A Formal Framework for Comparing Linked Data Fragments

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

[1] Dept. of Computer and Information Science (IDA), Linköping University, Sweden
olaf.hartig@liu.se
[2] Departamento de Ingeniería Matemática, Universidad de Chile
iletter@dim.uchile.cl
[3] Department of Computer Science, Universidad de 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 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 [14]. Moreover, every interface has its own particular properties regarding performance, bandwidth needs, cache effectiveness, etc. Clients can analyze the tradeoffs 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) [14], and Bindings-Restricted Triple Pattern Fragments (brTPF) [8]. Different options for LDF interfaces are shown in Figure 1.

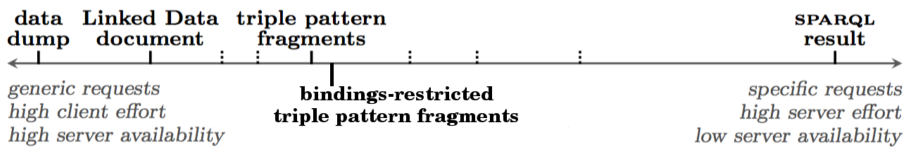


Fig. 1. Unidimensional view of Linked Data Fragments (from [8,14])

Linked Data Fragments have already had a considerable practical impact. For instance, since the proposal of TPFs, the LOD Laundromat Website has published more than 650,000 datasets on the Web with this interface [3]. Moreover, DBPedia [6] has also published a TPF interface which had an uptime of 99.99% during its first nine months working [14]. Up to now, the research and development of LDFs have produced interesting practical results, but they are definitely not the final answer to querying semantic data on the Web, and one may expect that many new fragments with different tradeoffs 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 tradeoff. 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.

One of our main hypotheses is that it is possible to compare LDF interfaces in different ways beyond a single axis as the one shown in Figure 1. In that figure three criteria are considered: (1) client vs. server effort, (2) general vs. specific requests, and (3) high vs. low availability. The figure suggests that whenever we need more specific requests, then we will have more server effort and less availability. Similarly, more general requests would imply higher availability and less server effort (and thus more client work) [14]. More generally, the single axis implies that these three criteria are correlated and, for example, the Linked Data Documents interface is always in between data dumps and SPARQL endpoints. However, lets 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."

It can be argued that $Q1$ cannot be answered by accessing a dataset using the Linked Data Document interface no matter how many requests the client sends to the server [7]. On the other hand, it is not difficult to show that both data dumps and SPARQL endpoints can actually answer $Q1$. 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 Section 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 Section 3, in Section 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 Section 5.

For the sake of space most of the details on the proofs have been omitted but can be found 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 resides in the client side and is in charge of the communication with the server. To this end, 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 *client 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 client 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 three types of queries: user queries, requests, and transformations.

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 [11,2] 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 [14].
- TPF+FILTER is the language composed of queries of the form $(tp \text{ FILTER } \theta)$ where tp is a triple pattern and θ is a SPARQL built-in condition [11].
- 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 [8]. 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 [12], $\text{vars}(tp)$ is the set of variables in tp, $[\![tp]\!]_G$ is the evaluation of tp over G [11], and $\bowtie$ is the join operator [11]. 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.
- URIF (URI-lookup fragment) is the language composed of queries of the form $(u, ?p, ?o)$ where u is a URI and $?p$ and $?o$ are distinct variables.

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.

Server-response transformations 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 server-response transformations can be expressed using languages that resemble an algebra over sets of solution mappings. We call such a language a *client language*. In this paper we denote such client languages by the set of algebra operators that they implement. For instance, the client language denoted by the set $\{\bowtie, \pi\}$ can be used

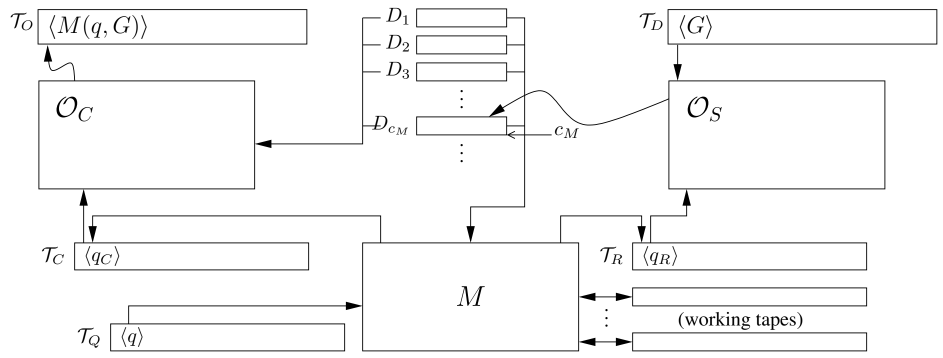


Fig. 2. $(\mathcal{L}_C, \mathcal{L}_S)$-LDFM M

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 client 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 having 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, 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 works as 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 client 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 $\mathcal{O}_S$ is able to write in them. Figure 2 shows a diagram of an LDFM.

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, oracle $\mathcal{O}_C$ evaluates q_C over data $D_1, \dots, D_{c_M}$, 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 scenario may be captured by an LDFM M whose server language $\mathcal{L}_S$ is coreSparql and the client 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 enters 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 $((?X, a, ?Y)\ \text{And}\ (?X, b, ?Y))$, 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 [10,1,7]. 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 protocol to access Web data. In this section we explain the rationale behind our design which also differentiates it from previous proposals.

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). 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 Section 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) 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 [8,14]). We deliberately ignored these options to keep our model sufficiently simple. However, corresponding features may easily 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 client 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 client 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 client 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 $\mathcal{L}_1, \mathcal{L}'_1$ be client languages and $\mathcal{L}_2, \mathcal{L}'_2$ 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 $(\emptyset, \text{DUMP}) \prec_e (\emptyset, \text{TPF})$. 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

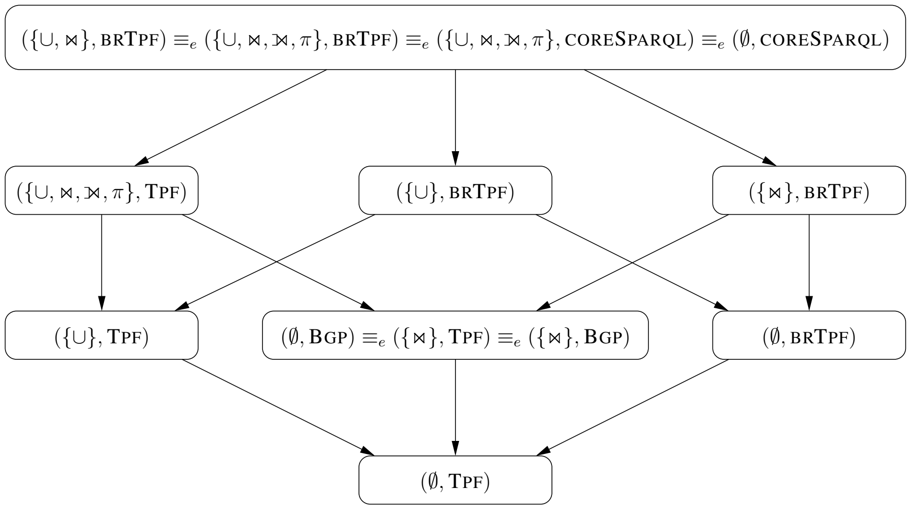
In this section we show the relationships between different pairs of client and server capabilities in terms of expressiveness. In particular, we establish a lattice that provides a full picture of many combinations of the server languages mentioned in Section 2.1 with almost every possible client language constructed by using some of the algebra operators in $\{\bowtie, \cup, ⟕, \pi\}$. Figure 3 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 [8]. It should be noticed that several other combinations of client and server languages might have been considered. We plan to cover more of them in the extended version of this paper.

Before going into the results, we make the following simple observation about the expressiveness of LDFMs. Let $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$ be arbitrary pairs of client-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 $(\mathcal{L}_1, \mathcal{L}_2) \preceq_e (\mathcal{L}'_1, \mathcal{L}'_2)$.

