Col-Graph: Towards Writable and Scalable Linked Open Data

Luis-Daniel Ibanez, Hala Skaf-Molli, Pascal Molli, Olivier Corby

Col-Graph: Towards Writable and Scalable Linked Open Data

Luis-Daniel Ibáñez[1], Hala Skaf-Molli[1], Pascal Molli[1], and Olivier Corby[2]

[1] LINA, University of Nantes
{luis.ibanez,hala.skaf,pascal.molli}@univ-nantes.fr
[2] INRIA Sophia Antipolis-Méditerranée
olivier.corby@inria.fr

Abstract. Linked Open Data faces severe issues of scalability, availability and data quality. These issues are observed by data consumers performing federated queries; SPARQL endpoints do not respond and results can be wrong or out-of-date. If a data consumer finds an error, how can she fix it? This raises the issue of the *writability* of Linked Data. In this paper, we devise an extension of the federation of Linked Data to data consumers. A data consumer can make partial copies of different datasets and make them available through a SPARQL endpoint. A data consumer can update her local copy and share updates with data providers and consumers. Update sharing improves general data quality, and replicated data creates opportunities for federated query engines to improve availability. However, when updates occur in an uncontrolled way, consistency issues arise. In this paper, we define fragments as SPARQL CONSTRUCT federated queries and propose a correction criterion to maintain these fragments incrementally without reevaluating the query. We define a coordination free protocol based on the counting of triples derivations and provenance. We analyze the theoretical complexity of the protocol in time, space and traffic. Experimental results suggest the scalability of our approach to Linked Data.

1 Introduction

The Linked Open Data initiative (LOD) makes millions of RDF-triples available for querying through a federation of SPARQL endpoints. However, the LOD faces major challenges including availability, scalability [3] and quality of data [1].

These issues are observed by data consumers when they perform federated queries; SPARQL endpoints do not respond and results can be wrong or out-of-date. If a data consumer finds a mistake, how can she fix it? This raises the issue of the *writability* of Linked Data, as already pointed out by T. Berners-Lee [2].

We devise an extension of Linked Data with data replicated by Linked Data consumers. Consumers can perform intensive querying and improve data quality on their local replicas. We call replicated subsets of data, *fragments*. First, any participant creates fragments from different data providers and make them available to others through a regular SPARQL Endpoint. Local fragments are

writable, allowing modifications to enhance data quality. Original data providers can be contacted to consume local changes in the spirit of *pull requests* in Distributed Version Control Systems (DVCS). Second, the union of local fragments creates an opportunistic replication scheme that can be used by federated query engines to improve data availability [13,17]. Finally, update propagation between fragments is powered by live feeds as in DBpedia Live [14] or sparqlPuSH [16].

Scientific issues arise concerning the consistency of these fragments. These questions have been extensively studied in Collaborative Data Sharing Systems (CDSS) [11], Linked Data with adaptations of DVCS [18,4] and replication techniques [10,25]. Existing approaches follow a total replication approach, *i.e.*, full datasets or their full histories are completely replicated at each participant or they require coordination to maintain consistency.

In this paper, we propose Col-Graph, a new approach to solve the availability, scalability and writability problems of Linked Data. In Col-Graph, we define fragments as SPARQL CONSTRUCT federated queries, creating a *collaboration network*, propose a consistency criterion and define a coordination-free protocol to maintain fragments incrementally without reevaluating the query on the data source. The protocol counts the derivations of triples for data synchronization and keeps provenance information to make decisions in case of a conflict.

We analyze the protocol's complexity and evaluate experimentally its efficiency. The main factors that affect Col-Graph performance are the number of concurrent insertions of the same data, the connectivity of the collaboration network and the overlapping between the fragments. Experimentations show that the overhead of storing counters is less than 6% of the fragment size, whenever there are up to 1,000 concurrent insertions or up to 10×10^{16} simple paths between the source and the dataset. Synchronization is faster than fragment reevaluation up to when 30% of the triples are updated. We also report better performance on synthetically generated social networks than on random ones.

Section 2 describes Col-Graph general approach and defines the correction criterion. Section 3 formalizes Col-Graph protocol. Section 4 details the complexity analysis. Section 5 details experimentations. Section 6 summarizes related work. Finally, section 7 presents the conclusions and outlines future work.

2 Col-Graph Approach and Model

In Col-Graph, consumers create *fragments*, *i.e.*, partial copies of other datasets, based on simple federated CONSTRUCT queries, allowing them to perform intensive queries locally on the union of fragments and make updates to enhance data quality. In Figure 1, $Consumer\_1$ copies fragments from DBPedia and DrugBank, $Consumer\_2$ copies fragments from DBPedia and MusicBrainz and $Consumer\_3$ copies fragments from $Consumer\_2$ and $Consumer\_3$.

