Communication Cost of Joins over Federated Data

Tamara Cucumides
University of Antwerp, Belgium

Juan Reutter
PUC Chile & IMFD Chile, Santiago, Chile

Abstract

We study the problem of querying different data sources, which we assume out of our control and that are made available by standard web communication protocols. In this scenario, the time spent communicating data often dominates the time spent processing local queries in each server. Thus, our focus is on algorithms that minimize the communication between the query processing server and the federated servers containing data.

However, any federated query can always be answered with linear communication, simply by requesting all the data to the federated sources. Further, one can show that certain queries do require this amount of communication. But sending all the data is definitely not a relevant algorithm from a practical point of view. This worst-case analysis is, therefore, not useful for our needs. There is a growing body of work in terms of designing strategies that minimize communication in query federation, but these strategies are commonly based in heuristics, and we currently miss a formal analysis providing guidelines for the design of such strategies.

We focus on the communication complexity of federated joins when the problem is parameterized by a measure commonly referred to as the certificate of the instance: a framework that has been used before in the context of set intersection and local query processing. We show how to process any conjunctive query in time given by the certificate of instances. Our algorithm is an adaptation of Minesweeper, one of the algorithms devised for local query processing, into our federating setting. When certificates are of the size of the instance, this amount to sending the entire database, but our strategy provides drastic reductions in the communication needed for queries and instances with small certificates. We also show matching communication lower bounds for cases where the certificate is smaller than the size of active domain of the instances.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory); Information systems → Relational database query languages

Keywords and phrases databases, database queries, query federation, communication complexity, adaptive algorithms

Digital Object Identifier 10.4230/LIPIcs.ICDT.2024.5

Funding This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002 and FONDECYT Grant No. 1221799.

1 Introduction

Federated querying services, in which users can use the Web to access data from different independent sources, are already a part of our current Web architecture. The Semantic Web initiative provides one way of doing this: assuming the data is represented under the RDF standard [12], then these repositories can be queried and merged together using SPARQL, the query language for RDF, by means of the SERVICE operator[4]. Remarkably, the RDF/SPARQL standard also allows for linking non-RDF data, by virtually masking it as intermediate answers of SPARQL queries [15, 22]. To illustrate the uses of web query federation, consider an application in which we recommend city attractions to tourists. Basic information about a city can be automatically obtained from Wikipedia, by querying its public wikidata endpoint at (`https://query.wikidata.org/`). But this information can be further filtered, for example, by looking at a weather API (so that recommendations agree with current weather), by comments in social networks, or by reviews from platforms such as Yelp or Foursquare. Because we need up-to-date information, queries must be carried out in real time, and the communication must be performed by http or other similar protocol.

We are thus looking to merge information from several web data repositories, which we can only access by means of http requests.

In the context of Web query federation, the main bottleneck in processing queries is not computational power, but web bandwidth usage. Thus, looking for efficiency in the context of query federation means communicating the fewer number of intermediate results between the different data repositories. Furthermore, there is usually an economic incentive for minimizing API requests, as several APIs and endpoints charge for their information.

Consequently, a good deal of research in query federation has been devoted towards algorithms reducing the amount of communication between servers (see e.g. [8, 19] for a good introduction). But so far most approaches rely on heuristics and data profiles, much resembling how traditional DBMS query planning work, and without strict algorithmic properties, nor tools providing guarantees that one approach works better than others.

We believe that current discussion on query federation would benefit from a formal study establishing the limits of what can be done in terms of query processing. Two main questions arising in this context are, first, to understand what are the theoretical limits in terms of communicating tuples (or bits) in query federation. And second, can we design querying strategies that work within these bounds?

Our answers to these questions are based on the framework of adaptive algorithms, as presented in the work of Demaine et al [7]. This framework can be understood as a relaxation of instance-optimal algorithms, wherein one shows that algorithms are optimal for classes of instances that are classified by specific parameters. Adaptive join algorithms have already been studied for traditional (non-federated) query processing in relational databases [16, 10], and we show that these algorithms are a good starting point for the design of algorithms with adaptive communication complexity. We also show that these algorithms are optimal in terms of communication. Up to our best knowledge, our work is the first to introduce the adaptive framework in terms of communication complexity.

1.1 Problem Definition and Main Results

Let $Q(\mathbf{x})$ be a conjunctive query over a schema with relations $R_1, \dots, R_n$. A federated instance I for Q is a distributed database instance for $R_1, \dots, R_n$, in which the interpretation R_i^I of each relation R_i resides in a different server. The evaluation $Q(I)$ of Q over a federated instance I is the standard evaluation of Q over the database instance containing the interpretation of all relations in I. In web query federation, we wish to materialize all these answers on a separate client, which we abstract as an additional server in our setting. More precisely, we focus on the following problem:

Federated Query Evaluation	
Input:	A conjunctive query Q,
	a federated instance I for Q.
Task:	Compute $Q(I)$ in a separate server.

As we have mentioned, we assume our servers to be connected only through the Web, and thus the most important bottleneck for Federated Query Evaluation is the amount of communication between each server. Our focus is, then, to solve Federated Query Evaluation using the least amount of communication.

Efficient algorithms for Federated Query Evaluation. What do we mean by *efficient* communication? Naturally, the best algorithm should communicate as few information as possible. For a query Q and a federated instance I, the bare minimum would be to communicate only the tuples in each relation that participate in $Q(I)$, that is, each relation R in Q should send only $|R \ltimes Q(I)|$ tuples. But this is too demanding: if $Q(I)$ is empty then we would not be allowed to communicate anything at all. Hence, we settle for algorithms that communicate $|R \ltimes Q(I)|$ tuples per server, plus an extra (small) amount of communication that may depend on Q and I.

In traditional join algorithms, the standard is to compute queries in linear time with respect to the database. But asking for *linear communication* is pointless, as the trivial algorithm where we share the entire database already satisfies this bound, even though most of the time this is not a practical option. Our answer builds from the notion of adaptive algorithms devised by Demaine et al. [7], and recently used in the context of (single server) query-processing [16, 10]. Let us first illustrate the idea with an example.

▶ **Example 1.** Consider query $Q(x, y, z, w) \leftarrow R(x, y) \wedge S(x, z) \wedge T(x, w)$. Assuming a federated instance where R, S and T reside in different servers, computing the answers of Q using a traditional left-deep plan, or even Yannakakis algorithm [24], would necessarily involve the pairwise intersection of the first components of R, S and T, in some order. This operation involves the intersection of several sets of elements, which requires a linear amount of communication, as per standard communication complexity results [11]. However, if we intersect all of R, S and T adaptively, there are several instances for which we can compute the answers of Q with much less communication: in the extreme case where all elements in (the first components of) R are smaller than those in S or T, we can realize that the answer is empty simply by asking R for their biggest element, and comparing it to those in S and T. Ideally, our algorithms should behave much better in these instances.

The *adaptive* analysis provides a way to measure how complicated are instances for processing queries such as the one in Example 1. More precisely, adaptive algorithms assign to each instance I a *certificate*, whose size is then used to bound the performance of algorithms. The notion of a certificate for join queries was already defined in [16]: in essence, a certificate is a set of comparisons between elements of the input relations that serves as a warranty for the output of the query. We can use the same ideas in our context of communication, arriving thus at the first goal of our work.

Goal 1. Devise an algorithm that solves Federated Query Evaluation, in which each relation R_i communicates at most $O(|R_i \ltimes Q(I)| + c)$ tuples, in terms of data complexity[1], where c is the size of the smallest certificate for I, as defined in [16]. In terms of bits of communication, assuming a shared dictionary, **Goal 1** implies communicating $O((|R_i \ltimes Q(I)| + c) \cdot \log n)$ bits in data complexity, where n is the size of active domain (the number of different elements) in I.