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 $\{\bowtie, \cup, ⟕, \pi\}$ in the client.

Theorem 1. $(\{\cup, \bowtie\}, \text{brTPF}) \equiv_e (\{\bowtie, \cup, ⟕, \pi\}, \text{coreSPARQL})$.



$(\{\cup, \bowtie\}, \text{BRTPF}) \equiv_e (\{\cup, \bowtie, \leftouterjoin, \pi\}, \text{BRTPF}) \equiv_e (\{\cup, \bowtie, \leftouterjoin, \pi\}, \text{CORESPARQL}) \equiv_e (\emptyset, \text{CORESPARQL})$
$(\{\cup, \bowtie, \leftouterjoin, \pi\}, \text{TPF})$
$(\{\cup\}, \text{BRTPF})$
$(\{\bowtie\}, \text{BRTPF})$
$(\{\cup\}, \text{TPF})$
$(\emptyset, \text{BGP}) \equiv_e (\{\bowtie\}, \text{TPF}) \equiv_e (\{\bowtie\}, \text{BGP})$
$(\emptyset, \text{BRTPF})$
$(\emptyset, \text{TPF})$

Fig. 3. Expressiveness lattice for LDFMs

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 $(t_1 \text{ OPT } t_2)$ over G where t_1 and t_2 are triple patterns. Since $[\![P]\!]_G = [\![t_1]\!]_G \leftouterjoin [\![t_2]\!]_G$, one can easily evaluate P with a $(\{\bowtie, \cup, \leftouterjoin, \pi\}, \text{CORESPARQL})$-LDFM by just evaluating t_1 and t_2 separately in the server, and then using $\leftouterjoin$ 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

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

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 $[\![t_1]\!]_G \rtimes [\![t_2]\!]_G$. 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 $\{\bowtie, \cup, \rtimes, \pi\}$ on the client do not add any expressiveness. Moreover, from proving Theorem 1 it is easy to also obtain that $(\{\cup, \bowtie\}, \text{brTpf}) \equiv_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{brTpf})$. Thus, we have that all the following four settings are equivalent in expressiveness:

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

These equivalences are shown at the top of the lattice in Figure 3.

Theorem 1 has several practical implications. One way to read this result is that whenever a brTpf interface is available, a machine having operators $\{\bowtie, \cup, \rtimes, \pi\}$ in the client has plenty of options to produce *query execution plans* to answer user queries. In particular, for user queries needing $\rtimes$ 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 Section 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. $(\{\bowtie\}, \text{brTpf}) \prec_e (\{\cup, \bowtie\}, \text{brTpf})$.

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 $(\{\bowtie\}, \text{brTpf}) \preceq_e (\{\cup, \bowtie\}, \text{brTpf})$. Thus, to prove the theorem it only remains to show that $(\{\cup, \bowtie\}, \text{brTpf}) \npreceq_e (\{\bowtie\}, \text{brTpf})$. We actually prove something stronger in the following lemma.

Lemma 1. $(\{\cup\}, \text{Tpf}) \npreceq_e (\{\bowtie, \rtimes, \pi\}, \text{brTpf})$.

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

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

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

As for Theorem 2, we only need to prove that $(\{\cup, \bowtie\}, \text{brTpf}) \npreceq_e (\{\cup\}, \text{brTpf})$ which follows from the next, stronger result.

Lemma 2. $(\{\bowtie\}, \text{Tpf}) \npreceq_e (\{\cup, \pi\}, \text{brTpf})$.

The lemma follows from the fact that a $(\{\bowtie\}, \text{Tpf})$-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 $(\{\cup, \pi\}, \text{brTpf})$-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. $(\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF}) \prec_e (\{\cup, \bowtie\}, \text{BRTPF})$.

We have that $(\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF}) \preceq_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{CORESPARQL})$ because it holds that $\text{TPF} \subseteq \text{CORESPARQL}$. By combining this with Theorem 1 we obtain that $(\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF}) \preceq_e (\{\cup, \bowtie\}, \text{BRTPF})$. Thus, to prove Theorem 4 it remains to show that $(\{\cup, \bowtie\}, \text{BRTPF}) \not\preceq_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF})$. We prove something stronger:

Lemma 3. $(\emptyset, \text{BRTPF}) \not\preceq_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF})$.

It turns out that FILTER is all that one should add to TPF to make it comparable with BRTPF. In fact, in terms of expressive power of LDFMs, TPF and BRTPF are equivalent regardless of the client language.

Proposition 1. $(\mathcal{L}, \text{BRTPF}) \equiv_e (\mathcal{L}, \text{TPF} + \text{FILTER})$ *for every client language* $\mathcal{L}$.

Given Proposition 1, in every combination in the lattice of Figure 3 we can replace BRTPF with 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 follows from our previous results.

Corollary 1. *The following relationships hold.*

1. $(\{\bowtie, \cup, \rtimes, \pi\}, \text{TPF})$ *and* $(\emptyset, \text{BRTPF})$ *are not comparable.*
2. $(\{\cup\}, \text{TPF})$ *and* $(\{\bowtie\}, \text{BRTPF})$ *are not comparable.*
3. $(\{\bowtie\}, \text{TPF})$ *and* $(\{\cup\}, \text{BRTPF})$ *are not comparable.*

The lattice of the expressiveness of LDFMs shown in Figure 3 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 client 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 [13]. 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* $\mathcal{L}'_1$*.*

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 Schmidt et al. [12].

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

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

Moreover, for any server language $\mathcal{L}_S$*, the following properties hold:*

3. $(\emptyset, \mathcal{L}_S) \equiv_{rc} (\{\bowtie\}, \mathcal{L}_S) \equiv_{rc} (\{\cup\}, \mathcal{L}_S)$
4. $(\emptyset, \mathcal{L}_S) \preceq_{rc} (\{\bowtie, \cup\}, \mathcal{L}_S) \preceq_{rc} (\{\leftouterjoin\}, \mathcal{L}_S)$
5. $(\{\leftouterjoin\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, \leftouterjoin\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, \leftouterjoin, \cup\}, \mathcal{L}_S) \equiv_{rc} (\{\bowtie, \leftouterjoin, \cup, \pi\}, \mathcal{L}_S)$

Notice that the pairs of client and server languages mentioned in the corollary can be organized into two additional lattices along the lines of the expressiveness lattice in Figure 3. More specifically, Properties 1 and 2 in Corollary 2 establish a *server-power demand lattice*, and Properties 3–5 establish a *result-construction 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 client 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 1. Let $(\mathcal{L}_1, \mathcal{L}_2)$ and $(\mathcal{L}'_1, \mathcal{L}'_2)$ be arbitrary pairs of client-server languages such that $\mathcal{L}_1 \subseteq \mathcal{L}'_1$ and $\mathcal{L}_2 \subseteq \mathcal{L}'_2$. Then, 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 $(\emptyset, \textsc{Bgp})$, $(\{\bowtie\}, \textsc{Tpf})$, and $(\{\bowtie\}, \textsc{Bgp})$ 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 $(\{\bowtie\}, \textsc{Bgp}) \prec_t (\{\bowtie\}, \textsc{Tpf})$. Moreover, $(\{\bowtie\}, \textsc{Tpf})$ and $(\emptyset, \textsc{Bgp})$ are not comparable in terms of $\preceq_t$. Regarding the number of requests it holds that $(\{\bowtie\}, \textsc{Bgp}) \equiv_r (\emptyset, \textsc{Bgp}) \prec_r (\{\bowtie\}, \textsc{Tpf})$.*

To see why the $\preceq_t$ incomparability result holds, consider the class $\mathcal{C}_1$ of SPARQL queries of the form $((?X_1, ?Y_1, ?Z_1)\ \textsc{And}\ (?X_2, ?Y_2, ?Z_2))$. 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 $((a_1, b_1, c_1)\ \textsc{And}\ \cdots\ \textsc{And}\ (a_k, b_k, c_k))$. 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 client language.