Consumers publish the updated dataset, allowing others to also copy fragments from them. They can also contact their data sources to ask them to incorporate their updates, in the spirit of DVCS pull requests. Updates at the fragment's source are propagated to consumers using protocols like sparqlPuSH [16]

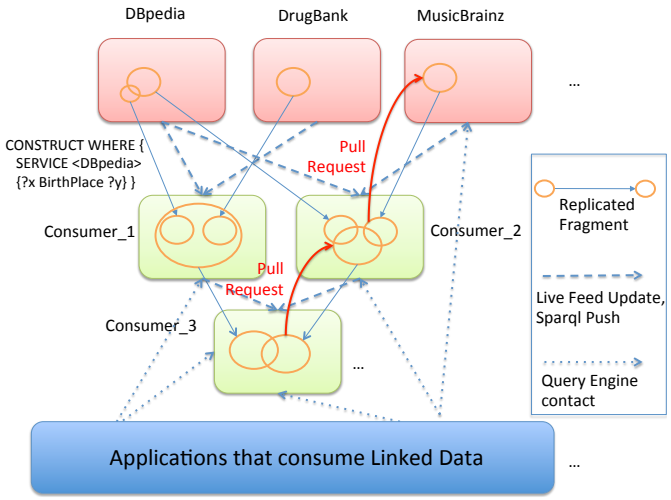


Fig. 1: Federation of Writable Linked Data

or live feeds [14]. As replicated fragments could exist on several endpoints, adequate federated query engines could profit to improve general data availability and scalability [13,17]. Following this approach, data providers can share the query load and the data curation process with data consumers. Since data consumers become also data providers, they can gain knowledge of queries targeting their fragments.

We consider that each Linked Data participant holds one RDF-Graph and exposes a SPARQL endpoint. For simplicity, we use P to refer to the RDF-Graph, the SPARQL endpoint or the name of a participant when is not confusing. An RDF-Graph is defined as a set of triples (s, p, o), where s is the subject, p is the predicate and o is the object. We suppose that a participant wants to copy *fragments* of data from other participants, *i.e.,* needs to copy a subset of their RDF-Graphs for a specific application [19] as in Figure 1.

Definition 1 (Fragment). *Let S be a SPARQL endpoint of a participant, a fragment of the RDF-Graph published by S, $F[S]$, is a SPARQL CONSTRUCT federated query [22] where all graph patterns are contained in a single SERVICE block with S as the remote endpoint. We denote as $eval(F[S])$ the RDF-Graph result of the evaluation of $F[S]$.*

A fragment $F[S]$ enables a participant T to make a copy of the data of S that answers the query. We denote the result of the evaluation of $F[S]$ materialized by a participant T as $F[S]@T$, *i.e.*, a fragment of *source* S materialized at *target* T. A fragment is partial if $F[S]@T \subset S$ or full if $F[S]@T = S$. The local

data of a participant is composed of its own data union the fragments copied from other participants. We call the directed labeled graph where the nodes are the participants and the edges $(S;T)$ labeled with fragments a *Collaboration Network*, CN. A CN defines how data are shared between participants and how updates are propagated. Participants can query and update the fragments they materialize, *e.g.*, $Consumer\_1$ in figure 1 can modify the fragments copied from DBPedia and DrugBank using SPARQL 1.1 Update [23]

When a source in a CN updates its data, the materialized fragments may become outdated. Fragments could be re-evaluated at the data source, but if the data source has a popular knowledge base*, i.e.,* many other participants have defined fragments on it, the continuous execution of fragments would decrease the availability of the source's endpoint. To avoid this, a participant may synchronize its materialized fragment *incrementally* by using the updates published by the source. Some popular data providers such as DBpedia Live [14] and MusicBrainz[3] publish live update feeds.

To track updates done by a participant, we consider an RDF-triple as the smallest directly manageable piece of knowledge [15] and the insertion and deletion of an RDF-triple as the two basic types of updates. Each update is *globally uniquely identifiable* and it turns the RDF-Graph into a new state. SPARQL 1.1 updates are considered as an ordered set of deleted and/or inserted triples. Each time we refer to an update, we implicitly refer to the inserted/deleted triple. Blank nodes are considered to be skolemized, *i.e.*, also globally identifiable[4].

Incrementally synchronizing a materialized fragment using only the updates published by a data source and the locally materialized fragment without reevaluating the fragment on the data source requires to exclude join conditions from fragments [8], therefore, we restrict to *basic fragments* [21]*, i.e.,* fragments where the query has only one triple pattern.

