq_e (\{\bowtie, \cup, ⟕, \pi\}, \textsc{Tpf})$. We prove something stronger:

Lemma 3. $(\emptyset, \textsc{brTpf}) \not\preceq_e (\{\bowtie, \cup, ⟕, \pi\}, \textsc{Tpf})$.

It turns out that Filter is all that one needs to add to Tpf to make it comparable with brTpf. In fact, in terms of expressive power of LDFMs, Tpf with Filter and brTpf are equivalent regardless of the client language.

Proposition 1. $(\mathcal{L}, \text{brTpf}) \equiv_e (\mathcal{L}, \text{Tpf} + \text{Filter})$ *holds for every response-combination language* $\mathcal{L}$.[1]

Given Proposition 1, in every combination in the lattice of Fig. 4 we can replace brTpf by Tpf+Filter and the relationships still hold.

The next result shows an equivalence concerning Tpf and Bgp.

Proposition 2. $(\emptyset, \text{Bgp}) \equiv_e (\{\bowtie\}, \text{Tpf}) \equiv_e (\{\bowtie\}, \text{Bgp})$

Our final result in this section is a set of incompatibilities for Tpf and brTpf which follow from our previous results.

Corollary 1. *The following relationships hold.*

1. $(\{\bowtie, \cup, ⟕, \pi\}, \text{Tpf})$ *and* $(\emptyset, \text{brTpf})$ *are not comparable in terms of* $\preceq_e$.
2. $(\{\cup\}, \text{Tpf})$ *and* $(\{\bowtie\}, \text{brTpf})$ *are not comparable in terms of* $\preceq_e$.
3. $(\{\bowtie\}, \text{Tpf})$ *and* $(\{\cup\}, \text{brTpf})$ *are not comparable in terms of* $\preceq_e$.

The lattice of the expressiveness of LDFMs shown in Fig. 4 is constructed by composing all the results in this section.

4 Comparisons Based on Classical Complexity Classes

Besides expressiveness, another classical measure is the (computational) complexity of query evaluation. In this section we present a simple analysis to provide a comparison of LDFs settings in terms of the complexity of the query evaluation problem for the server and response-combination languages. In particular, we focus on the *combined complexity* that measures the complexity of problems for which a query and a dataset are both assumed to be given as input [12]. We begin by defining two new comparison notions.

Definition 3. *We say that* $(\mathcal{L}_1, \mathcal{L}_2)$ is at most as server-power demanding as $(\mathcal{L}'_1, \mathcal{L}'_2)$, *denoted by* $(\mathcal{L}_1, \mathcal{L}_2) \preceq_{sp} (\mathcal{L}'_1, \mathcal{L}'_2)$, *if the combined complexity of the evaluation problem for* $\mathcal{L}_2$ *is at most as high as the combined complexity of the evaluation problem for* $\mathcal{L}'_2$. *Similarly,* $(\mathcal{L}_1, \mathcal{L}_2)$ is at most as result-construction demanding as $(\mathcal{L}'_1, \mathcal{L}'_2)$, *denoted by* $(\mathcal{L}_1, \mathcal{L}_2) \preceq_{rc} (\mathcal{L}'_1, \mathcal{L}'_2)$, *if the combined complexity of the evaluation problem for* $\mathcal{L}_1$ *is at most as high as the combined complexity of the evaluation problem for* $\mathcal{L}'_1$.

[1] This result and the next are given as propositions instead of theorems because they are simple to prove with standard notions of logic (as detailed in the aforementioned appendix of this paper) and they do not add an important separation in the expressiveness lattice (Fig. 4).

We write $(\mathcal{L}_1, \mathcal{L}_2) \equiv_c (\mathcal{L}'_1, \mathcal{L}'_2)$ if $(\mathcal{L}_1, \mathcal{L}_2) \preceq_c (\mathcal{L}'_1, \mathcal{L}'_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2) \preceq_c (\mathcal{L}_1, \mathcal{L}_2)$, for $c \in \{sp, rc\}$. The next result follows trivially from the results of Pérez et al. [10] and Schmidt et al. [11] that show that for the And-fragment and the Union-fragment of SPARQL, the evaluation problem is in PTime, respectively, for the And-Union-fragment it is NP-complete, and for fragments containing Opt it is PSpace-complete.