We argue that neither left-deep plans nor Yannakakis's algorithm (for acyclic queries) fulfill the requirements of **Goal 1**, and, up to our best knowledge, neither does any of the known approaches for solving SPARQL Query Federation (see e.g. [19, 5, 6, 14, 13, 18]).

[1] In *data complexity* the size of query Q is considered to be fixed, and therefore so is the number of attributes in each relation.

Our proposal is to use known adaptive algorithms for query processing, which can be reshaped into well functioning algorithms in the distributed context. More precisely, in Section 4, we show an algorithm for processing join queries that achieves the communication established by **Goal 1**. This algorithm is the adaptation of Minesweeper[16] over the distributed context.

We remark that our focus is on *deterministic* algorithms, in which we compute queries without the need of random inputs and without a probability of error. This is in line with most database algorithms deployed in practice: a vast majority of them are deterministic. Still, understanding the communication implications of randomized algorithms for federated query processing is a very interesting line of research for future work. We do have randomized algorithms with good guarantees for joining two relations [20] and for intersecting sets [3], and these can be deployed as a part of a distributed left-deep plan for processing federated queries. However, this deployment would have the same problem of standard left-deep plan algorithms, where we may end up doing needless intersections because the results of the query do not match somewhere else upstream. Finally, we note that our setting is fundamentally different from the one studied by Beame, Koutris and Suciu [2], where storage is assumed to be shared amongst a big amount of servers. In fact, if we apply results of Beame et al. to our federated instances, these would default to requiring a linear amount of communication between each relation, which, as explained, is not enough for our proposes.

Proving optimal communication. The second goal is to show that our algorithms communicate an optimal number of bits. For an instance I with a domain of n elements, we assume that the encoding of tuples requires $O(\log n)$ bits (in data complexity). This is more that what can be achieved by intricate data compressing techniques, but it is much closer to the way words are truly stored on database systems. Thus, if one looks for lower bounds measured in bits, we should focus on lower bounds communicating around $c \log n$ bits, plus the output of queries. This is our second goal.

Goal 2. Prove that any algorithm for Federated Query Evaluation must communicate at least $\Omega(|R_i \ltimes Q(I)| + c) \cdot \log n)$ bits in data complexity, where c is the certificate size for I. As it is standard in the literature, we plan to show this by establishing lower bounds on the communication complexity of Federated Query Evaluation. This framework is formally described in Section 2.

We fulfill **Goal 2** for boolean, cyclic queries: given such a query Q, we can show that it must communicate $\Omega(c \log n)$ bits, in data complexity, and provide an alternative $\Omega(\log \binom{n}{c/2})$ bound for boolean acyclic queries. Our construction, however, requires that the size of the certificate is less than the number of elements in the instance, that is, less than the size of the active domain. It is pointless to show a general lower bound for arbitrary queries and larger certificates, as certain queries cannot have bigger certificates. However, we can show a lower bound that applies to infinitely many queries: no matter how large is the certificate and the number of elements in an instance, there is always a boolean query Q that requires $\Omega(c \log n)$ bits to be processed. Naturally, these bounds can be directly transferred to non-boolean queries.

Assumptions in our model. We assume that instances are made of natural numbers, but our techniques can be extended if one works instead with elements from any other ordered, enumerable domain, provided we have a suitable dictionary for these values. We represent federated instances assuming that every relation resides in a different server, but our scenario is general enough to model federated queries involving RDF graphs (which are commonly modeled as a single tertiary or quaternary relation), and for cases in which the queries issued to each different server are more complex than just querying a relation, such as with the SPARQL SERVICE operator. In any of these cases, we just assume that the relations in our join query represent a view with the results of a more complex query issued at the particular server. Further, we also assume our queries are connected. When Q has more than one connected component, then any strategy must involve processing all these components independently, and then combining them using a cross product.

2 Preliminaries

Database basics and notation. We use $[n]$ as a shorthand for $\{1, \dots, n\}$. Conjunctive queries (CQ) are constructs of the form $Q(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}) \wedge \cdots \wedge R_n(\mathbf{y_n})$, where each R_i is a (not necessarily distinct) relation name, each $\mathbf{y_i}$ is a tuple of (not necessarily distinct) variables and/or constants, and $\mathbf{x}$ is a tuple of variables also mentioned in $\mathbf{y_1}, \dots, \mathbf{y_n}$. If all such variables are mentioned in $\mathbf{x}$, then Q is *full*. Further, a query Q is acyclic if it has a join tree, see [24]. We use atoms(Q) to refer to the sets of atoms in Q. The evaluation of a CQ Q over an instance I is the set of tuples $\sigma(\mathbf{x})$, for each assignment σ from the variables of Q to elements in I (and that is the identity on constants), and such that for each atom $R(\mathbf{y})$ in Q, the tuple $\sigma(\mathbf{y})$ is in R^I. For a full query Q without constants, we make use of a function $s_Q(\cdot, \cdot)$ that receives an atom $R(y_1, \dots, y_k)$ in Q and an integer $i \leq k$, and maps to the corresponding position in $\mathbf{x} = x_1, \dots, x_n$ such that $s_Q(R(y_1, \dots, y_k), i) = j$ if $y_i = x_j$. Often, both Q and the atom $R(y_1, \dots, y_k)$ will be understood from context, so we just denote s_Q as a unary function s, as in $s(i) = j$.

Communication Complexity. Here we just outline the concepts that are necessary for stating our results. We refer to e.g. [11] for a comprehensive treatment on this subject. The (two-way) communication complexity of a function $f : X \times Y \to Z$ is defined in terms of *protocols*. Formally, a protocol $\mathcal{P}$ over $X \times Y$ with range Z is a binary tree where each internal node v is labeled either with a function $a_v : X \to \{0, 1\}$ or $b_v : Y \to \{0, 1\}$ that indicates how to walk through the tree (right or left) depending on the input, and each leaf is labeled with $z \in Z$. The cost of $\mathcal{P}$ on input (x, y) is the length of the path taken on input (x, y) and the cost of $\mathcal{P}$ is the height of the tree. The communication complexity of f is the minimum cost of $\mathcal{P}$, over all protocols that compute f.

The communication complexity captures the minimum amount of bits that need to be communicated for a protocol to compute f. As we have mentioned, our focus is on deterministic protocols. For these, the most widely used technique to prove communication lower bounds are fooling sets, defined next.

▶ **Definition 2** (Fooling set). *Let $f : X \times Y \to \{0, 1\}$. A set $S \subset X \times Y$ is called a fooling set for f if there exists $z \in \{0, 1\}$ such that for every $(x, y) \in S$, $f(x, y) = z$ and for every distinct pair (x_1, y_1) and $(x_2, y_2) \in S$ either $f(x_1, y_2) \neq z$ or $f(x_2, y_1) \neq z$*

The following result connects fooling sets with communication lower bounds.

▶ **Lemma 3** ([11]). *If f has a fooling set of size t, then the communication complexity of f is at least $\log_2 t$*

3 Communication Complexity of set intersection

To illustrate our framework we start with the simplest join query: the intersection of two unary relations R and S. Recall that in our federated setting, both R and S reside in different servers, and the answers must be transmitted to a separate master server.

3.1 Adaptive algorithm

Instead of demanding for the full contents of R and S, the master can coordinate a sort-merge intersection, by iteratively probing R for their next element (from the lowest element to the biggest), S for their next element bigger than that of R, and so on until all common elements have been found.

As it turns out, this algorithm is optimal in the following, adaptive, way. Consider, for every instance I over R and S, the number of *changes* that we do between R and S when both relations R and S are sorted onto a single merged list, counting common elements as if they always induce a change.