Figure 2 illustrates a CN and how updates are propagated on it. $P1$ starts with data about the *nationality* and *KnownFor* properties of M_Perey (prefixes are omitted for readability). $P2$ materializes from $P1$ all triples with the *knownFor* property. With this information and its current data, $P2$ inserts the fact that M_Perey discovered Francium. On the other hand, $P3$ materializes from $P1$ all triples with the *nationality* property. $P3$ detects a mistake (nationality should be $French$, not $French\_People$) and promptly corrects it. $P4$ constructed a dataset materializing from $P2$ the fragment of triples with the property *discoverer* the fragment of triples with the property *nationality* from $P3$. $P1$ trusts $P4$ about data related to M_Perey, so she materializes the relevant fragment, indirectly consuming updates done by $P2$ and $P3$.

Updates performed on materialized fragments are not necessarily integrated by the source, *e.g*, the deletion done by $P3$ did not reach $P1$, therefore, equivalence between source and materialized fragment cannot be used as consistency criterion for CNs. Intuitively, each materialized fragment must be equal to the

[3] `http://musicbrainz.org/doc/MusicBrainz_Database#Live_Data_Feed`

[4] `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-skolemization`

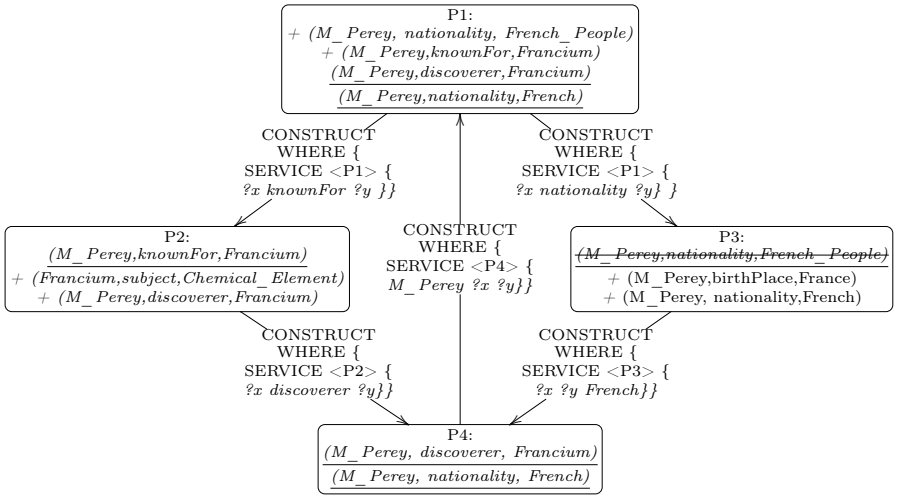


Fig. 2: Collaboration network with Basic Fragments. Underlined triples are the ones coming from fragments, triples preceded by a '+' are the ones locally inserted, struck-through triples are the ones locally deleted.

evaluation of the fragment at the source after applying *local* updates, *i.e.*, the ones executed by the participant itself and the ones executed during synchronization with other fragments.

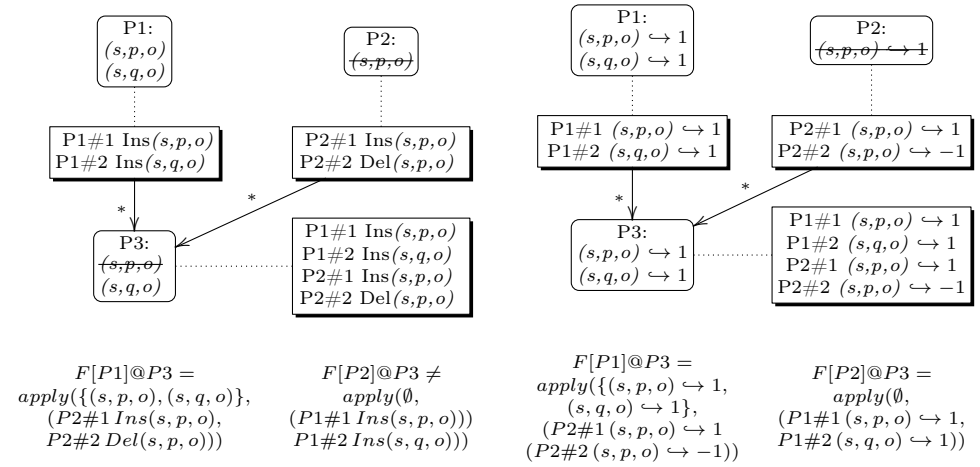
Definition 2 (Consistency Criterion). *Let $CN = (P, E)$ be a collaboration network. Assume each $P_i \in P$ maintains an ordered set of uniquely identified updates Δ_{P_i} with its local updates and the updates it has consumed from the sources of the fragments $F[P_j]@P_i$ it materializes. Given a Δ_P, let $\Delta_P^{F[S]}$ be the ordered subset of Δ_P such that all updates* concern *$F[S]$,* i.e.*, that match the graph pattern in $F[S]$. Let $apply(P_i, \Delta)$ be a function that applies an ordered set of updates Δ on P_i.*

CN is consistent *iff when the system is idle, i.e., no participant executes local updates or fragment synchronization, then:*

$$(\forall P_i, P_j \in P : F[P_i]@P_j = apply(eval(F[P_i]), \Delta_{P_j}^{F[P_i]} \setminus \Delta_{P_i})$$

The $\Delta_{P_j}^{F[P_i]} \setminus \Delta_{P_i}$ term formalises the intuition that we need to consider only local updates when evaluating the consistency of each fragment, *i.e.*, from the updates concerning the fragment, remove the ones coming from the source.