Corollary 2. *For any server language $\mathcal{L}_S$, the following properties hold:*

1. $(\emptyset, \mathcal{L}_S) \equiv_{rc} (\{\bowtie\}, \mathcal{L}_S) \equiv_{rc} (\{\cup\}, \mathcal{L}_S)$
2. $(\emptyset, \mathcal{L}_S) \preceq_{rc} (\{\bowtie, \cup\}, \mathcal{L}_S) \preceq_{rc} (\{⟕\}, \mathcal{L}_S)$
3. $(\{⟕\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, ⟕\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, ⟕, \cup\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, ⟕, \cup, \pi\}, \mathcal{L}_S)$

Moreover, for any response-combination language $\mathcal{L}_C$, the following properties hold:

4. $(\mathcal{L}_C, \textsc{Bgp}) \equiv_{sp} (\mathcal{L}_C, \textsc{brTpf}) \equiv_{sp} (\mathcal{L}_C, \textsc{Tpf}) \equiv_{sp} (\mathcal{L}_C, \textsc{Dump})$
5. $(\mathcal{L}_C, \textsc{Bgp}) \preceq_{sp} (\mathcal{L}_C, \textsc{coreSparql})$

Notice that the pairs of response-combination and server languages mentioned in the corollary can be organized into two additional lattices along the lines of the expressiveness lattice in Fig. 4. That is, Properties 1–3 in Corollary 2 establish a *result-construction demand lattice*, and Properties 4 and 5 establish a *server-power demand lattice*. However, both of these lattices consist of only a single path from top to bottom.

5 Additional Complexity Measures

In the previous sections we provide a base for comparing different combinations of client/server capabilities considering expressiveness and complexity. While these comparisons are a necessary starting point, from a practical point of view one would also want to compare the computational resources that have to be payed when using one LDF interface or another. More specifically, assume that you have two combinations of client and server capabilities that are equally expressive, that is, $(\mathcal{L}_1, \mathcal{L}_2) \equiv_e (\mathcal{L}'_1, \mathcal{L}'_2)$. Then, we know that every task that can be completed in $(\mathcal{L}_1, \mathcal{L}_2)$ can also be completed in $(\mathcal{L}'_1, \mathcal{L}'_2)$. The question however is: are we paying an additional cost when using one setting 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 this section we show the suitability of our proposed framework to also analyze this aspect of LDFs.

We begin this section with a definition that formalizes two important resources used when consuming Linked Data Fragments, namely, the number of requests sent to the server, and the total size of the data transferred from the server to the client.

Definition 4. *For an LDFM M, an RDF query q, and an RDF graph G, we define the* number of requests of M with input (q, G)*, denoted by $r_M(q, G)$, as* *the final value of counter c_M during the computation of M with input (q, G). Similarly, the* amount of data transferred by M with input (q, G)*, denoted by $t_M(q, G)$, is defined as the value $|D_1| + |D_2| + \cdots + |D_{r_M(q,G)}|$.*

We can now define the request and transfer complexity of classes of RDF queries.

Definition 5. *Let f be a function from the natural numbers. A class $\mathcal{C}$ of RDF queries has* request complexity at most f under $(\mathcal{L}_1, \mathcal{L}_2)$ *if there exists an $(\mathcal{L}_1, \mathcal{L}_2)$-LDFM M that computes every query $q \in \mathcal{C}$ such that for every $q \in \mathcal{C}$ and RDF graph G it holds that $r_M(q, G) \leq f(|q| + |G|)$. Similarly we say that $\mathcal{C}$ has* transfer complexity at most f under $(\mathcal{L}_1, \mathcal{L}_2)$ *if there exists an $(\mathcal{L}_1, \mathcal{L}_2)$-LDFM M that computes every $q \in \mathcal{C}$ such that $t_M(q, G) \leq f(|q| + |G|)$ for every $q \in \mathcal{C}$ and RDF graph G.*

We now have all the necessary to present our main notions to compare different classes of RDF queries in terms of the resources needed to compute them with LDFMs.

Definition 6. *Let $\mathcal{L}_1, \mathcal{L}'_1$ be response-combination languages and $\mathcal{L}_2, \mathcal{L}'_2$ be server languages. Then,* $(\mathcal{L}_1, \mathcal{L}_2)$ 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 class $\mathcal{C}$ of RDF queries expressible in both $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$, if $\mathcal{C}$ has request complexity at most f under $(\mathcal{L}'_1, \mathcal{L}'_2)$, then $\mathcal{C}$ has request complexity at most f under $(\mathcal{L}_1, \mathcal{L}_2)$. We similarly define the notions of being* at most as data-transfer demanding*, and denote it using $\preceq_t$.*

Regarding the notions in Definition 6 we make the following general observation.