Theorem 5. *The following strict relationships hold.*

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

The first point of this last theorem can be intuitively read as follows: in terms of bandwidth, the best possible query plan for an LDFM that accesses a brTpf interface and then constructs the output using operators in $\{\cup, \bowtie, \rtimes, \pi\}$, is strictly better than the best possible query plan that accesses a brTpf interface and then constructs the output using operators in $\{\cup, \bowtie\}$. The second point has a similar interpretation regarding the best possible query plan 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 Section 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 URI-lookup interface that we capture by the language UriF in Section 2.1.

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. In particular, we have recently proposed the LDQL language [9] , which is a navigational language designed to query Semantic Web data based on the URIF interface. Although we have presented a fairly complete formal analysis of LDQL [9], 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.

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 Record 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: ISWC (2014)
4. Berners-Lee, T.: Design issues: Linked Data. `https://www.w3.org/DesignIssues/LinkedData.html` (July 2006)
5. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst. 5(3), 1–22 (2009)
6. DBpedia. `http://dbpedia.org/`
7. Hartig, O.: SPARQL for a Web of Linked Data: Semantics and Computability. In: Proceedings of the 9th Extended Semantic Web Conference (ESWC) (2012)
8. Hartig, O., Buil-Aranda, C.: Bindings-Restricted Triple Pattern Fragments. In: Proc. of the 15th Int. Conf. on Ontologies, Databases, and Applications of Semantics (ODBASE) (2016)
9. Hartig, O., Pérez, J.: LDQL: A Query Language for the Web of Linked Data. J. Web Sem. 41, 9–29 (2016)
10. Mendelzon, A.O., Milo, T.: Formal Models of Web Queries. Inf. Syst. 23(8), 615–637 (1998)
11. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst. 34(3), 16:1–16:45 (2009)
12. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL Query Optimization. In: Proceedings of the 13th International Conference on Database Theory (ICDT) (2010)
13. Vardi, M.Y.: The Complexity of Relational Query Languages. In: STOC (1982)
14. 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 Sem. 37-38, 184–206 (2016)

A Proofs of Section 3

A.1 Proof of Theorem 1

We first prove that $(\{\bowtie, \cup, \ltimes\!\!\!\!\bowtie, \pi\}, \text{CORESPARQL}) \preceq_e (\{\bowtie, \cup\}, \text{BRTPF})$. Let M be a $(\{\bowtie, \cup, \ltimes\!\!\!\!\bowtie, \pi\}, \text{CORESPARQL})$-LDFM, and assume that M receives as input (q, G). We will construct a $(\{\bowtie, \cup\}, \text{BRTPF})$-LDFM that simulates M, such that $\bar{M}(q, G) = M(q, G)$. Recall that M works by sending some requests to the server, receiving them in result containers $D_1, D_2, \dots, D_{c_M}$ and finally constructs a query over them using $\{\bowtie, \cup, \ltimes\!\!\!\!\bowtie, \pi\}$ that the client oracle computes in the output tape. We construct $\bar{M}$ with the following idea: $\bar{M}$ will mimic the complete process done by M by using its unbounded local computational power plus BRTPF calls to the server, and then reconstructing the output of M using $\{\cup, \bowtie\}$.

$\bar{M}$ works initially as follows. It first constructs the BRTPF query $(?s, ?p, ?o)$, sends it to the server and stores its result in D_1. Notice that $D_1 = \{\nu \mid (\nu(?s), \nu(?p), \nu(?o)) \in G\}$, and thus D_1 essentially stores the complete graph G. After this, $\bar{M}$ uses its unbounded computational power to mimic the full computation of M including the calls that M send to the server oracle but considering D_1 as the input graph and using only its internal working tapes. Finally, $\bar{M}$ mimics the call that M sends to its client oracle to compute $M(q, G)$. Notice that after all this process $\bar{M}$ has the complete output of M in one of its working tapes. We next show how $\bar{M}$ can use it to produce this very same output in its own output tape.

We first show the special case in which $M(q, G) = \{\mu\}$, that is the case in which M produces a single mapping as output. Let $\text{dom}(\mu) = \{?x_1, ?x_2, .., ?x_\ell\}$ and assume that $\mu(?x_i) = k_i$ for $i \in \{1, \dots, \ell\}$. Recall that $\bar{M}$ has the set $\{\mu\}$ in one of its working tapes, and has the complete graph G in D_1. Now for every $i \in \{1, \dots, \ell\}$ our LDFM $\bar{M}$ does the following. For every $\{?s \to a, ?p \to b, ?o \to c\} \in D_1$, $\bar{M}$ sends the BRTPF queries $((?x_i, b, c), \{?x_i \to k_i\})$, $((a, ?x_i, c), \{?x_i \to k_i\})$ and $((a, b, ?x_i), \{?x_i \to k_i\})$ to the server and stores the respective results in its result containers. We claim that it cannot be the case that the result of all these queries is empty. On the contrary assume that all are empty. Then there is not a single triple in G containing the value k_i which contradicts the fact that value k_i is in the domain of one of the mappings in $M(q, G)$. Thus we know that there is at least one of the result containers that is not empty after all these calls. Lets call D^i to an arbitrary of the non empty result mappings. Notice that D^i has a single mapping, lets call it ν^i. Moreover, ν^i is a mapping such that $\nu^i(?x_i) = k_i$ and $\text{dom}(\nu^i) = ?x_i$. Finally, $\bar{M}$ can construct the client query $q_C = D^1 \bowtie D^2 \bowtie \cdots \bowtie D^\ell$ which can be evaluated by the client oracle to give exactly a set with a single mapping ν such that $\text{dom}(\nu) = \{?x_1, ?x_2, \dots, ?x_\ell\}$ and such that $\nu(?x_i) = k_i$ for $i \in \{1, \dots, \ell\}$. Thus, we have that $\nu = \mu$ which implies that $\bar{M}(q, G) = \{\mu\} = M(q, G)$.

Now assume that $M(q, G) = \{\mu^1, \mu^2, \dots, \mu^p\}$. From the strategy shown above $\bar{M}$ can produce a result container $\bar{D}^i = \{\mu^i\}$ for each $i \in \{1, \dots, p\}$. Then $\bar{M}$ produces the client query $\bar{D}^1 \cup \bar{D}^2 \cup \cdots \cup \bar{D}^p$. By the discussion above, it is straightforward to prove that the evaluation of this query is exactly $M(q, G)$, which completes our proof.

We still need to prove that $(\{\bowtie, \cup\}, \text{BRTPF}) \preceq_e (\{\bowtie, \cup, \ltimes\!\!\!\!\bowtie, \pi\}, \text{CORESPARQL})$. This is very easy by using Proposition **??** that we prove later in the appendix. From that proposition we obtain that $(\{\bowtie, \cup\}, \text{BRTPF}) \preceq_e (\{\bowtie, \cup\}, \text{TPF} + \text{FILTER})$. Moreover, since $\{\bowtie, \cup\} \subseteq \{\bowtie, \cup, \rtimes, \pi\}$ and $\text{TPF} + \text{FILTER} \subseteq \text{CORESPARQL}$ we obtain that $(\{\bowtie, \cup\}, \text{TPF} + \text{FILTER}) \preceq_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{CORESPARQL})$. Thus composing $(\{\bowtie, \cup\}, \text{BRTPF}) \preceq_e (\{\bowtie, \cup\}, \text{TPF} + \text{FILTER})$ and $(\{\bowtie, \cup\}, \text{TPF} + \text{FILTER}) \preceq_e (\{\bowtie, \cup, \rtimes, \pi\}, \text{CORESPARQL})$ we obtain the property that we needed to prove.