Unfortunately, applying remote operations as they come does not always comply with Definition 2 as shown in Figure 3a: $P3$ synchronizes with $P1$, applying the updates identified as $P1\#1$ and $P1\#2$, then with $P2$, applying the updates identified as $P2\#1$ and $P2\#2$, however, the fragment materialized from $P2$ is not consistent. Notice that, had $P3$ synchronized with $P2$ before than with $P1$, its final state would be different ((s, p, o) would exist) and the fragment materialized from $P1$ would not be consistent.



(a) Applying updates as they come does not comply with the correction criterion.

(b) The Annotated RDF-Graph enables a consistent Collaboration Network

Fig. 3: Illustration of the consistency criterion. Rounded boxes represent the graphs, and shaded boxes the sequences of updates. $*$ represents a full fragment.

3 A Protocol for Synchronization of Basic Fragments

To achieve consistency in every case, we propose, in the spirit of [7], to count the number of insertions and deletions of a triple, *i.e.,* we annotate each RDF-triple with positive or negative integers, positive values indicate insertions and negative values deletions. This allows a uniform representation of data and updates, yielding a simple way to synchronize fragments.

Definition 3 (Annotated RDF-triple, Graph and Update).

1. *Let t an RDF-triple and $z \in \mathbb{Z}^*$. $t \hookrightarrow z$ is an* annotated RDF-triple*, with t being the* triple *and z the* annotation*.*
2. *An* annotated RDF-Graph G^A *is a set of annotated RDF-triples such that* $(\forall t, z | t \hookrightarrow z \in G^A : z > 0)$
3. *An* annotated update u^A *is represented by an annotated RDF-triple. More precisely, $t \hookrightarrow 1$ for insertion of t and $t \hookrightarrow -1$ for deletion of t.*

Annotations in RDF-Graphs count the number of *derivations* of a triple. An annotation value higher than one indicates that the triple exists in more than one source or there are several paths in CN leading from the source of the triple to the participant. Annotations in updates indicate, if positive, that z derivations of t were inserted; if negative, that z derivations of t were deleted. For example, an annotated RDF-triple $t_1 \hookrightarrow 2$ means that either t_1 has been inserted by two different sources or the same insert arrived through two different paths in the CN. The annotated update $t_2 \hookrightarrow -1$ means that t_2 was deleted at one source

or by some participant in the path between the source and the target; $t_3 \hookrightarrow -2$ means that either t_3 was deleted by two sources or by some participant in the path between two sources and the target.

To apply annotated updates to annotated RDF-Graphs, we define an *Update Integration* function:

Definition 4 (Update Integration). *Let A the set of all annotated RDF-Graphs and B the set of all annotated updates. Assume updates arrive from source to target in FIFO order. The* Update Integration *function $\uplus : A \times B \to A$ takes an annotated RDF-Graph $G^A \in A$ and an annotated update $t \hookrightarrow z \in B$:*

$$G^A \uplus t \hookrightarrow z = \begin{cases} G^A \cup \{t \hookrightarrow z\} & if\ (\nexists w : t \hookrightarrow w \in G^A) \\ G^A \setminus \{t \hookrightarrow w\} & if\ t \hookrightarrow w \in G^A \wedge w + z \leq 0 \\ (G^A \setminus \{t \hookrightarrow w\}) \cup \{t \hookrightarrow w + z\} & if\ t \hookrightarrow w \in G^A \wedge w + z > 0 \end{cases}$$

The first piece of the Update Integration function handles incoming updates of triples that are not in the current state. As we are assuming FIFO in the update propagation from source to target, insertions always arrive before corresponding deletions, therefore, this case only handles insertions. The second piece handles deletions, only if the incoming deletion makes the annotation zero the triple is deleted from the current state. The third piece handles deletions that do not make the annotation zero and insertions of already existing triples by simply updating the annotation value.

We now consider each participant has an annotated RDF-Graph G^A and an ordered set of annotated updates U^A. SPARQL queries are evaluated on the RDF-Graph $\{t \,|\, t \hookrightarrow z \in G^A\}$. SPARQL Updates are also evaluated this way, but their effect is translated to annotated RDF-Graphs as follows: the insertion of t to the insertion of $t \hookrightarrow 1$ and the deletion of t to the deletion of the annotated triple having t as first coordinate. Specification 1.1 details the methods to insert/delete triples and synchronize materialized fragments. Figure 3b shows the fragment synchronization algorithm in action. A proof of correctness follows the same case-base analysis developed to prove [10].