For example, take $R = \{2, 3, 5\}$ and $S = \{1, 3, 5, 7, 9\}$. When we sort all elements in R and S, we start from element 1 (belonging to S), then we *change* to 2 (belongs to R), then 3, which belongs to both R and S and thus we also count, then to 5, which is also counted, then *change* to 7 (only in S) and advance without changes to 9, since this element is also only in S. There are thus four *changes*.

These changes corresponds, informally, to what Demaine et al. coined as the *certificate* for the intersection of R and S, and the number of changes corresponds to the *size of the smallest certificate*. The intent of Demaine et al. was to study algorithms that would run in linear time with respect to this size. In our case, we can do the same in terms of communication: when intersecting R and S using our coordinated sort-merge intersection, we only communicate $\theta(c \log n)$ bits, where c is the size of the smallest certificate.

Note that the size of the smallest certificate for R and S ranges from 1 (say, when all elements in R are smaller than those in S) to $2\min(|R|, |S|)$ (for example, when R contains all even numbers up to a given integer N, and S contains all odd numbers up to N). Thus, asymptotically, this algorithm communicates the same information than just sending R or S (up to encoding of elements). But, the smaller this number is for these relations, the smaller the communication issued by our algorithm. This is in concordance with a simple intuitive analysis of this algorithm: in practice it just sounds much better than sending the complete relations.

3.2 Lower bounds

We can also use communication complexity to show that our algorithm is optimal when the certificate size is relatively smaller than the number of elements in R and S.

▶ **Proposition 4.** *The communication complexity of Federated Query Evaluation, on input $Q(x) \leftarrow R(x) \wedge S(x)$ and an instance with n elements and certificate size up to c, is $\Omega(\log \binom{n}{c/2})$.*

When c is smaller that n (say, bounded by n^{ϵ}, for a fixed $\epsilon < 1$) then $\log \binom{n}{c/2}$ is $\Omega(c \log n)$, which is what we are looking for, as it shows that our algorithm is optimal in terms of communication. While this result does not give the best bounds when certificates are comparable to n, one important advantage of our sort-merge intersection is that it can be carried out using standard database technology, whereas more nuanced algorithms may involve requests that cannot be processed over a web-based database endpoint.

Proof of Proposition 4. We show that the problem of checking whether the boolean version of Q is empty already requires said communication. Naturally, this is also a lower bound for the non-boolean query Q, since one can always use the answers of Q to answer the boolean version. The proof is by building a fooling set: a collection of instances $I_i = (R^{I_i}, S^{I_i})$, in such a way that Q is empty over any such pair, but nonempty over (R^{I_i}, S^{I_j}) for $j \neq i$.

Define then, for each set $A \subseteq [n]$ of size $|A| = c/2$, the instance I_A given by $R^{I_A} = [n] \setminus A$ and $S^{I_A} = A$.

The number of different pairs (R^{I_A}, S^{I_A}) is $\binom{n}{c/2}$. Further, we verify that $R^{I_A} \cap S^{I_A} = \emptyset$, but $R^{I_A} \cap S^{I_B} \neq \emptyset$ for $A \neq B$, so this collection is indeed a fooling set of size $\binom{n}{c/2}$.

It remains to show that all such instances have certificate at most c. By construction, pairs (R^{I_A}, S^{I_A}) may have up to $2|A| = c$ changes. Further, (R^{I_A}, S^{I_B}) for $A \neq B$ require 1 change per element in $A \cap B$, and every other element in B induces at most 2 changes, for a grand total which is always bounded by c. ◀

The communication bound holds (in data complexity) when intersecting more than two relations. Indeed, for a query $Q(x) \leftarrow T_1(x) \wedge \cdots \wedge T_\ell(x)$, assume one server contains T_1, and the other contains all remaining relations. We can then reuse the proof above, setting $T_1^{I_A}$ as R^{I_A} and each of $T_2^{I_A}, \ldots, T_\ell^{I_A}$ as S^{I_A}, to obtain a similar bound[2]. Moreover, this lower bound transfers to the federated multiparty case where each T_i resides in a different server. If there was an algorithm using less communication to evaluate Q in the multiparty setting, then we can mimic this algorithm on the two-party setting: every time a relation T_i, $i > 2$ communicates with the master server, in the two party setting this communication happens instead between the master and the server containing relations $T_2, \ldots, T_\ell$.

4 Communication Complexity of natural joins

In this section, we present algorithms for solving Federated Query Evaluation with low communication. We will begin by recalling the extension of certificates to relational queries introduced in [16], and we will follow with the algorithm and its analysis.

4.1 Certificates

As in Section 3, the idea is that certificates play the role of *minimal witnesses* for the evaluation of a query over a relational instance.

Let R be a relation of arity k. An *index tuple* for R is a tuple $(a_1, \ldots, a_\ell)$ of $\ell \leq k$ elements, where each a_i is either -1 or a natural number.

We use index tuples to produce atoms of the form $R[a_1, \ldots, a_\ell]$, for $(a_1, \ldots, a_\ell)$ an index tuple. These atoms represent, in each instance I, an element from a specific tuple in the instantiation R^I of R over I. Specifically, the atom above refers to the tuple in which the first position contains the a_1-th smallest element amongst all elements in the first position of tuples in R^I, the a_2-th smallest element in the second position, and so on.

This is formalized as follows. The interpretation $R^I[a]$ of $R[a]$ over an instance I, for $a \geq 0$, is the a-th smallest value in the first position of tuples in R^I, i.e., the a-th smallest value retrieved by the evaluation in I of query:

$$Q(x_1) \leftarrow R(x_1, \ldots, x_k).$$

Further, $R^I[-1]$ represents the largest element in R^I retrieved by such query.

[2] Technically speaking, the certificate for these instances is slightly bigger. We give more details in the following section.

Next, $R^I[a_1, \dots, a_{i-1}, a_i]$, for $a_i \geq 0$, represents the a_i-th smallest element amongst all tuples in R^I whose first $i-1$ elements correspond to $R[a_1], \dots, R[a_1, \dots, a_{i-1}]$. That is, $R^I[a_1, \dots, a_i]$ is the a_i-th smallest element retrieved by the evaluation over I of query:

$$Q(x_i) \leftarrow R(R^I[a_1], R^I[a_1, a_2] \dots, R^I[a_1, \dots, a_{i-1}], x_i, \dots, x_k).$$

Likewise, $R^I[a_1, \dots, a_{i-1}, -1]$ is the largest element retrieved by such query.

We use the notation $\mathit{tup}^I(R[a_1, \dots, a_\ell])$, for $\ell \leq k$, to refer to the tuple given by $(R^I[a_1], R^I[a_1, a_2], \dots, R^I[a_1, \dots, a_\ell])$. In our application we will always have that a_i, when positive, is at most the number of elements retrieved by the corresponding query. If, in any case, there were less elements, then a_i will just represent the largest element, as with -1.

▶ **Example 5.** Consider an instance I where

$$R^I = \{(1,1), (1,2), (3,3), (5,5)\}.$$

Then, interpretation $R^I[0]$ of $R[0]$ is element 1, the smallest element in the first position of a tuple in R^I. Next, $R^I[0,1]$ corresponds to element 2: the second smallest element in the second position of a tuple in R^I that starts with 1. Finally, construct $R^I[-1,0]$ is 5: the smallest element in the second position of a tuple in R^I that starts with $R^I[-1] = 5$.

We use $\mathit{tup}^I(R[0,1])$ to refer to $(R^I[0], R^I[0,1]) = (1,2)$.

Index tuples allow us to pinpoint elements in instances, without referring to the actual element, only their ordering within a certain relation. The notion of certificate is then based on comparing two of these atoms.