A.2 Proof of Lemma 1

Consider the CORESPARQL query $q = ((?X, a, 2) \text{ UNION } (3, b, 4))$. It is clear that q is computable by a $(\{\cup\}, \text{TPF})$-LDFM by sending two requests to the server and then combining the result with $\cup$. We next prove that q is not computable by a $(\{\bowtie, \rtimes, \pi\}, \text{BRTPF})$-LDFM. In particular, let $G = \{(1, a, 2), (3, b, 4)\}$. Notice that $[\![q]\!]_G = \{\mu_1, \mu_2\}$, where μ_1 is a mapping such that $\text{dom}(\mu) = \{?X\}$ and $\mu(?X) = 1$, and μ_2 is the empty mapping (that is, $\text{dom}(\mu_2) = \emptyset$). Let M be an arbitrary $(\{\bowtie, \rtimes, \pi\}, \text{BRTPF})$-LDFM and assume that $M(q, G) = [\![q]\!]_G$. One can prove that no matter how M works, its output $M(q, G)$ satisfies the following property:

Claim: if a mapping in $M(q, G)$ has variable $?X$ in its domain, then every mapping in $M(q, G)$ has $?X$ in its domain.

Notice that this is a contradiction with the assumption that $M(q, G) = [\![q]\!]_G$, which completes our proof.

We now prove the above claim. Recall $M(q, G)$ is the resault of the evaluation of a query q_C by $\mathcal{O}_C$. So we will prove our claim by induction on the numbers of operations that make q_C. First if $q_C = D_i$, then $M(q, G)$ it's equivalent to an answer containers that comes of the evaluation of q_i by $\mathcal{O}_S$. Then if there is $\mu \in M(q, G)$ such that $?x \in dom(\mu) = var(q_i)$ then for every $\nu \in \bar{D}$ we have that $?x \in var(q_i) = dom(\nu)$. We have then the base case.

Now for the induction step. For this will have to work every posible operation applied on q_C. First if $q_C = \pi_V(d)$. Well if there is $\mu \in M(q, G)$ such that $?x \in dom(\mu)$ that means two things: first there is a mappings $\mu_i \in d$ such that $?x \in dom(\mu_i)$ and that $?x \in V$. Well by induction step then for every $\nu_i \in d$ we have that $?x \in dom(\nu_i)$. But then, since $?x \in V$ we have that for all $\nu \in M(q, G)$ we have that $?x \in dom(\nu)$.

Now if $q_C = d_1 \bowtie d_2$. If there is $\mu \in M(q, G)$ such that $?x \in dom(\mu)$ then there is two compatibles mappings $\mu_1 \in d_1, \mu_2 \in d_2$ such that $\mu = \mu_1 \cup \mu_2$. Well then since $dom(\mu) = dom(\mu_1) \cup dom(\mu_2)$, without loss of generality we have that $?x \in dom(\mu_1)$. Well then we have proved that there is a mapping $\mu_1 \in D_1$ such that $?x \in dom(\mu_1)$. By induction the for every mapping $\nu_1 \in D_1$ we have that $?x \in dom(\nu_1)$. Now let $\eta \in M(q, G)$ be any mapping. Then there is two compatible mappings $\eta_1 \in d_1, \eta_2 \in d_2$ such that $\eta = \eta_1 \cup \eta_2$. But then, since $\eta_1 \in d_1$ we have that $?x \in dom(\eta_1) \subset dom(\eta)$.

Finally if $q_C = d_1 \rtimes d_2 = (d_1 \bowtie d_2) \cup (d_1 \setminus d_2)$. Now let be $\mu \in M(q, G)$ such that $?x \in dom(\mu)$. There is two cases, first if $\mu \in d_1 \bowtie d_2$ we fall on the later case and then for every $\nu_1 \in d_1$ we have that $?x \in dom(\nu_1)$ and that for every $\eta \in d_1 \bowtie d_2$ we have that $?x \in dom(\eta)$. Well then, for every $\nu \in M(q, G)$ we have that $d_1 \bowtie d_2$ or that

$\nu \in d_1 \setminus d_2 \subset d_1$; and in any case we have that $?x \in dom(\nu)$. Now if $\mu \in d_1 \setminus d_2$, we have that $\mu \in d_1$. Then by induction step we have that for every $\nu_1 \in d_1$ we have that $?x \in dom(\nu_1)$. Then on the same manner as before we can deduce that for all $\eta \in M(q, G)$ we have that $?x \in dom(\eta)$. Now we can conclude like before.

A.3 Proof of Lemma 2

We need to prove that $(\{\bowtie\}, \text{TPF}) \not\preceq_e (\{\cup, \pi\}, \text{BRTPF})$. Let M be a $(\{\cup, \pi\}, \text{BRTPF})$-LDFM. We first prove the following claim.

Claim: for every query q and every graph G it holds that if there is a mapping $\mu \in M(q, G)$ then $|\operatorname{dom}(\mu)| \leq 3$.

To see this, notice that if D is a result container during the computation of M, then D is a set of solution mappings obtained from executing a BRTPF query. Then, it is clear that if $\mu \in D$ we have that μ is defined in at most three variables and then it holds that $|\operatorname{dom}(\mu)| \leq 3$. Now assume that M produces an output query q_O using operators $\{\cup, \pi\}$. It is not difficult to see that q_O is equivalent to a query of the form

$$\pi_{S_1}(T_1) \cup \pi_{S_2}(T_2) \cup \cdots \cup \pi_{S_k}(T_k)$$

where every T_i is one of M's result containers. Thus, if $\mu \in M(q, G)$ then $\mu \in \pi_{S_i}(T_i) = \pi_{S_i}(D_j)$ for some j, and thus given that every mapping in D_j is defined in at most 3 variables, we have that $|\operatorname{dom}(\mu)| \leq 3$ which is the property that we wanted to show.

Consider now the query $q = ((?X, ?Y, ?Z) \text{ AND } (?U, ?V, ?W))$ which is clearly computable by a $(\{\bowtie\}, \text{TPF})$-LDFM. Moreover, given the above claim it is easy to conclude that q is not computable by a $(\{\cup, \pi\}, \text{BRTPF})$-LDFM, since there are graphs for which q produces mappings with more that 3 variables in its domain. Thus we prove that there exists a query that is computable under $(\{\bowtie\}, \text{TPF})$ but not under $(\{\cup, \pi\}, \text{BRTPF})$ which shows that $(\{\bowtie\}, \text{TPF}) \not\preceq_e (\{\cup, \pi\}, \text{BRTPF})$.

A.4 Proof of Lemma 3

Let μ_1 be a mapping such that $\operatorname{dom}(\mu_1) = \{?X\}$ and $\mu_1(?X) = 1$. Consider a query q such that for every G it holds that $[\![q]\!]_G = \{\mu_1\}$ if $(1, a, 1) \in G$ and $[\![q]\!]_G = \emptyset$ if $(1, a, 1) \notin G$. It is clear that q can be computed by a $(\emptyset, \text{BRTPF})$-LDFM with just one request to server of the form $((?X, a, ?X), \{\{?X \rightarrow 1\}\})$. Now consider the graph $G' = \{(1, a, 1), (2, a, 1), (1, a, 2), (2, a, 2)\}$. Notice that $[\![q]\!]_{G'} = \{\mu_1\}$. Let M be an arbitrary $(\{\bowtie, \cup, \ltimes\!\!\!\!\bowtie, \pi\}, \text{TPF})$-LDFM and assume that $M(q, G') = [\![q]\!]_{G'}$. One can prove that no matter how M works, its output $M(q, G')$ satisfies the following property:

Claim: if there is a mapping μ in $M(q, G')$ such that $\mu(?X) = 1$, then there exists a mapping $\mu' \in M(q, G')$ such that $\mu'(?X) = 2$, and moreover, for every pair of mappings $\nu, \nu' \in M(q, G')$ and every variable $?Y$ such that $?Y \neq ?X$, it holds that $\nu(?Y) = \nu'(?Y)$.

Thus, given that $\mu_1 \in M(q, G')$ we obtain that $\mu_2 \in M(q, G')$ where μ_2 is a mapping such that $\mathrm{dom}(\mu_2) = \{?X\}$ and $\mu_2(?X) = 2$, which contradicts the fact that $M(q, G') = [\![q]\!]_{G'}$.