3.1 Provenance for Conflict Resolution

In section 3 we solved the problem of consistent synchronization of basic fragments. However, our consistency criterion is based on the mere existence of triples, instead of on the possible conflicts between triples coming from different fragments and the ones locally inserted. Col-Graph's strategy in this case is that each participant is responsible for checking the semantic correctness of its dataset, as criteria often varies and what is semantically wrong for one participant, could be right for another. Participants can delete/insert triples to fix what they consider wrong. Participants that receive these updates can edit in turn if they do not agree with them.

In the event of two triples being semantically incompatible, the main criteria to choose which one of them delete is the *provenance* of the triples. With this

```
Annotated Graph G^A,
Ordered Set ΔP_ID

void insert(t):
    pre: t ∉ {t'|t ↪ x ∈ G^A}
    G^A := G^A ∪ t ↪ 1
    Append(ΔP_ID, t ↪ 1)

void delete(t):
    pre: t ∈ {t'|t' ↪ x ∈ G^A}
    G^A := G^A ⊎ t ↪ −z
    Append(ΔP_ID, t ↪ −z)

void sync(F[P_x], ΔP_x):
    for t ↪ z ∈ ΔP_x:
        if t ↪ z ∉ ΔP_ID:
            G^A := G^A ⊎ t ↪ z
            Append(ΔP_ID, t ↪ z)
```

Specification 1.1: Class Participant when triples are annotated with elements of Z.

```
IRI P_ID,
Annotated Graph G^A,
Ordered Set ΔP_ID

void insert(t):
    pre: t ∉ {t'|t ↪ x ∈ G^A}
    G^A := G^A ∪ t ↪ P_ID
    Append(ΔP_ID, t ↪ P_ID)

void delete(t):
    pre: t ∈ {t'|t' ↪ x ∈ G^A}
    G^A := G^A ⊎ t ↪ −m
    Append(ΔP_ID, t ↪ −m)

void sync(F[P_x], ΔP_x):
    for t ↪ m ∈ ΔP_x:
        if t ↪ m ∉ ΔP_ID:
            G^A := G^A ⊎ t ↪ m
            Append(ΔP_ID, t ↪ m)
```

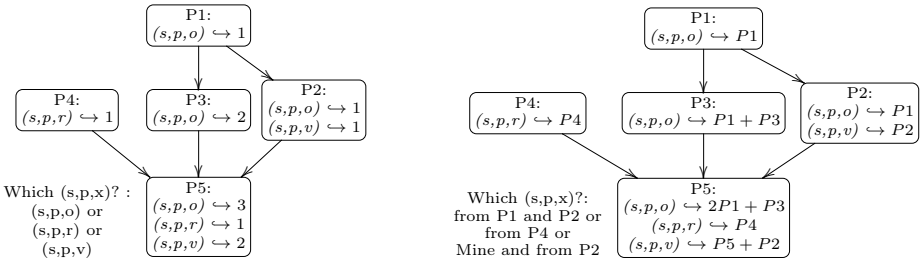
Specification 1.2: Class Participant when triples are annotated with elements of the monoid M.

information, the decision can be made based on the trust on its provenance. As in [11], we propose to substitute the integer annotations of the triple by an element of a commutative monoid that embeds $(Z, +, 0)$. We recall that a commutative monoid is an algebraic structure comprised by a set K, a binary, associative, commutative operation $\oplus$ and an identity element $0_K \in K$ such that $(\forall k \in K \,|\, k \oplus 0_K = k$; a monoid $M = (K, \oplus, 0_K)$ embeds another monoid $M' = (K', \oslash, 0_{K'})$ iff there is a map $f : K \to K'$ such that $f(0_K) = f(0_{K'})$ and $(\forall a, b \in K : f(a \oplus b) = f(a) \oslash f(b))$. If we annotate with a monoid that embeds $(Z, +, 0)$, only a minor change is needed in our synchronization algorithm to achieve consistency. This monoid is used to encode extra provenance information.

Definition 5. *Assume each participant in the collaboration network has a unique ID, and let X be the set of all of them. Let $M = (Z[X], \oplus, 0)$ be a monoid with:*

1. *The identity 0.*
2. *The set $Z[X]$ of polynomials with coefficients in Z and indeterminates in X.*
3. *The polynomial sum $\oplus$, for each monomial with the same indeterminate: $aX \oplus bX = (a + b)X$*
4. *M embeds $(Z, +, 0)$ through the function $f(a_1X_1 \oplus \cdots \oplus a_nX_n) = \sum_1^n a_i$*

Each time a participant inserts a triple, she annotates it with its ID with coefficient 1. The only change in definition 4 is the use of $\oplus$ instead of $+$. Speci-



(a) Without provenance, P5 only information is the number of derivations. She does not know the author of the facts.