▶ **Definition 6** (Argument [16]). *An argument $\mathcal{A}$ is a set of comparisons of the form:*

$$R[\mathbf{a}] \ \theta \ R[\mathbf{b}],$$

with a, b index tuples $\theta \in \{<, =, >\}$ and R, S atoms of the query. An instance I satisfies an argument if $R^I[\mathbf{a}] \ \theta \ S^I[\mathbf{b}]$ holds in I for every comparison in $\mathcal{A}$.

We are ready to define the notion of certificate. For a full query Q and an instance I, a witness for $Q(I)$ is a set consisting of one atom $R[a_1, \dots, a_k]$ for each atom $R(x_1, \dots, x_k)$ in Q, and such that there exists an output tuple $\mathbf{t}$ in $Q(I)$ for which $\mathit{tup}^I(R[a_1, \dots, a_k]) = (\mathbf{t}_{\mathbf{s(1)}}, \dots, \mathbf{t}_{\mathbf{s(k)}})$ holds for each atom in Q (here s is the function $s_Q(\cdot,\cdot)$ defined in Section 2).

▶ **Definition 7** (Certificate [16]). *An argument $\mathcal{A}$ is a* certificate *for an instance I over a query Q if i) I satisfies $\mathcal{A}$, and ii) for any other instance J satisfying $\mathcal{A}$, the set of witnesses for $Q(I)$ and $Q(J)$ coincide. The size of a certificate is the number of comparisons in $\mathcal{A}$.*

As we are interested in the size of the smallest certificate, we informally refer to *the size of the certificate* for I over Q when we really talk about the size of the smallest certificate for I over Q.

▶ **Example 8.** Consider a query $Q(x,y,z) \leftarrow R(x,y) \wedge S(x,z) \wedge T(y,z)$, and an instance I given by $R^I = \{(1,1), (1,2), (3,3), (5,5)\}$, $S^I = \{(1,1), (2,2)\}$ and $T^I = \{(1,1)\}$.

The tuple $(1,1,1)$ is the sole output for this query. A possible certificate for Q consists of the following 8 arguments:

$$\begin{array}{lll} R[0] = R[0,0], & R[0,0] = S[0], & S[0] = S[0,0], \\ S[0,0] = T[0], & T[0] = T[0,0], & S[0,-1] = S[0,0] \\ S[-1] < R[1], & T[-1] < R[0,1]. & \end{array}$$

All these arguments, together, imply that the smallest tuple (in lexicographical order) of R, T and S on any instance satisfying the certificate is always witness for the query. Further, because of the argument $S[-1] < R[1]$, the largest element in the first position of S is smaller than the second element in the first position of R. This rules out any join where x is not the element $R^I[0]$. Next, argument $T[-1] < R[0,1]$ indicates that the largest element in the first position of T is smaller than the element in the second position of a tuple in R starting with $R^I[0]$. This rules out the possibility of a join in y apart from $T[0]$, conditioned to $x = R^I[0]$. Finally, argument $S[0,-1] = S[0,0]$ says that there is only one tuple in S whose value in the first position is $S[0]$. This means that the only witness for z, conditioned to $x = R^I[0]$ and $y = T[0]$, is $S[0,0]$. All of these facts together imply that any instance J satisfying the certificate cannot have any other answer to Q apart from $(R^I[0], R^I[0,0], T^I[0])$.

We remark that the certificate need not be linked to the size of the instance, nor to the query ouput. In general, the size of certificates range from 1 to the size of the instance [16].

Order of tuples is crucial for certificates. The other important subtlety of certificates is that the ordering of tuples is crucial for the size of the certificates. To see this, consider query $Q(x_1,x_2) \leftarrow R(x_1,x_2) \wedge S(x_1,x_2)$, and an instance I given by $R^I = [1] \times [n]$ and $S^I = [2] \times [n]$. Then R and S do not match on x_1, and the argument $R[-1] < S[0]$ is a certificate for I and Q. But now, let us assume we have ordered our tuples in the opposite way. Now the query is $R(x_2,x_1) \wedge S(x_2,x_1)$, and the interpretations are $R^I = [n] \times [1]$ and $S^I = [n] \times [2]$. The certificate must now include all n arguments of the form $R[i][-1] < S[i][-1]$.

Our results in this section are consistent with the size of the certificate, given a particular ordering of tuples: communication depends on the certificate size, which again may be higher or lower depending on this ordering.

Khamis et al. have studied notions certificates which do not depend on the ordering of variables [10]. And indeed, one could adapt our algorithms for this notion of certificate. We have decided to work with the original proposal of Ngo et al. [16] because this leads to algorithms that are easily implemented over existing database architectures and systems. in Section 6 we explore an intermediate solution based on certificates for several different orderings of tuples, which, again, can be easily implemented on existing architectures, specially in cases such as RDF, where the arity of relations is low (see e.g. [23, 9, 1]).

4.2 Distributed Minesweeper

Minesweeper [16] is an algorithm to process natural join queries that has been shown to run with adaptive time guarantees for classes of queries with acyclicity or bounded treewidth properties. The main idea of Minesweeper consists of repeatedly issuing so-called “probe points”, or tuples of elements, which are then queried to see if they belong to the output of the query or not. Then, either the probe point is a valid output tuple, or else we can exhibit a “gap” around it, indicating that no instance tuple of this relation has elements inside this gap. These gaps are then stored as constraints, so that the next point to probe is such that it does not satisfy any constraints. The algorithm runs until there are no new points to probe.

The distributed version. Our distributed version adapts the original algorithm onto the distributed setting, and is designed to process queries regardless of their treewidth. Further, we streamline the communication to avoid a factor that is exponential with respect to the query, that is present in the original Minesweeper. Probe points are now issued in lexicographical ordering, and the algorithm divides them into a probe for each relation.

Further, the master now caches all answers from the relations to avoid issuing the same point two times. We begin by illustrating the algorithm by means of an example, and then follow to present the most important parts. We will finish with the analysis, which is tailored specifically towards the communication, and (because of the way we analyze probe points) requires additional techniques from those in the original Mineswweeper [16].

▶ **Example 9.** Consider again query $Q(x, y, z) \leftarrow R(x, y) \wedge S(x, z) \wedge T(y, z)$ from Example 8, and the federated instance I given by the interpretations $R^I = \{(1, 1), (1, 2), (3, 3), (5, 5)\}$, $S^I = \{(1, 1), (2, 2)\}$ and $T^I = \{(1, 1)\}$. The sole output in $Q(I)$ is $(1, 1, 1)$.

In the distributed version of Minesweeper, we start with probe point $(1, 1, 1)$. This implies asking all of R, S and T for pair $(1, 1)$. All three servers will return **true**, indicating they contain the pair $(1, 1)$, and the algorithm stores this in the cache. Next is $(1, 1, 2)$, which implies asking R for $(1, 1)$, S for $(1, 2)$ and T for $(1, 2)$. We will not probe R this time, because pair $(1, 1)$ is already in the cache. However, pair $(1, 2)$ is not in S, so it returns the constraint $\langle 1, (2, \infty)\rangle$, which indicates that there are no tuples of the form $(1, a)$ in S, with $a \in [2, \infty]$. As explained in Example 8, this information is captured by the argument $S[0, -1] = S[0, 0]$ in the certificate for Q and I. Analogously, relation T also returns $\langle 1, (2, \infty)\rangle$.