We now prove the above claim. Remember that $M(q, G')$ comes from the evaluation of q_C by $\mathcal{O}_C$. So we will prove our claim by induction on numbers of operations that make q_C. On the base case $q_C = D_i$, then $M(q, G')$ is actually equivalent to the a answer container of a query q_i sent to the server. First we will prove it if $|var(q_i)| = 1$, on that case the mappings must be $\nu = \mu_1$ and $\nu' = \mu_2$. Note that the left implies right part comes from the fact that the only q_i as above are $(?x, a, 1), (1, a, ?x), (?x, a, 2), (2, a, ?x)$, which fulfills our claim. Now for the other one, suppose q_i has μ_2 but no μ_1. Then the machine can have a register that have μ_1 but not μ_2. Indeed, it can just send $h(q_i)$, where h is the homomorphism that changes 1 and 2. Note that G' is invariant under h, so the answer of $h(q_i)$ must be the same as q_i but changing 1 with 2, this is a contradiction with what we proved. So we are done on this case, now if $|var(q_i)| = 2$, the only query with answer that include a mapping that send $?x$ to 1 are $(?x, a, 1)$,$(?x, a, 2)$,$(?x, ?s, 1)$,$(?x, ?s, 2)$ and the queries made changing the last component with the first one. Note that all those query have answers that goes along our claim. Finally if $|var(q_i)| = 3$ that means it can only be $(?x, ?s, ?y)$ or $(?y, ?s, ?x)$ who also fulfils our property. We have the base case.

Now the induction step; we will prove only the if, cause the only if follow by symmetry of our proposition. If $q_C = d_1 \cup d_2$, and let be $\nu \in M(q, G')$. Well then, w.l.g., $\nu \in d_1$. From our induction hypotheses then this happens if and only if it's exists the mappings $\nu' \in d_1$ and then $\nu' \in M(q, G')$.

Now if $q_C = d_1 \bowtie d_2$ and let be $\nu \in M(q, G')$ that means there are two compatible mapping $f_1 \in d_1, f_2 \in d_2$ such that $\nu = f_1 \cup f_2$. First suppose $?x \in dom(f_1) \cap dom(f_2)$. Well since $\nu(?x) = 1$ this implies $f_1(?x) = f_2(?x) = 1$. By induction hypothesis then there exists mapping $f_1' \in d_1, f_2' \in d_2$ that are equal to f_1 and f_2 respectively, except on $?x$ where $f_1'(?x) = f_2'(?x) = 2$. Note these are clearly compatible. Now note that $\nu' = f_1' \cup f_2' \in d_1 \bowtie d_2$ is our mapping. Indeed, clearly $\nu'(?x) = 2$, and by definition, for $?y \in dom(\nu') \setminus \{?x\}$ there exists i such that $\nu'(?y) = f_i'(?y) = f_i(?y) = \nu(?y)$. Now lets rewind and suppose, w.l.g., that $?x \notin dom(f_2)$. Well in that case we use the induction hypothesis on f_1 and get a mapping f_1' with the same properties that on the latest case. Notice that f_1' and f_2 are compatible, since f_1 and f_2 are and f_1' is only different from f_1 on a variable that is not on f_2 domains. We then can define $\nu' = f_1' \cup f_2 \in d_1 \bowtie d_2$, which is our mapping. Indeed, $\nu'(?x) = f_1'(?x) = 2$, and by definition for $?y \in dom(\nu') \setminus \{?x\}$ we will have two cases. If $?y \in dom(f_1')$ well then $\nu'(?y) = f_1'(?y) = f_1(?y) = \nu(?y)$ and if $y \in dom(f_2)$ then $\nu'(?y) = f_2(?y) = \nu(?y)$. Then we have this case.

Now if $q_C = d_1 ⟕ d_2 = (d_1 \bowtie d_2) \cup (d_1 \setminus d_2)$, and let be $\nu \in M(q, G')$. Note that if $\nu \in d_1 \bowtie d_2$ we fall on the latest case, so let's suppose $\nu \in d_1 \setminus d_2$. Well that means $\nu \in d_1$ and for all $f \in d_2$ we have that ν_1 and f are not compatible. From the first part and our induction hypothesis we then have that $\nu' \in d_1$.

We have to show that $\nu' \in d_1 \setminus d_2$. Well if that's not the case there exists a function $f' \in d_2$ such that ν' and f' are compatible, that means that $f'(?x) = 2$. By our induction hypothesis we then have there is a mapping $g \in d_2$ that $g(?x) = 1$ and for all $?y \in dom(f') \setminus \{?x\}$ we have $g(?y) = f'(?y)$. If we show that g and ν are compatible we arrive at a contradiction, then we must have $\nu' \in d_1 \setminus d_2$ and that finish the proof. Indeed note that $g(?x) = \nu(?x) = 1$, and $\forall ?y \in dom(\nu) \setminus \{?x\}$ we have that $\nu'(?y) = \nu(?y)$. But then $\nu'(?y)$ was compatible with f'; we have two cases then. If $?y \in dom(f')$ then $\nu'(?y) = f'(?y)$, but by definition of g we then have that $\nu(?y) = \nu'(?y) = f'(?y) = g(?y)$. Now if $?y \notin dom(f')$ then $?y \notin dom(g)$. Well then $\forall ?z \in dom(\nu) \cap dom(g)$ we have $g(?z) = \nu(?z)$. And so g and ν are compatible.

Finally if $q_C = \pi_V(d_1)$. If $\nu \in M(q, G')$ that means there is a mapping μ such that $mu(?y) = \nu(?y)$ for all $?y \in V$. If $?x \notin V$ we have our property, so let's work when $?x \in V$ and $\nu(?x) = 1$. By our induction hypothesis then there is a mapping μ' such that $\mu'(?x) = 2$ and on all other variables it is equal to μ. We will show that $\nu' = \Pi_V(\mu') \in \bar{D}$ is the mapping we are searching for. Indeed, $?x \in V$, so $\nu'(?x) = \mu'(?x) = 2$. Also, $\forall ?y \in dom(\nu) \setminus \{?x\} = V \setminus \{?x\}$ we have that $\nu(?y) = \mu(?y) = \mu'(?y) = \nu'(?y)$. This finishes our proof.

A.5 Proof of Proposition 1

We first prove that $(\mathcal{L}, \text{BRTPF}) \preceq_e (\mathcal{L}, \text{TPF} + \text{FILTER})$ for every $\mathcal{L}$. Let D be a result container of a $(\mathcal{L}, \text{BRTPF})$-LDFM and assume that D stores the result of (t, Ω). Then, this request can be obtained by using the following server request in a $(\mathcal{L}, \text{TPF} + \text{FILTER})$ machine. Just consider a query of the form

$$q_{(t,\Omega)} \;=\; t \text{ FILTER} \left(\bigvee_{\nu \in \Omega} \left(\bigwedge_{?X \in \mathrm{dom}(\nu) \cap \mathrm{vars}(t)} ?X = \nu(?X) \right) \right).$$

We next show that $[\![q_{(t,\Omega)}]\!]_G = [\![(t,\Omega)]\!]_G$. Let $\mu \in [\![q_{(t,\Omega)}]\!]_G$ then $\mu \in [\![t]\!]_G$ and $\mu \models \alpha$ where α is the FILTER expression. Thus in particular $\mu \models \alpha_\nu$ for a $\nu \in \Omega$. That is, for every $?X \in \mathrm{dom}(\nu) \cap \mathrm{vars}(t)$ we have that $\mu \models (?X = \nu(?X))$ which implies that $\mu(?X) = \nu(?X)$, and then μ and ν are compatibles. Notice that this implies that $\mu \cup \nu \in [\![t]\!]_G \bowtie \Omega$, and since $\mathrm{dom}(\mu) \subseteq \mathrm{vars}(t)$ we have that $\mu \in \pi_{\mathrm{vars}(t)}([\![t]\!]_G \bowtie \Omega)$ and thus $\mu \in [\![(t,\Omega)]\!]_G$. For the other direction, assume that $\mu \in [\![(t,\Omega)]\!]_G$. Thus $\mu \in \pi_{\mathrm{vars}(t)}([\![t]\!]_G \bowtie \Omega)$ which implies that there exists $\mu' \in [\![t]\!]_G$ and $\nu \in \Omega$ such that μ' and ν are compatible and $\mu = (\mu' \cup \nu)|_{\mathrm{vars}(t)}$. Given that $\mathrm{dom}(\mu') = \mathrm{vars}(t)$ we have that $\mu = \mu' \in [\![t]\!]_G$. Moreover, given that $\mu' = \mu$ and ν are compatible, we have that $\mu(?X) = \nu(?X)$ for all $?X \in \mathrm{dom}(\nu) \cap \mathrm{vars}(t)$ (since $\mathrm{vars}(t) = \mathrm{dom}(\mu)$), and thus $\mu \models \alpha_\nu$ which implies that $\mu \in [\![q_{(t,\Omega)}]\!]_G$. This is enough to prove that every $(\mathcal{L}, \text{BRTPF})$-LDFM can be simulated with a $(\mathcal{L}, \text{TPF} + \text{FILTER})$-LDFM.