(b) With provenance, P5 also knows who inserted what and if it was concurrent, enabling trust based decisions to solve conflicts.

Fig. 4: Difference between annotating with Z (4a) versus annotating with M (4b). All fragments are full.

fication 1.2 describes the algorithm to insert/delete triples and synchronize fragments with triples annotated with elements of M.

When annotating with Z, the only information encoded in triples is their number of derivations. M adds (i) Which participant is the *author* of the triple. A triple stored by a participant P with an annotation comprised by the sum of n monomials indicates that the triple was inserted *concurrently* by n participants from which there is a path in CN to P. (ii) The number of paths in the Collaboration Network in which all edges concern the triple, starting from the author(s) of the triple to this participant, indicated by the coefficient of the author's ID.

Figure 4 compares annotations with Z versus annotations with M. In the depicted collaboration network, the fact *(s,p,o)* is inserted concurrently by P1 and P3, *(s,p,v)* is inserted concurrently by P2 and P5 and *(s,p,r)* inserted only by P4. When the synchronization is finished, P5 notices that it has three triples with s and p as subject and predicate but different object values. If P5 wants to keep only one of such triples based on trust, the Z annotations (4a) do not give her enough information, while the M annotations (4b) give more information for $P5$ to take the right decision. She can know that the triple (s, p, o) was inserted by two participants $P1$ and $P3$, while (s, p, r) was only inserted by $P4$ and that (s, p, v) was inserted by $P2$ and herself.

4 Complexity Analysis

In this section, we analyze the complexity in time, space and traffic of RDF-Graphs annotated with M and their synchronization, to answer the question: *how much does it cost to guarantee the correctness of a collaboration network?*.

Concerning time complexity, from specifications 1.1 and 1.2, we can see that for the insert and delete methods is constant. For the synchronization of a fragment $F[P_x]@P_y$, the complexity is $n(x_1 + x_2)$ where n is the number of incoming updates, x_1 the complexity of checking if an update is in ΔP_y (which can be

considered linear) and x_2 the complexity of the $\uplus$ function. For Z annotations, $\uplus$ is constant, for M is linear on the size of the largest polynomial.

Concerning space complexity, the overhead is the size of the annotations. For an annotated triple t at a participant P, the relevant factors are: (i) the set of participants that concurrently inserted t from which there is a path to P such that all edges concern t, that we will denote β_t (ii) the number of paths to P in the collaboration network from the participants $P_1 \dots P_n$ that concurrently inserted t such that all edges concern t. For a participant P_i, we denote this number as $\rho_{t \leftarrow P_i}$. Let $sizeOf$ be a function that returns the space needed to store an object. Assume that the cost of storing ids is a constant ω. Then, for $t \hookrightarrow z, z \in Z[x]$ we have $sizeOf(z) = |\beta_t|\omega + \sum_{P_i \in \beta_t} sizeOf(\rho_{t \leftarrow P_i})$. Therefore, for each triple we need to keep a hash map from ids to integers of size $|\beta_t|$. The worst case for $|\beta_t|$ is a *strongly connected* Collaboration Network CN where all participants insert t concurrently, yielding an array of size $|CN|$. The worst case for $\rho_{t \leftarrow P_i}$ is a *complete network*, as the number of different simple paths is maximal and in the order of $|CN|$!

The size of the log at a participant P depends on two factors (i) the *dynamics* of P, *i.e.*, the number of local updates it does. (ii) the *dynamics* of the fragments materialized by P, i.e., the amount of updates at the sources that concern them.

In terms of the number of messages exchanged our solution is optimal, only one contact with the update log of each source is needed. In terms of message size, the overhead is in principle the same as the space complexity. However, many compression techniques could be applied.

The solution described so far uses an update log that is never purged. Having the full update history of a participant has benefits like enabling *historical queries* and *version control*. However, when space is limited and/or updates occur often, keeping such a log could not be possible. To adapt our solution to data feeds we need to solve two issues: (i) How participants materialize fragments for the first time? (ii) How to check if an incoming update has been already received?

To solve the first issue, an SPARQL extension that allows to query the annotated RDF-Graph and return the triples *and* their annotations is needed, for example the one implemented in [24]. To solve the second issue, we propose to add a second annotation to updates, containing a set of participant identifiers ϕ_u representing the participants that have already received and applied the update. When an update u is created, ϕ_u is set to the singleton containing the ID of the author, when u is pushed downstream, the receiving participant checks if his ID is in ϕ_u, if yes, u has already been received and is ignored, else, it is integrated, and before pushing it downstream it adds its ID to ϕ_u. Of course, there is a price to pay in traffic, as the use of ϕ increases the size of the update. The length of ϕ_u is bounded by the length of the longest simple path in the Collaboration-Network, which in turn is bounded by the number of participants.