The next pair in lexicographical order is $(1, 1, 3)$, but because of the constraints in S and T we know that no tuple $(1, 1, b)$, for $b > 1$ is in the output of the query. Thus, the next point issued is $(1, 2, 1)$. R has the pair $(1, 2)$ so we store this in the cache for R. S is not queried because $(1, 1)$ is already cached for S, and T is queried for $(2, 1)$, returning $\langle (2, \infty)\rangle$. As before, this constraint can be matched to an argument in the certificate, in this case to $T[-1] < R[0, 1]$. Continuing the search for the next probe point, in lexicographical order, that satisfies all constraints, we arrive at $(2, 1, 1)$. Relation R is probed with pair $(2, 1)$, relation S with $(2, 1)$ and relation T is not probed, because $(1, 1)$ is in the cache of T. Relation R does not have $(2, 2)$, so it returns the constraints $\langle (2, 2)\rangle$. Relation S returns $\langle 2, (1, 1)\rangle$. The next and final point is $(3, 1, 1)$. Here R does return **true**, but S returns instead the constraint $\langle (3, \infty)\rangle$. The last two rounds can be matched to the argument $S[-1] < R[1]$, we verify that there are no further matches for variable x, and finish the search for output tuples.

The algorithm. For conciseness, the algorithm is shown for full queries, without self-joins, and that feature more than one relation. This is without loss of generality. If a query is not full, then we first compute the answer for the full query, and then project locally at the master server. We do not consider this a shortcoming of the algorithm, as the lower bounds in Section 5 are always given for boolean queries. Self-joins are treated by packing together all atoms of the same relation R in a view, probing instead the server for R for the results of this view. Finally, queries involving only one relation can be dealt with separately by just sending the query to the corresponding server.

We assume that the ordering of variables is preserved between the head and the body of queries: any variable x that appears before a variable y in the head $Q(\mathbf{x})$ of the query, always appears before y in any of the atoms $R_i(\mathbf{y}_i)$ in the body. In Section 6 we discuss alternative algorithms that can deal different orderings in atoms.

Algorithms 1 and 2 contain the main parts of the distributed version of Minesweeper. Algorithm 1 requires getProbePoint($\mathbf{t}$) (Algorithm 3 in Appendix A.1), which returns the first tuple that is both greater than $\mathbf{t}$ (in lexicographical order) and that satisfies the constraints. Once this tuple is generated, the algorithm builds the projection of $\mathbf{t}$ to the variables of each relation R of the query, and sends this to probe$_R$(). In turn, probe$_R$($\mathbf{t}$) checks whether such tuple belongs to R, and outputs either **true**, if it does, or a constraint if it does not. The algorithm generates the constraints that relate to the most significant position in $\mathbf{t}$; this is important since we are searching for new probe points in lexicographical ordering[3]. Finally, distributed Minesweeper terminates when there are no more tuples to generate. The final answer is constructed by evaluating Q over the cached data.

Algorithm 1 Distributed Minesweeper - Master server, query Q.

```
1:  $\mathbf{t} = (0, \dots, 0)$
2:  while $\mathbf{t}$ =getProbePoint($\mathbf{t}$) $\neq$ NULL do
3:      for $R(x_1, \dots, x_k) \in$ atoms($Q$) do
4:          $\mathbf{t}_R \leftarrow (\mathbf{t}_{s(1)}, \dots, \mathbf{t}_{s(k)})$
5:          if $(\mathbf{t}_R, R)$ is not in the cache then
6:              return$_R \leftarrow$ probe$_R(\mathbf{t}_R)$
7:              if return$_R$ = true then
8:                  store $(\mathbf{t}_R, R)$ in the cache
9:              else(return$_R$ is a constraint)
10:                 store (return$_R$, $R$) as a constraint
11: return evaluation of $Q$ over the cache.
```

Algorithm 2 probe$_R(\mathbf{t})$ - server storing R.

```
1:  $k \leftarrow$ arity($R$)
2:  for $i = 1$ to $k$ do
3:      if $R(\mathbf{t}_1, \dots, \mathbf{t}_i, x_{i+1}, \dots, x_k)$ is empty then
4:          if $R(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, x_i, \dots, x_k) \wedge x_i > \mathbf{t}_i$ is empty then
5:              return $\langle \mathbf{t}_1, \dots, \mathbf{t}_{i-1}, (\mathbf{t}_i, \infty) \rangle$
6:          else
7:              $e \leftarrow$ smallest element such that $R(\mathbf{t}_1, \dots, \mathbf{t}_{i-1}, e, x_{i+1}, \dots, x_k) \wedge e > \mathbf{t}_i$ is not empty
8:              return $\langle \mathbf{t}_1 \dots, \mathbf{t}_{i-1}, (\mathbf{t}_i, e-1) \rangle$
9:  return true
```

Analysis. To bound the communication of distributed Minesweeper, we focus on the number of probe points generated by getProbePoint() and the communication associated with each one of them. We divide probe points $\mathbf{t}$ into three types: (1) probe points $\mathbf{t}$ that are part of the output, (2) probe points $\mathbf{t}$ for which some R returns a constraint that can be associated with a comparison in the certificate, and (3) probe points $\mathbf{t}$ such that all relations either return **true** or a constraint over private attributes (i.e. attributes that appear only in one atom of the query). The following proposition bounds the communication of Distributed Minesweeper. In order to abstract from encoding issues, we give bounds in terms of the number of tuples (or constraints) communicated by the algorithm.

▶ **Proposition 10.** *The number of tuples communicated by Distributed Minesweeper to each server $R \in$ atoms(Q) is in $O(c + |R \ltimes Q(I)|)$, in data complexity, where c is the certificate size for I over Q.*

[3] In contrast, the original Minesweeper performs a search in the vicinity of $\mathbf{t}$. We avoid this search because it may lead to more communication.

Assuming servers use a dictionary for their values, the communication outlined above can be written as $O((c + |R \ltimes Q(I)|) \cdot \log n)$ if we assume the query processing server also has access to the dictionary, or $O(c \log n + ||R \ltimes Q(I)||)$ if we do not assume this, where $||R \ltimes Q(I)||$ is the amount of bits required to encode $R \ltimes Q(I)$.[4]

The proof, as we explained before, follows by showing that the number of iterations of the algorithm (i.e. the number of probe points) is at most $2v(2c + |Q(I)|)$, where c is the size of the certificate for Q and I and v is the number of variables in Q. This number comes up by directly counting probe points of type (1) and (2) and charging probe points of type (3) to the priors. Unlike the original Minesweeper, our probe points are generated in lexicographical order. Although this allows us to simplify the constraints that are returned while roughly keeping the number of probe points, it also changes the way we count such probe points, and especially how we deal with case (3). Finally, when addressing the communication between a relation R and the master, we only count the first time $\mathbf{t}_R$ is probed into R, since the result will either be discarded by the constraint returned to this first call, or will be already in our cache.

A natural question at this point is whether the communication bounds exhibited are a property of Minesweeper only, or if they are shared by other worst-case algorithms such as Yannakakis or Leapfrog Trie Join (LFTJ)[21], and what communication guarantees we can achieve (if any) when they are adapted to the federated scenario. Ngo. et al [16] tackled a similar question: whether those worst-case optimal algorithms could run in an instance optimal runtime and the answer was negative for both. Following similar arguments, we can prove that there are instances for which both, Yannakakis and LFTJ need to communicate significantly more tuples, compared to distributed Minesweeper (see Appendix A.2). Let us end this section with a few additional remarks from the point of view of database practice.

A note on implementation via database queries. In the context of Web federation, we cannot impose servers to adhere to a protocol of our choosing. In this case, most probably we will only have the ability to issue SQL or SPARQL queries remotely, to the endpoint. One of the advantages of our build is that it can be easily simulated with database queries (lines 3, 4 and 7 in Algorithm 2), and we only pay the cost of issuing (at most) $k + 1$ queries instead of one call to probe_R, and this can be further alleviated with more caching.