We now prove the opposite direction. Let D be a result container of a $(\mathcal{L}, \text{TPF} + \text{FILTER})$-LDFM and assume that D stores the result of $(t \text{ FILTER } \theta)$ when it is executed with input graph G. We next show how a $(\mathcal{L}, \text{BRTPF})$-LDFM M can use two requests to the server to obtain the same result D in one of its result containers. M first constructs a BRTPF request $(t, \{\mu_\emptyset\})$ where $\mu_\emptyset$ is the empty mapping and stores it in a result container, say D'. Notice that D' contains $[\![t]\!]_G$. After that, M uses its internal computational power to construct the set of mappings $\Omega = \{\mu \in D' \mid \mu \models \theta\}$, and sends to the server the BRTPF request (t, Ω). Given that $D' = [\![t]\!]_G$ then Ω is exactly $[\![(t\,\text{FILTER}\,\theta)]\!]$. Thus $[\![(t, \Omega)]\!]_G = [\![t]\!]_G \bowtie [\![(t\,\text{FILTER}\,\theta)]\!]_G = [\![(t\,\text{FILTER}\,\theta)]\!]_G$, and thus M can produce exactly D as one of its result containers. This is enough to prove that every $(\mathcal{L}, \text{TPF} + \text{FILTER})$-LDFM can be simulated with a $(\mathcal{L}, \text{BRTPF})$-LDFM.

A.6 Proof of Proposition 2

First let see that:

$$(\emptyset, \text{BGP}) \prec_e (\{\bowtie\}, \text{TPF})$$

Let $\mathcal{C}$ be a class expresable on $(\emptyset, \text{BGP})$. We will show that it is expresable under $(\{\bowtie\}, \text{TPF})$. Let M be the $(\emptyset, \text{BGP})$-LDFM machine such that $\mathcal{C} = \mathcal{C}(M)$. We will make a $(\{\bowtie\}, \text{TPF})$-LDFM machine $\bar{M}$ such that $M(q, G) = \bar{M}(q, G)$, for ever graph G and $q \in \mathcal{C}$.

For this first remember that M do some computation and then send some query q_1 on BGP language to the server. Our machine $\bar{M}$ just have to see a way around the fact that q_1 have AND between its triple patterns. So for that the idea is to decompose the query q_1 on its triplets that are separated by the AND. Then we send each one of the separated to the server and we get answer containers of the triple patterns. Then we recompose the answer of the query; if we need to continue computing it's easy to see $\bar{M}$ could do it on it's working tape, and if we need it to evaluate q on G we do it with $\bowtie$ with the help of q_C.

Let us formalize the last idea. So M work doing some sort of computation and send queries of the form $q_j = (t_1 \wedge t_2 \wedge ... \wedge t_n)$ to the server, that response on some answer container D_j.After that, since M doesn't have any language on the client side it have to select, someway, a register D_m and send it as a query to the client. Now to simulate this on $\bar{M}$ instead of sending q_j to the server first we write q_j on our workband and we send each t_k of the query separated to the server, this gives responses $\bar{D}_j^k$. With this we can obtaining the registers D_j on the workband, $\bar{M}$ can do the computation itself. Now as a query to the client we send $\bar{D}_m^1 \bowtie \bar{D}_m^2 \bowtie ... \bowtie \bar{D}_m^n$. For this to work we just have to show that $\bar{D}_m^1 \bowtie \bar{D}_m^2 \bowtie ... \bowtie \bar{D}_m^n = D_m$.

Indeed, consider a mapping $\mu \in D_m$. So this mapping is an answer to the query $q_m = (t_1\,\text{AND}\,t_2\,\text{AND}\,...\,\text{AND}\,t_n)$. So note that for every triple t_k we can define the restriction μ_{t_k} as that $dom(\mu_{t_k}) = var(t_k)$ and $\forall ?x \in var(t_k), \mu_{t_k}(?x) = \mu(?x)$. So note that if $k \neq l$ then μ_{t_k} and μ_{t_l} are compatible. Indeed, if $?x \in dom(\mu_{t_k}) \cap dom(\mu_{t_j})$ then, by construction, $\mu_{t_k}(?x) = \mu(?x) = \mu_{t_l}(?x)$. Note that by definition $\mu_{t_k}(t_k) = \mu(t_k)$. Also note that $\mu_{t_k} \in \bar{D}_m^k$, indeed, if $\mu \in D_m$ that means that $\mu(q_m) \in G$, which means that $\mu(t_k) = \mu_{t_k}(t_k) \in G, \forall k$, which means that $\mu_{t_k} \in \bar{D}_m^k$. So then, note that $\mu = \mu_{t_1} \cup \mu_{t_2} \cup ... \cup \mu_{t_n}$, where $(\mu_{t_i})_{i=1}^n$ are compatible. In fact, we already show that they are compatible pairwise, which be induction leads they are compatible like a family, and by construction we have the equality of the mappings. So that means that $\mu \in \bar{D}_m^1 \bowtie \bar{D}_m^2 \bowtie ... \bowtie \bar{D}_m^n$.

Now let be $\mu \in \bar{D}_m^1 \bowtie \bar{D}_m^2 \bowtie ... \bowtie \bar{D}_m^n$, that means there are compatible mapping such that $\mu_{t_1} \cup \mu_{t_2} \cup ... \cup \mu_{t_m} = \mu$ and $(\mu_{t_i})_{i=1}^n$ are compatibles. By definition of $\bar{D}_m^i$ that means that $\mu_{t_i}(t_i) \in G, \forall i = 1, .., n$. However that implies that $\mu(q_m) \in G$ since the mappings are compatibles. But that mean that $\mu \in D_m$, which finishes the proof.

Now note that since $\text{TPF} \subset \text{BGP}$ we already have $(\{\bowtie\}, \text{TPF}) \preceq_e (\{\bowtie\}, \text{BGP})$. So we have just have to prove that $(\{\bowtie\}, \text{BGP}) \preceq_e (\emptyset, \text{BGP})$.

So let $\mathcal{C}$ an expresable class of queries under $(\{\bowtie\}, \text{BGP})$ with a machine M. We will construct a machine $\bar{M}$ that express it under $(\emptyset, \text{BGP})$ as it follows.

M works doing some computation, send queries q_j to the server, which send D_j as an answer. Finally it selects come registers and send $B_1 \wedge B_2, .., \wedge B_m$ to the client, where B_i is some register on the machine. Our machine $\bar{M}$ will do the same, but also will store which query was sent for each register. Then when M selects the registers to send them to the client united by a joint, our new machine $\bar{M}$ will select the send registers but wil send them united by a $\wedge$ to the server, which will give some answer $\bar{B}$ and that will be our answer to the client.

Look that, like our last proof, all we need for that to work is that $\bar{B} = B_1 \bowtie B_2, ..., \bowtie B_m$, which is proven like the last case. We conclude the equivalence.