To summarize, the performance of our solution is mainly affected by the following properties of the CN: (i) The probability of concurrent insertion of the same data by many participants. The higher this probability, the number of terms of the polynomials is potentially higher. (ii) Its *connectivity*. The more

connected, the more paths between the participants and the potential values of ρ are higher. If the network is poorly connected, few updates will be consumed and the effects of concurrent insertion are minimized. (iii) The *overlapping* between the fragments. If all fragments are full, all incoming updates will be integrated by every participant, maximizing the effects of connectivity and concurrent insertion. If all fragments are disjoint, then all updates will be integrated only once and the effects of connectivity and concurrent insertion will be neutralized.

5 Experimentations

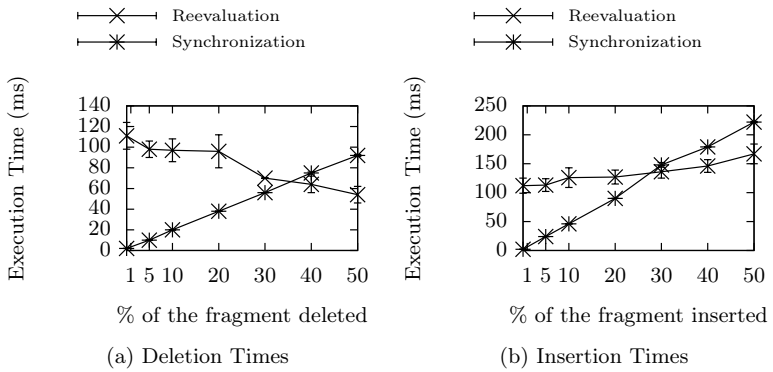
We implemented specification 1.2 on top of the SPARQL engine Corese[5] v3.1.1. The update log was implemented as a list of updates stored in the file system. We also implemented the ϕ annotation described in section 4 to check for double reception. We constructed a test dataset of 49999 triples by querying the DBpedia 3.9 public endpoint for all triples having as object the resource `http://dbpedia.org/resource/France`. Implementation, test dataset, and instructions to repeat the experiments are freely available[6].

Our first experiment studies the execution time of our synchronization algorithm. The goal is to confirm the linear complexity derived in section 4 and to check its cost w.r.t fragment re-evaluation. We defined a basic fragment with the triple pattern *?x :ontology/birthPlace ?z* (7972 triples 15% of the test dataset's size). We loaded the test dataset in a source, materialized the fragment in a target and measured the execution time when inserting and when deleting 1, 5, 10, 20, 30, 40 and 50% of triples concerning the fragment. As baseline, we set up the same datasets on two RDF-Graphs and measured the time of clearing the target and re-evaluating the fragment. Both source and target were hosted on the same machine to abstract from latency.

We used the Java MicroBenchmark Harness[7] v. 0.5.5 to measure the average time of 50 executions across 10 JVM forks with 50 warm-up rounds, for a total of 500 samples. Experiments were run on a server with 20 hyperthreaded cores with 128Gb of ram an Linux Debian Wheezy. Figure 5 shows a linear behaviour, consistent with the analysis in section 4. Synchronization is less expensive than re-evaluation up to approx. 30% of updates. We believe that a better implementation that takes full advantage of streaming, as Corese does by processing data in RDF/XML, could improve performance. Basic fragments are also very fast to evaluate, we expect than in future work, when we can support a broader class of fragments, update integration will be faster in most cases.

Our second experiment compares the impact on annotation's size produced by two of the factors analyzed in section 4: concurrent insertions and collaboration network connectivity, in order to determine which is more significant. We loaded the test dataset in: (i) An RDF-Graph. (ii) An annotated RDF-Graph, simulating n concurrent insertions of all triples, at n annotated RDF-Graphs

[5] `http://wimmics.inria.fr/corese`
[6] `https://code.google.com/p/live-linked-data/wiki/ColGraphExperiments`
[7] `http://openjdk.java.net/projects/code-tools/jmh/`



(a) Deletion Times (b) Insertion Times

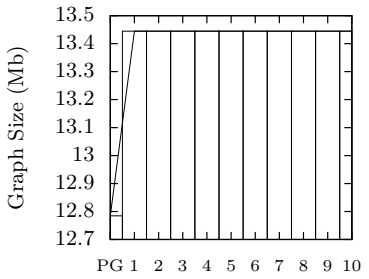
Fig. 5: Comparison of execution time (ms) between synchronization and fragment reevaluation. Error bars show the error at 95%.

with id `http://participant.topdomain.org/$i$`, with $i \in [0, n]$ (iii) An annotated RDF-Graph, simulating the insertion of all triples in other graph with id "http://www.example.org/participant", arriving to this one through m different simple paths, and measured their size in memory on a Macbook Pro running MacOS Lion with java 1.7.0_10-ea-b13 and Java HotSpot(TM) 64-Bit Server VM (build 23.6-b04, mixed mode).