Certificate sizes in real life. Another important question is what happens for certificates in real life queries. A thorough analysis is out of the scope of this paper, but certificates tend to be much smaller for queries with high selectivity. For example, recall query $R(x, y) \wedge S(x, z) \wedge T(x, w)$ from the introduction. This query abstracts high-selectivity SPARQL queries that are typical in benchmarks (see e.g. the LUBM benchmark `http://swat.cse.lehigh.edu/projects/lubm/`), such as *retrieve name and address of all professors working in a university*. Here relation R abstracts the professor-works-in relation, and the other relations abstract personal information from every type of person in the database, and are therefore much bigger. The certificate for this query is of size comparable to the number of elements in the first position of R (the number of professors). This adequately captures the fact that R is the relation inducing high-selectivity, and we remark our algorithm achieves this without the need of any heuristics.

[4] If the dictionary is not known to the master server, then one needs to retrieve final tuples from the server containing R with an extra $|R \ltimes Q(I)|$ requests, which require $(|R \ltimes Q(I)| \log n + ||R \ltimes Q(I)||)$ bits of communication.

5 Lower Bounds

In this section we show lower bounds for the communication complexity of the Federated Query Evaluation problem, for arbitrary conjunctive queries. Recall that our goal is to show a $\Omega(c \log n)$ lower bound, where n is the number of elements in the instance, and c is the certificate size. We begin with communication bounds that are applicable to *any* conjunctive query. For acyclic queries, we cannot do much more than Proposition 4, but cyclic queries allow for a tighter bound that does satisfy our goal.

However, this first result requires that the certificate is at most the size of the number of elements in the instance. It is not possible push forward a general result applicable to any query, because certain queries cannot have bigger certificates. But for queries of a specific form, we can go much further: we finish this section showing that, no matter the instance and the certificate sizes, there are always queries for which Federated Query Evaluation needs to communicate around $c \log n$ bits.

For obvious reasons, queries in this section are assumed to contain more than one relation, as otherwise we can always process them locally.

▶ **Proposition 11.** *The communication complexity of Federated Query evaluation, on input a query Q (using more than one relation) and an instance I with at most n elements and certificate size c at most $\frac{n}{2}$ (i.e. $c \leq \frac{n}{2}$) is, in data complexity:*

- $\Omega(\log \binom{n}{c/2})$ *bits, if Q is acyclic, and*
- $\Omega(c \log n)$ *bits if Q is cyclic.*

Proof. For both cases, the proof follows by constructing a fooling set of adequate size. We show the proof for a simpler class of cyclic queries. We leave the general case, as well as the acyclic case, for the full version.

Our class of queries use only binary relations, and we assume that in the cycle of the graph of Q there is a variable x that participates in exactly two different atoms, one using relation R and the other using relation S. Without loss of generality, we also assume this variable is in the rightmost position in both R and S. If this is not the case, one can always reorder all attributes in relations before processing the query in the master server.

As in the proof of Proposition 3, our lower bounds are for two-party communication (R and S), assuming they already know the rest of the database, and this transfers to our multiparty framework without any added communication, in data complexity.

Fix an integer $k > 0$ for a tuple $b = b_1, \ldots, b_k$, consider relations $R_{b_1,\ldots,b_k}$ and $S_{b_1,\ldots,b_k}$, defined as follows.

- $R_{b_1,\ldots,b_k}$ is the union of sets of tuples $\{(i, a) \mid a \neq b_i\}$, for $1 \leq i \leq k$, and
- $S_{b_1,\ldots,b_k}$ contains all tuples (i, b_i), for $1 \leq i \leq k$.

Our fooling set is constructed by instances $I_{b_1,\ldots,b_k}$ in which the interpretation of R and S correspond to one of the n^k pairs of instances $(R_{b_1,\ldots,b_k}, S_{b_1,\ldots,b_k})$, and the interpretation of all other relations in the query contains all k pairs $(1,1),\ldots,(k,k)$

The following claim establishes that the set $\{I_{b_1,\ldots,b_k}, b_j \in [n]\}$ is indeed a fooling set for the federated evaluation problem, on input Q. In turns, this entails that the (deterministic) communication complexity of this problem is at least $\log n^k = k \log n$.

▷ Claim 12. The evaluation of Q over any instance $I_{b_1,\ldots,b_k}$ is empty. The evaluation of Q over any instance whose interpretation are relations $R_{b_1,\ldots,b_k}$ and $S_{b'_1,\ldots,b'_k}$, where at least one b_i is different to b'_i, is not empty.

To finish the proof we need to count the size of the certificates we may form out of our instances, and ensure they satisfy the necessary bounds.

Let us first analyze the certificate for some instance $I_{b_1,\dots,b_k}$. For a given $i \leq k$, we need arguments $R[i, b_i - 1] < S[i, 0]$ and $S[i, -1] < R[i, b_i]$, plus a series of extra arguments that depend on the query: if the rest of the query uses (binary) relations $T_1, \dots, T_\ell$, then we need arguments $T_j[i] = T_j[i, 0]$, $T_j[i, 0] = T_j[i, -1]$, for each $1 \leq j \leq \ell$, and then $T_j[i] = T_{j+1}[i]$ for $1 \leq j \leq \ell - 1$. Finally, we also need arguments $R[i] = T_1[i]$ and $S[i] = T_1[i]$. The last two arguments ensure that relations contains no more values in its first components: $R[-1] = R[k]$, $S[-1] = S[k]$, $T_1[-1] = T_1[k]$. This gives us $4 + 3\ell$ arguments for each $1 \leq i \leq k$, plus three additional ones, for a grand total that is less than $4k(\ell + 2)$.

We also need to review the instances obtained by combining two instances from the fooling set. Let us thus build the certificate for the instance grouping $R_{b_1,\dots,b_k}$, $S_{b'_1,\dots,b'_k}$ and the remaining instances as in any $I_{b_1,\dots,b_k}$. Any $i \leq k$ for which $b_i = b'_i$ functions exactly like the case above. If $b_i \neq b'_i$, then the tuple $tup(S[i, 0])$ matches with the corresponding tuple in R. Let us assume that $b_i > b'_i$. Then we cover this using arguments $S_{[}i, 0] = R[i, b'_i]$ and $S_{[}i, -1] < R[i, b'_i + 1]$. If $b_i < b'_i$, this means that we have a gap in R before reaching b'_i, so we use instead arguments $S_{[}i, 0] = R[i, b'_i - 1]$ and $S_{[}i, -1] < R[i, b'_i]$. The rest of the certificate is as before, and thus the certificate for this instance is of the same size.

Summing up, the communication complexity is $k \log n$, when given input instances of certificate size at most $4k(\ell + 2)$. Choosing $k = c/(4(2 + \ell))$, we obtain the required complexity bound: the communication complexity on instances of certificate size at most c is $c/(4(2 + \ell)) \log n$, which is $\Omega(c \log n)$ in data complexity. ◀

A second lower bound. Our next result offers a lower bound for arbitrary certificate sizes, albeit not for any query: the larger the certificate size, the larger the relations must be, as the certificate is bounded by the number of tuples in each relation. The proof of this proposition follows from boosting the number of available tuples used for the proof of Proposition 11.