Note 2. Let $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$ be arbitrary pairs of response-combination/server languages s.t. $\mathcal{L}_1 \subseteq \mathcal{L}'_1$ and $\mathcal{L}_2 \subseteq \mathcal{L}'_2$. Since $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$, any $(\mathcal{L}'_1, \mathcal{L}'_2)$-LDFM that can be used to compute the class of RDF queries computable under $(\mathcal{L}'_1, \mathcal{L}'_2)$ can also be used to compute every RDF query that is computable under $(\mathcal{L}_1, \mathcal{L}_2)$. Therefore, it follows trivially that $(\mathcal{L}'_1, \mathcal{L}'_2) \preceq_r (\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2) \preceq_t (\mathcal{L}_1, \mathcal{L}_2)$.

We next show some (less trivial) results that provide more specific comparisons with respect to the above introduced notions. To this end, we write $(\mathcal{L}_1, \mathcal{L}_2) \prec_c (\mathcal{L}'_1, \mathcal{L}'_2)$ to denote that $(\mathcal{L}_1, \mathcal{L}_2) \preceq_c (\mathcal{L}'_1, \mathcal{L}'_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2) \not\preceq_c (\mathcal{L}_1, \mathcal{L}_2)$, for $c \in \{r, t\}$.

Recall that $(\emptyset, \text{BGP})$, $(\{\bowtie\}, \text{TPF})$, and $(\{\bowtie\}, \text{BGP})$ are all equivalent in terms of expressive power. The next result proves formally that, in terms of the data transferred, they can actually be separated.

Proposition 3. *It holds that $(\{\bowtie\}, \text{BGP}) \prec_t (\{\bowtie\}, \text{TPF})$. Moreover, $(\{\bowtie\}, \text{TPF})$ and $(\emptyset, \text{BGP})$ are not comparable in terms of $\preceq_t$. Regarding the number of requests it holds that $(\{\bowtie\}, \text{BGP}) \equiv_r (\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{TPF})$.*

To see why the $\preceq_t$ incomparability result holds, consider the class $\mathcal{C}_1$ of SPARQL queries of the form $((?X_1, ?Y_1, ?Z_1)$ AND $(?X_2, ?Y_2, ?Z_2))$. It can be shown that any $(\emptyset, \textsc{Bgp})$-LDFM M that computes $\mathcal{C}_1$ is such that $t_M(q, G)$, as a function, is in $\Omega(|G|^2)$. On the other hand there exists a $(\{\bowtie\}, \textsc{Tpf})$-LDFM M' such that $t_{M'}(q, G)$ is in $O(|G|)$. This shows that $(\emptyset, \textsc{Bgp}) \not\preceq_t (\{\bowtie\}, \textsc{Tpf})$. Consider now the class $\mathcal{C}_2$ of SPARQL queries of the form $((a_1, b_1, c_1)$ AND $\cdots$ AND $(a_k, b_k, c_k))$. One can show that any $(\{\bowtie\}, \textsc{Tpf})$-LDFM M that computes $\mathcal{C}_2$ is such that $t_M(q, G)$ is in $\Omega(|q|)$ in the worst case. On the other hand, $\mathcal{C}_2$ can be computed with a $(\emptyset, \textsc{Bgp})$-LDFM that, in the worst case, transfers a single mapping (the complete query result) thus showing that $(\{\bowtie\}, \textsc{Tpf}) \not\preceq_t (\emptyset, \textsc{Bgp})$. Class $\mathcal{C}_2$ can also be used to show that $(\emptyset, \textsc{Bgp}) \prec_r (\{\bowtie\}, \textsc{Tpf})$. Our final result shows that even though $(\{\cup, \bowtie\}, \textsc{brTpf})$ is very expressive, one may need to pay an extra overhead in terms of transfer and request complexity compared with a setting with a richer response-combination language.

Theorem 5. *The following strict relationships hold.*

1. $(\{\cup, \bowtie, ⟕, \pi\}, \textsc{brTpf}) \prec_t (\{\cup, \bowtie\}, \textsc{brTpf})$
2. $(\{\cup, \bowtie, ⟕, \pi\}, \textsc{brTpf}) \prec_r (\{\cup, \bowtie\}, \textsc{brTpf})$

The first point of this last theorem can be intuitively read as follows: in terms of bandwidth, the best possible query plans for an LDFM that access a brTpf interface and then construct the output using operators in $\{\cup, \bowtie, ⟕, \pi\}$, are strictly better than the best possible query plans that access a brTpf interface and then construct the output using operators in $\{\cup, \bowtie\}$. The second point has a similar interpretation regarding the best possible query plans in terms of the number of requests sent to the server.