B Proofs of Section 5

B.1 Proof of Proposition 3

Consider the class $\mathcal{C}_1$ of queries of the form $((?X_1, ?Y_1, ?Z_1)$ AND $(?X_2, ?Y_2, ?Z_2))$. It can be shown that any $(\emptyset, \text{BGP})$-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 $(\{\bowtie\}, \text{TPF})$-LDFM M' such that $t_{M'}(q, G)$ is in $O(|G|)$. This shows that $(\emptyset, \text{BGP}) \not\preceq_t (\{\bowtie\}, \text{TPF})$. Consider now the class $\mathcal{C}_2$ of SPARQL queries of the form $((a_1, b_1, c_1)$ AND $\cdots$ AND $(a_k, b_k, c_k))$, With all a_i's, b_i's and c_i's distinct values. One can show that any $(\{\bowtie\}, \text{TPF})$-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 $(\emptyset, \text{BGP})$-LDFM that, in the worst case, transfers a single mapping (the complete query result) from server to client thus showing that $(\{\bowtie\}, \text{TPF}) \not\preceq_t (\emptyset, \text{BGP})$. Notice that this last fact implies that $(\{\bowtie\}, \text{TPF}) \not\preceq_t (\{\bowtie\}, \text{BGP})$. This last property plus the fact that $(\{\bowtie\}, \text{BGP}) \preceq_t (\{\bowtie\}, \text{TPF})$ implies that $(\{\bowtie\}, \text{BGP}) \prec_t (\{\bowtie\}, \text{TPF})$. Class $\mathcal{C}_2$ can also be used to show that $(\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{TPF})$.

Consider the class $\mathcal{C}_1$ of queries of the form $((?X_1, ?Y_1, ?Z_1)$ AND $(?X_2, ?Y_2, ?Z_2))$. It can be shown that any $(\emptyset, \text{BGP})$-LDFM M that computes $\mathcal{C}_1$ is such that $t_M(q, G)$, as a function, is in $\Omega(|G|^2)$. Indeed, note that since M doesn't have any algebraic operation on the client size $M(q, G)$ must be equivalent to an answer container, let's call it $\bar{D}$. Also note, from the query q, it follows that any μ in $M(q, G)$ must fulfill $|\operatorname{dom}(\mu)| = 6$. So that means that whatever query $\bar{q}$, M sent to the server to obtaining $\bar{D}$ we know that $\bar{q}$ had at least two triple patterns with distinct variable. It's easy to see that kind of query make us have $\Omega(|G|^2)$ data transfer.

On the other hand there exists a $(\{\bowtie\}, \text{TPF})$-LDFM M' such that $t_{M'}(q, G) = 2|G|$. Indeed M' work sending $q_1 = (?X_1, ?Y_1, ?Z_1)$ and then $q_2 = (?X_2, ?Y_2, ?Z_2)$ to the client, obtaining answer container D_1 and D_2 respectively and then sending the query $D_1 \bowtie D_2$ to the client. It easy to see that $M'(q, G)$ and that $|D_1| = |D_2| = |G|$ and so M' works with $2|G|$ transfer. This shows that $(\emptyset, \text{BGP}) \not\preceq_t (\{\bowtie\}, \text{TPF})$.

Consider now the class $\mathcal{C}_2$ of SPARQL queries of the form $((a_1, b_1, c_1)\ \text{AND} \cdots \text{AND}\ (a_k, b_k, c_k))$, With all a_i's, b_i's and c_i's distinct values. One can show that any $(\{\bowtie\}, \text{TPF})$-LDFM M that computes $\mathcal{C}_2$ is such that $t_M(q, G)$ is in $\Omega(|q|)$. Indeed suppose there is a $(\{\bowtie\}, \text{TPF})$-LDFM M that can do better than that. It's easy to see two things; first M can't send a triple pattern that contains a variable. Cause with a graph large enough we could easily surpass $\Omega(|q|)$. Also M can't send all the triple pattern of every q, cause that would make it work with $\Omega(|q|)$ transfer. So there is at least one triple of some q_0 that M doesn't send. Let's say, without loss of generality, it's (a_1, b_1, c_1) (the first triple of q_0) and consider the graphs $G = \cup_{i=1}^{|q_0|}\{(a_i, b_i, c_i\}$ and $G' = \cup_{i=2}^{|q_0|}\{(a_i, b_i, c_i\}$. Note that since M only send triple patterns but doesn't send the only triple where G and G' are different then $M(q_0, G) = M(q_0, G')$. However note $[\![q_0]\!]_G \neq [\![q_0]\!]_{G'}$ and so M answer G or G' incorrectly. And so it follows our claim.

On the other hand, $\mathcal{C}_2$ can be computed with a $(\emptyset, \text{BGP})$-LDFM that, in the worst case, transfers a single mapping (the complete query result) from server to client thus showing that $(\{\bowtie\}, \text{TPF}) \not\preceq_t (\emptyset, \text{BGP})$. Notice that this last fact implies that $(\{\bowtie\}, \text{TPF}) \not\preceq_t (\{\bowtie\}, \text{BGP})$. This last property plus the fact that $(\{\bowtie\}, \text{BGP}) \preceq_t (\{\bowtie\}, \text{TPF})$ implies that $(\{\bowtie\}, \text{BGP}) \prec_t (\{\bowtie\}, \text{TPF})$.

Now class $\mathcal{C}_2$ can also be used to show that $(\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{TPF})$. Indeed, we already know we can do it on one request to the server with $(\emptyset, \text{BGP})$. Suppose we could do it with the same amount of request with $(\{\bowtie\}, \text{TPF})$-LDFM M. Consider then the query in $\mathcal{C}_2$; $q = (a_1, b_1, c_1)\ \text{AND}(a_2, b_2, c_2)$ and the graph $G = \{(a_1, b_1, c_1), (a_2, b_2, c_2)\}$. Note that $[\![q_1]\!]_G = \{\mu\}$ where μ is the empty mapping. Notice that to obtain a answer container that contains μ on that graph M could only send $q_1 = (a_1, b_1, c_1)$ or $q_2 = (a_2, b_2, c_2)$ to the server. Without lose of generality suppose M sends q_1. Then note M can't send any other request, cause we are assuming M works on only one server request. So now suppose M works on $G' = \{(a_1, b_1, c_1)\}$. Then M will send as first query q_1 obtaining as an answer container $D_1 = \{\mu\}$. However $[\![q]\!]_{G'} = \emptyset$ but M can't send any query to the client that have answer $\emptyset$ having only access to D_1 and $\bowtie$ (as $D_1 \bowtie D_1 = D_1$); and so $M(q, G') \neq [\![q]\!]_{G'}$. So M can't work on only one request to the server. This shows that $(\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{TPF})$.

Now for our last claim $(\{\bowtie\}, \text{BGP}) \equiv_r (\emptyset, \text{BGP})$ we only need to prove that $(\emptyset, \text{BGP}) \prec_r (\{\bowtie\}, \text{BGP})$. Indeed, first we will show we can always assume a $(\emptyset, \text{BGP})$-LDFM M works on at most 2 request to the server. Indeed; remember M works by sending some queries to the server, obtaining some answer container, and then sending one of them as an answer to the client. Note however that the construction of the query to the server can depend on the answer containers. To fix we make $\bar{M}$ such that $M(q, G) = \bar{M}(q, G)$ with only 2 request by doing the following; $\bar{M}$ first send the query $(?s, ?o, ?p)$, which we already discussed give us a answer container with the entire graph. Then any query M send to the server, $\bar{M}$ will calculate it's answer container on it's working tape. Finally, whatever answer container M send as a query to the client; $\bar{M}$ will send the query that gives that answer container and then send it to the client. Note with this procedure M and $\bar{M}$ sends both the same query to the client, and so it follows that $M(q, G) = \bar{M}(q, G)$, and $\bar{M}$ works on 2 request to the server.