▶ **Proposition 13.** *For every $n, c > 1$ there is a query Q for which the communication complexity of Federated Query Evaluation on input Q and instances I with at most n elements and certificate size at most c must communicate $\Omega(c \log n)$ bits.*

6 Moving beyond certificates based on ordering of tuples

There is a close relationship between the way certificates are built and the constraints returned by Minesweeper. At the same time, distributed Minesweeper is modular enough so that any other notions of certificates and constraints can be plugged into our algorithm, without any essential modifications. Khamis et al., joining the team that proposed the original Minesweeper algorithm, studied certificates–and constraints–defined as *gap boxes*, or multidimensional cubes over the space given by all variables participating in a query[10]. Gap boxes make up for a very elegant notion of certificate, which tend to be much smaller than the one we defined in this paper, and would therefore lead to much smaller communication in the distributed version. Unfortunately, working with gap boxes takes us a bit too far away from what can be expected from federated servers; we are mostly stuck with probe algorithms that can easily be expressed in common database query languages. For this reason, and inspired by the *Triple Pattern Fragment* SPARQL Federation infrastructure [23], we study a milder improvement based on incorporating different *orders* for probing relations.

Pattern Fragment Certificates. Let R be a relation of arity k, and o be a permutation of $[k]$. Then $o(R^I)$ is the permutation of each tuple in R^I according to o, that is, the set of all tuples $(\mathbf{t}_{o(1)}, \dots, \mathbf{t}_{o(k)})$, for $(\mathbf{t}_1, \dots, \mathbf{t}_k)$ in R^I. Further, for an index tuple $[a_1, \dots, a_k]$, the construct $o(R^I)[a_1, \dots, a_k]$ represent the instantiation of $[a_1, \dots, a_k]$, but using $o(R^I)$ instead of R^I.

A *pattern fragment* (PF)-argument $\mathcal{T}$ is a set of comparisons $R^{o_R}[\mathbf{a}]\ \theta\ S^{o_S}[\mathbf{b}]$, where R and S are relations of arities k_R and k_S, a, b are index tuples, $\theta \in \{<, =, >\}$ and o_R and o_S are permutations of $[k_R]$ and $[k_S]$, respectively. An instance I satisfies a TPF-argument of the form above if $o_R(R^I)[\mathbf{a}] \quad \theta \quad o_S(S^I)[\mathbf{b}]$ holds in I for each such comparison in $\mathcal{T}$. The definition of certificate transfers verbatim: a PF-argument $\mathcal{T}$ is a PF-certificate for an instance I over a query Q if I satisfies $\mathcal{T}$ and for any other instance J satisfying $\mathcal{T}$, the set of witnesses for $Q(I)$ and $Q(J)$ coincide.

▶ **Example 14.** PF-certificates can be much smaller. For example, recall the query $Q(x_1, x_2) \leftarrow R(x_2, x_1) \wedge S(x_2, x_1)$ and instance I given by $R^I = [n] \times [1]$ and $S^I = [n] \times [2]$, as defined in Section 4, which had a certificate with n arguments. The PF-certificate now consists only of the comparison $R^o[-1] < S^o[0]$, where o is the permutation defining $o(1) = 2, o(2) = 1$. This argument state that, when R and S are inverted, the largest element in the first position of the inversion of R is smaller than the smallest element in the first position of the inversion of S.

PF-Minesweeper. In order to adapt our algorithm, we assume that each federated server storing a relation R of arity k has made available some of the possible $k!$ permutations for R. Triple Pattern Fragments, for example, mandates that all 6 permutations must be made available for RDF graphs, which are stored as triples. Then, instead of issuing probe_R on line 7 of Algorithm 1, the master issues a version $\text{probe}^o_R(o(\mathbf{t}))$ for each permutation o made available by the server of R. All of these probes are of course answered by the server storing R, and the response is the expected response for a server storing $o(R^I)$ receiving $(o(\mathbf{t}))$. With a little care, all the constraints returned by such algorithms can be combined together, and getProbePoint() can be made to work just as before. The resulting communication is similar to what we had before, except we replicate the whole probing process for each available permutation. Since the number of permutations ultimately depend on the atoms used in the query, we arrive at the same data complexity communication bounds.

▶ **Proposition 15.** *The number of tuples communicated by PF-Minesweeper to each server $R \in atoms(Q)$ is in $O(c + |R \ltimes Q(I)|)$, in data complexity.*

What have we gained? In terms of combined complexity, we can now reduce the number of calls to depend only on the size of the relations, and not on the total number of variables in the query; this is because we can now deal with self-joins directly instead of packing them into views. Further, as more permutations are made available, certificates can only be smaller, and thus the communication in this respect diminishes. However, more orders require more communication in each step, as each relation must be probed once per each permutation available. We believe that this trade-off is worthy of future study, as it may provide practical guidelines for future implementation of web query federation strategies.

7 Conclusions and future work

Query federation systems have already been implemented for a few years now, but our work is the first to provide a formal framework in which one can analyze federation strategies without resorting to heuristics or probabilistic distributions. Naturally, our work would best be deployed in a context where certificates can be small. Yet, highlighting the theoretical importance of certificates in the context of web federation is already an important contribution, and our work can guide the design of federated strategies even in context where certificates sizes are comparable to the size of instances.

As for distributed minesweeper, we expect it to shine the most on contexts where queries involve several joins, such as in graphs. The reason we expect this is partly because this has been the case for worst-case optimal algorithms in general [17], and partly because we expect that a distributed version of Yannakakis (using sort-merge join for pointwise semijoins) would probably perform reasonably in practice, even if we know of pathological cases where the communication is much worse. We are looking forward to a prototype implementation of our algorithms, to help us understand how much do these theoretical results transfer to practice.

Lastly, we would like to incorporate randomized protocols into our framework, in which actions can be dictated by a random string. Most communication protocols can be improved when randomization is allowed, so it may be the case that we can do this for database queries as well. Proving lower bounds in this case would involve orchestrating distributions and certificates of instances, an interesting problem for future research.

References

1 Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L Reutter, Javiel Rojas-Ledesma, and Adrián Soto. Worst-case optimal graph joins in almost no space. In *Proceedings of the 2021 International Conference on Management of Data*, pages 102–114, 2021. doi:10.1145/3448016.3457256.

2 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):1–58, 2017. doi:10.1145/3125644.

3 Joshua Brody, Amit Chakrabarti, Ranganath Kondapally, David P Woodruff, and Grigory Yaroslavtsev. Beyond set disjointness: the communication complexity of finding the intersection. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 106–113, 2014. doi:10.1145/2611462.2611501.

4 Carlos Buil-Aranda, Marcelo Arenas, Oscar Corcho, and Axel Polleres. Federating queries in sparql 1.1: Syntax, semantics and evaluation. *Journal of Web Semantics*, 18(1):1–17, 2013. doi:10.1016/j.websem.2012.10.001.

5 Carlos Buil-Aranda, Axel Polleres, and Jürgen Umbrich. Strategies for executing federated queries in sparql1. 1. In *International Semantic Web Conference*, pages 390–405. Springer, 2014. doi:10.1007/978-3-319-11915-1_25.

6 Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017. doi:10.3233/SW-160217.

7 Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338634.

8 Aidan Hogan. Web of data. In *The Web of Data*, pages 15–57. Springer, 2020. doi:10.1007/978-3-030-51580-5.

9 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for sparql. In *International Semantic Web Conference*, pages 258–275. Springer, 2019. doi:10.1007/978-3-030-30793-6_15.

10 Mahmoud Abo Khamis, Hung Q Ngo, Christopher Ré, and Atri Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)*, 41(4):1–45, 2016. doi:10.1145/2967101.

11 Eyal Kushilevitz. Communication complexity. In *Advances in Computers*, volume 44, pages 331–360. Elsevier, 1997. doi:10.1016/S0065-2458(08)60342-3.

12 Brian McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In *Handbook on ontologies*, pages 51–65. Springer, 2004.

13 Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. The odyssey approach for optimizing federated sparql queries. In *International semantic web conference*, pages 471–489. Springer, 2017. doi:10.1007/978-3-319-68288-4_28.

14 Gabriela Montoya, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. Federated sparql queries processing with replicated fragments. In *International Semantic Web Conference*, pages 36–51. Springer, 2015. doi:10.1007/978-3-319-25007-6_3.