Although in this section we did not present a complete lattice as for the case of expressiveness in Sect. 3, these results show the usefulness of our framework to formally compare different options of Linked Data Fragments.

6 Concluding Remarks and Future Work

In this paper we have presented LDFMs, the first formalization to model LDF scenarios. By proving formal results based on LDFMs we show the usefulness of our model to analyze the fine-grain interplay between several metrics. We think that our formalization is a first step towards a theory to compare different access protocols for Semantic Web data. We next describe some possible directions for future research regarding LDFMs, extensions to the model, and its usage in some alternative scenarios.

In this paper we consider a specific set of client and server capabilities but our framework is by no means tailored to them. In particular, it would be really interesting to consider more expressive operators in the client languages and also new LDF interfaces, and compare them with the ones presented in this paper. One notable interface that is widely used in practice and that we plan to integrate in our study is the URI-lookup interface to retrieve Linked Data documents [4,5].

Besides the classical metrics (expressiveness and computational complexity), in this paper we considered only the number of requests sent to the server and the data transferred from server to client. It is easy to include other practical metrics in our framework. One important practical metric might be the amount of data transferred from the client to the server. In particular this metric might be very important for the brTpf interface which requires sending solution mappings from the client to the server. Notice that this metric can be formalized by simply considering the space complexity on the request tape $\mathcal{T}_R$ of an LDFM. Similarly, if we consider the space complexity of the client query tape $\mathcal{T}_C$, then we can restrict the size of the output query which makes sense as a restriction for clients with local memory constraints.

Finally, our model and results can be used as a first step towards a foundation for the theoretical study of Semantic Web query planning; more specifically, we would like to compile into our model already proposed languages for querying Linked Data, and to formally study what are the server interfaces and client capabilities needed to execute queries expressed in these languages, considering also the cost of compilation and execution according to our formal metrics. One possible starting point would be to study languages designed for live queries on the Web of Linked Data. For instance, we have recently proposed LDQL [8], which is a navigational language designed to query Semantic Web data based on the URI-lookup interface. Although we have presented a fairly complete formal analysis of LDQL [8], the computational complexity considered was only a classical analysis that disregards some important features of querying the Web such as server communication, latency, etc. Our machine model plus the results on comparisons of different LDFs can help to derive a more realistic complexity analysis for languages such as LDQL. We plan to tackle this problem in our future work.

Acknowledgements. Hartig's work has been funded by the CENIIT program at Linköping University (project no. 17.05). Pérez is supported by the Millennium Nucleus Center for Semantic Web Research NC120004, and ENLACE-Fondecyt VID-UChile.

References

1. Abiteboul, S., Vianu, V.: Queries and computation on the web. Theor. Comput. Sci. **239**(2), 231–255 (2000)
2. Arenas, M., Gutierrez, C., Miranker, D.P., Pérez, J., Sequeda, J.: Querying semantic data on the web. SIGMOD Rec. **41**(4), 6–17 (2012)
3. Beek, W., Rietveld, L., Bazoobandi, H.R., Wielemaker, J., Schlobach, S.: LOD laundromat: a uniform way of publishing other people's dirty data. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 213–228. Springer, Cham (2014). doi:10.1007/978-3-319-11964-9_14
4. Berners-Lee, T.: Design issues: linked data, July 2006
5. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semant. Web Inf. Syst. **5**(3), 1–22 (2009)

6. Hartig, O.: SPARQL for a web of linked data: semantics and computability. In: Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V. (eds.) ESWC 2012. LNCS, vol. 7295, pp. 8–23. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30284-8_8
7. Hartig, O., Buil-Aranda, C.: Bindings-restricted triple pattern fragments. In: Debruyne, C., Panetto, H., Meersman, R., Dillon, T., Kühn, E., O'Sullivan, D., Ardagna, C.A. (eds.) OTM 2016. LNCS, vol. 10033, pp. 762–779. Springer, Cham (2016). doi:10.1007/978-3-319-48472-3_48
8. Hartig, O., Pérez, J.: LDQL: a query language for the web of linked data. J. Web Semant. **41**, 9–29 (2016)
9. Mendelzon, A.O., Milo, T.: Formal models of web queries. Inf. Syst. **23**(8), 615–637 (1998)
10. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3), 16:1–16:45 (2009)
11. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proceedings of the 13th International Conference on Database Theory (ICDT) (2010)
12. Vardi, M.Y.: The complexity of relational query languages. In: STOC (1982)
13. Verborgh, R., Sande, M.V., Hartig, O., Herwegen, J.V., Vocht, L.D., Meester, B.D., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web. J. Web Semant. **37–38**, 184–206 (2016)