So now suppose there is $\mathcal{C}$ a class of queries computable under $(\emptyset, \text{BGP})$, such that $(\emptyset, \text{BGP})$ always make more request than $(\{\bowtie\}, \text{BGP})$. Note by what we proved it's only possible if $\mathcal{C}$ is computable on at least two request on $(\emptyset, \text{BGP})$ and there is a $(\{\bowtie\}, BGP)$-LDFM M that compute the query on only one request, and using one answer container D_1, that comes from a query made to the server q_1. So note that then $M(q, G)$ is equivalent to $D_1 \bowtie D_1 \bowtie ... = D_1$. However, since q_1 doesn't depends on any answer container (cause there aren't any answer containers when we send it) we could use the procedure with only one request with a $(\emptyset, \text{BGP})$-LDFM $\bar{M}$; just send q_1 to the server and then D_1 to the client. Note that with this procedure $M(q, G) = \bar{M}(q, G)$ for all graph G and $q \in \mathcal{C}$; and then we are computing $\mathcal{C}$ on one request on $(\emptyset, \text{BGP})$ which is a contradiction. So there is not such a class $\mathcal{C}$; and so $(\emptyset, \text{BGP}) \preceq_r (\{\bowtie\}, \text{BGP})$, finishing our claims.

B.2 Proof of Theorem 5

First note that it's clear that $(\{\cup, \bowtie, ⟕, \pi\}, \text{BRTPF}) \preceq_t (\{\cup, \bowtie\}, \text{BRTPF})$ and $(\{\cup, \bowtie, ⟕, \pi\}, \text{BRTPF}) \preceq_r (\{\cup, \bowtie\}, \text{BRTPF})$. We just have to show that the inequalities are strict.

For both of them we consider the query:

$$q = \text{SELECT}_{\emptyset}(?X_1, b, c)$$

Now if we denote $\text{TRUE} = \{\mu\}$, where μ is the empty mapping and $\text{FALSE} = \emptyset$ it follows that:

$$[\![q]\!]_G = \begin{cases} \text{TRUE} & \exists a, (a, b, c) \in G \\ \text{FALSE} & \nexists a, (a, b, c) \in G \end{cases}$$

We will first proof the data transfer inequality. For this note the query can be answered with $|[\![(?X_1, b, c)]\!]_G|$ transfer on $(\{\cup, \bowtie, ⟕, \pi\}, \text{BRTPF})$; just send $(?X_1, b, c)$

to the server obtaining a register container D_1; and then send the query $q_C = D_1$ to the client.

Now let's consider a $(\{\cup, \bowtie\}, \text{BRTPF})$-LDFM $\bar{M}$ and suppose it can do it using transfer $|[\![(X_1, b, c)]\!]_G|$ on every graph. First note that $\bar{M}(q, G)$, which should be TRUE or FALSE, must be equivalent to an answer container of $\bar{M}$. Indeed, first note that if the answer if TRUE we can't asamble it using $\bowtie$ and $\cup$ and answer containers that aren't TRUE; and so it must be an answer container. Now if the answer is FALSE we could asamble it using two containers D_1, D_2 that have no compatible mappings beetween them. Note that this implies $|D_1|, |D_2| > 0$, which implies $\bar{M}$ is working with data transfer at least $|D_1| + |D_2| > 1$. However if the answer is $False$ then $|[\![(?X, b, c)]\!]_G| = 1$, and it's a contradiction that $\bar{M}$ is working on the same transfer as $|[\![(?X, b, c)]\!]_G|$.

So now consider $\bar{M}$ sends some tripple pattern, without a binding, containing some variable. If that's the case then $\bar{M}$ would work on more than $|[\![(?X_1, b, c)]\!]_G|$; cause any tripple pattern with a variable, on some graph, have at least the same size as $|[\![(?X_1, b, c)]\!]_G|$; and then $\bar{M}$ must also obtain a TRUE or FALSE on a answer container to answer. So then $\bar{M}$ can't send tripple patterns with variables without biding.

So now suppose $\bar{M}$ send some query of the type $t_i|b_i$ to the server. Denote the answer of these B^i, and denote $b_i = \{?x_j \to \{k_j^1, k_j^2, ..., k_j^i\}\}$. Note that b_{i+1} could depend on B^i. Alse note that with our asumption $|\cup_{l,s} \{k_l^s\}| \leq |[\![(?X_1, b, c)]\!]_G|$, Cause if that's not the case we could easily have that $|[\![(?X_1, b, c)]\!]_G| < \sum |B_i|$ That' means that the binding must admit a numbers of values that cant be greater than $|[\![(?X_1, b, c)]\!]_G|$. In particular note that for every graph we can't send more queries than $|[\![(?X_1, b, c)]\!]_G|$.

So let be G a graph, with $N = |[\![(?X_1, b, c)]\!]_G| > 0$, and suppose that every $k_l^s \in G$. Note that then $\bar{M}$ will send the query $t_i|b_i$, obtain the answer B^i and use them to answer the query. So know consider G_1, that is a graph that is exactly equal to G but every k_1^s is know instead $\bar{k}_1^s \neq k_1^s$. So now $\bar{M}$ think that the register B_1 is empty, cause we are not allowing variables to take values of the graph. This will change it's strategy to compute b_2. So now consider G_2, but not every k_2^s is know instead $\bar{k}_2^s \neq k_2^s$. Note that then $\bar{M}$ will think B_2 is empty, what will change the strategy so compute b_3. Repeat this process up to N times, obtaining a graph $\bar{G}$. Note then that when $\bar{M}$ works on $\bar{G}$ it will think $B_1, B_2, ..., B_N$ are empty. Also note that $\bar{M}$ can't send more request to the server, cause it already have N empty answer containers, and so have $N = |[\![(?X, b, c)]\!]_G|$ of data transfer up to this point. Now since $\bar{M}$ can't send more queries, $\bar{M}$ can't diferentiate if the graph $\bar{G}$ is equal to the empty graph, and so it answer incorrectly on $\bar{G}$ or on the empty graph. So $\bar{M}$ can't send tripple patterns with biding.

So then $\bar{M}$ can't send tripple patterns with variables, with binding or without them. So then $\bar{M}$ can only send tripple patterns without variables. But note that on them the biding doesn't have any effect, and $\cup$ and $\bowtie$ doesn't have any effect when sending q_C to the client. So then if $\bar{M}$ could do this strategy on better transfer than $|(?X_1, b, c)|$ so could we do it with a $(\{\bowtie, \cup, \rtimes, \pi\}, \text{BRTPF})$-LDFM; using the same strategy that $\bar{M}$, where the last thing is a contradiction.

So then $\bar{M}$ can't send anything to the server to do better than $|(?X_1, b, c)|$ transfer, which is clearly a contradiction. So $\bar{M}$ can't do better, and so we the first part of our theorem.

Now for the request complexity inequality. For this we consider the same query q. Note that we can answer the query on one request to the server with $(\{\cup, \bowtie, \rtimes, \pi\}, \text{BRTPF})$ with the same strategy that we used on the last part. It's also easy to see we couldn't do any better (We need to at least use some register to construct to answer to the client). Now suppose we can do it the same amount of request on a $(\{\cup, \bowtie\}, \text{BRTPF})$-LDFM $\bar{M}$

Indeed, note that the answer, that is TRUE or FALSE, must come from a answer container of $\bar{M}$. For that note that $\bar{M}$ can only do one request to the server, and thus will only have one answer container D_1. If it's diferent than $True$ or $False$ then there is at least one mapping $\mu \in D_1$ such that there is $?x \in dom(\mu)$. It's clear that then whenever we operate combinations of D_1 with $\bowtie$ and $\cup$ we will still have some mapping ν with $?x \in dom(\nu)$. And so the answer it's diferent from TRUE or FALSE; and so the answer must come from a answer container.

Now consider $G_1 = \{(1, b, c)\}$. Remember that $\bar{M}$ can only do one resquest to the server, and since it must obtain TRUE, $\bar{M}$ is obligated to first send $q_1 = (1, b, c)$. Note that then when working with $G_2 = \{(2, b, c)\}$ $\bar{M}$ will first send the same query q_1, obtaining FALSE as it's only answer container. So then, to answer correctly, $\bar{M}$ should send another request, which is a contradiction that $\bar{M}$ can do better than on a $(\{\cup, \bowtie, \rtimes, \pi\}, \text{BRTPF})$-LDFM. And so we conclude the second part of the theorem.