15 Matthieu Mosser, Fernando Pieressa, Juan Reutter, Adrián Soto, and Domagoj Vrgoč. Querying apis with SPARQL: language and worst-case optimal algorithms. In *European Semantic Web Conference*, pages 639–654. Springer, 2018. doi:10.1007/978-3-319-93417-4_41.

16 Hung Q Ngo, Dung T Nguyen, Christopher Re, and Atri Rudra. Beyond worst-case analysis for joins with minesweeper. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 234–245, 2014. doi:10.1145/2594538.2594547.

17 Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*, pages 1–8. Association for Computing Machinery, 2015. doi:10.1145/2764947.2764948.

18 Muhammad Saleem, Alexander Potocki, Tommaso Soru, Olaf Hartig, and Axel-Cyrille Ngonga Ngomo. Costfed: Cost-based query optimization for sparql endpoint federation. *Procedia Computer Science*, 137:163–174, 2018. doi:10.1016/j.procs.2018.09.016.

19 Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *International semantic web conference*, pages 601–616. Springer, 2011. doi:10.1007/978-3-642-25073-6_38.

20 Dirk Van Gucht, Ryan Williams, David P Woodruff, and Qin Zhang. The communication complexity of distributed set-joins with applications to matrix multiplication. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 199–212, 2015. doi:10.1145/2745754.2745779.

21 Todd L Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012. doi:10.48550/arXiv.1210.0481.

22 Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Sam Coppens, Joaquim Gabarró Vallés, and Rik Van de Walle. Functional descriptions as the bridge between hypermedia apis and the semantic web. In *Proceedings of the third international workshop on restful design*, pages 33–40, 2012. doi:10.1145/2307819.2307828.

23 Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. Triple pattern fragments: a low-cost knowledge graph interface for the web. *Journal of Web Semantics*, 37:184–206, 2016. doi:10.1016/j.websem.2016.03.003.

24 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

A Appendix

A.1 Details from Section 4.2

For an atom $R(y_1,\ldots,y_k)$, and a constraint $\alpha = \langle a_1,\ldots,a_\ell,(l,h)\rangle$, the pair (α, R) dominates a tuple $\mathbf{t}$ if $a_i = \mathbf{t}_{s(i)}$ for each $1 \le i \le \ell - 1$, and $l \le \mathbf{t}_{s(\ell)} \le h$. A dominating constraint-relation pair is *infinite* when $h = \infty$. Algorithm 3 now presents how we compute the next probe point.

Algorithm 3 getProbePoint($\mathbf{t}$).

```
n ← arity($\mathbf{t}$)
**if** $\mathbf{t} = -1^n$ **then**
    **return** $1^n$
$\mathbf{t} \leftarrow (\mathbf{t}_1,\ldots,\mathbf{t}_{n-1},\mathbf{t}_n + 1)$
**while true do**
    **if** $\mathbf{t}$ is not dominated by any stored constraint-relation pair **then**
        **return t**
    $i \leftarrow$ the smallest position of a pair $(l,h)$ in a constraint dominating $\mathbf{t}$
    **if** $h = \infty$ **then**
        **if** i = 1 **then**
            **return** NULL
        $\mathbf{t}_j = 1$ for each $j \ge i$
        $\mathbf{t}_{i-1} = \mathbf{t}_{i-1} + 1$
    **else**
        $e \leftarrow$ The highest element $h$ in pairs $(l,h)$ in every constraint $\langle \mathbf{t}_1,\ldots,\mathbf{t}_{i-1},(l,h)\rangle$.
        $\mathbf{t}_i = e + 1$
        $\mathbf{t}_j = 1$ for each $j > i$
```

A.2 Counterexample for distributed Yannakakis and LFTJ

The following instance is presented in [16] to show that both Yannakakis and LFTJ cannot achieve instance-optimal runtime: this instance also provides an example where LFTJ communication is non-optimal.

Consider the query

$$Q \leftarrow R_1(a_1,a_2) \wedge R_2(a_2,a_3) \ldots \wedge R_m(a_m,a_{m+1}).$$

and the instance $I_{m,M}$ with $m \ge 5$ and M an integer, in which each R_i^I consists in exactly m blocks where the j-th block will be defined as follows

- for $j = i$ the block is just the tuple $\{((j-1)M+1,(j-1)M+1)\}$
- for $j = i-1$ (or m when $i = 1$) the block is empty
- for $j \in [k] - \{i, i-1\}$ the j-th block is

$$[(j-1)M+2, jM] \times [(j-1)M+2, jM]$$

The output of this instance is empty and it was shown in [16] that this instance has a certificate of size $O(mM)$ that consists on the following comparisons:

$$\begin{aligned} R_1[1,1] &< R_2[1] \\ R_1[i,1] &> R_2[1], \text{ for } i > 1 \\ R_2[i,1] &> R_3[M+1], \text{ for } i > 1 \\ &\vdots \\ R_{m-1}[i,1] &> R_m[(m-2)M+1], \text{ for } i > 1 \end{aligned}$$

As per Proposition 10 distributed minesweeper will need to communicate $O(mM)$ tuples with each R to compute the answer. Let's see now how can we solve this query with (a distributed version of the) LFTJ. Take an arbitrary attribute ordering $A_{i_1}, \dots, A_{i_n+1}$ and focus on the first two attributes A_{i_1} and A_{i_2}:

In order to be able to compare the communication complexity of both our distributed Minesweeper and a federated version of LFTJ we will endow the latter with a cache and communication-free semi-join reductions and let's consider two cases:

1. The first one is when $|i_1 - i_2| = 1$. In this case, the algorithm will compute the reduction of the corresponding relation $R(A_{i_1}, A_{i_2})$ on A_{i_1} and A_{i_2} which we assume involves no communication. But after that it needs to go through (and communicate) all tuples in the reduced relation that is of size $\Omega(mM^2)$
2. On the other hand, when $|i_1 - i_2| > 1$ then the first thing is to compute the intersections on both attributes A_{i_1} and A_{i_2} and perform a cross-product. Both intersections are of size $\Omega(mM)$ and even though communicating the tuples to compute the cross-product itself is not an issue, in the next step we will necessarily communicate $\Omega(mM^2)$

Now let's move into the Yannakakis algorithm: the idea here is to exploit the fact that Yannakakis needs to perform semi-joins. Consider for example the following acyclic query Q

$$Q \leftarrow S(x) \wedge R(x, y) \wedge U(y, w) \wedge T(x, z) \wedge V(z, p).$$

And, for each integer m, the instance I_m in which $S^{I_m} = [m]$, $T^{I_m} = [1]x[m]$, $R^{I_m} = [2]x[n]$, and $U^{I_m} = V^{I_m} = [m]x[a]$. Notice that the certificate of I_m is always of size 1, comprising the sole argument $T[-1] < R[0]$.

While there are several version of the Yannakakis algorithm, all versions we are aware of involve a bottom-up reduction of the database. This implies computing the semijoin between R and U and T and V. However, computing $R \rtimes U$ or $U \rtimes R$ involves the intersection of the second component of R with the first component of U, which requires the communication of n bits, as per standard communication complexity results (see e.g. [11]).

The results above can be summarized into the following observations:

▶ **Observation 16.** *There is an acyclic query Q and a family (I_m), $m > 1$ of instances whose certificate for Q is of size $O(mM)$, but for which the Leapfrog trie join algorithm must necessarily communicate $O(mM^2)$ bits.*

▶ **Observation 17.** *There is an acyclic query Q and a family (I_m), $m > 1$ of instances whose certificate for Q is of size 1, but for which the Yannakakis algorithm must necessarily communicate m bits.*