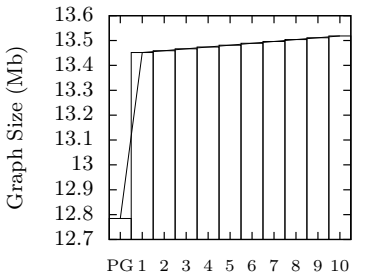
Figure 6 shows the results. Both cases represent nearly the same overhead, between 5 and 6 percent. Concurrency makes annotation's size grow sub-linearly. With respect to path number, annotation's size grows even slower , however, after 10×10^{17} paths, the *long* type used in our implementation overflows, meaning that in scenarios with this level of connectivity, the implementation must use BigInt arithmetics. In conclusion, after paying the initial cost of putting annotations in place, Col-Graph can tolerate a high number of concurrent inserts and a high network connectivity.

The goal of our final experiment is to study the effect of network's topology on Col-Graph's annotation's size. We argue that the act of materializing fragments and sharing updates is socially-driven, therefore, we are interested in analyzing the behaviour of Col-Graph on social networks. We generated two sets of 40 networks with 50 participants each, all edges defining full fragments, one following the random Ërdos-Renyi model [5] and other following the social-network oriented Forest Fire model [12]. Each networkset is composed of 4 subsets of 10 networks with densities $\{0.025, 0.05, 0.075, 0.1\}$. Table 1 shows the average of the average node connectivity of each network set. Social networks in are less connected than random ones, thus, we expect to have better performance.

We loaded the networks on the Grid5000 platform (`https://www.grid5000.fr/`) and made each participant insert the same triple to introduce full concurrency, thus, fixing the overlapping and concurrency parameter in their worst case. Then, we let them synchronize repeatedly until quiescence with a 1 hour timeout. To detect termination, we implemented the most naive algorithm: a



(a) Simulation of one insertion arriving through multiple paths.



(b) Simulation of concurrent insertions arriving through one path.

Fig. 6: Space Overhead of the Annotated Graph w.r.t a plain graph (PG). Both Concurrency and Connectivity represent approx. 6% of overhead each.

central overlord controls the execution of the whole network . We measured the maximum and average coefficient values and the maximum and average number of terms of annotations.

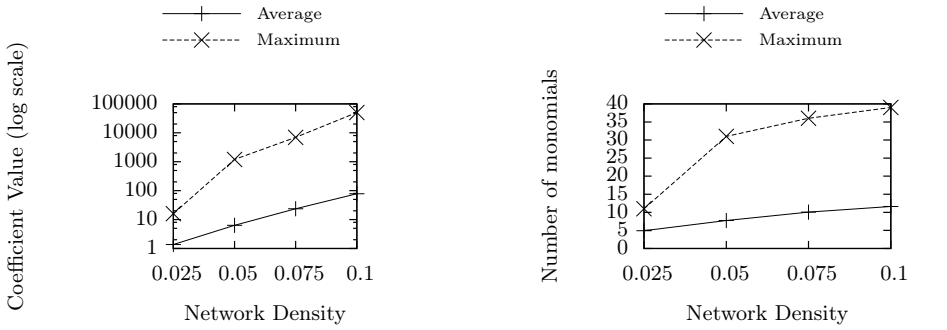
Figure 7 shows the results for Forest Fire networks. The gap between the average and maximum values indicates that topology has an important effect: only few triples hit high values. From the Ërdos-Renyi dataset, only networks with density 0.025 and finished before timeout without having a significant difference with their ForestFire counterparts. These results suggest that high connectivity affects the time the network takes to converge, and, as the number of rounds to converge is much higher, the coefficient values should also be much higher. We leave the study of convergence time and the implementation of a better termination detection strategy for future work.

	0.025	0.05	0.075	0.1
Forest Fire	0.0863	0.2147	0.3887	0.5543
Ërdos-Renyi	0.293	1.3808	2.5723	3.7378

Table 1: Average node connectivities of the experimental network sets

6 Related Work

Linked Data Fragments (LDF) [21] proposes data fragmentation and replication as an alternative to SPARQL endpoints to improve availability. Instead of answering a complex query, the server publishes a set of fragments that corresponds to specific triple patterns in the query, offloading to the clients the responsibility of constructing the result from them. Col-Graph allows clients to define the



(a) Average and maximum coefficient value

(b) Average and maximum number of terms

Fig. 7: Performance of the synchronization algorithm when applied on networks generated with the Forest Fire model

fragments based on their needs, offloading also the fragmentation. Our proposal also solves the problem of *writability*, that is not considered by LDF.

[4,18] adapt the Distributed Version Control Systems Darcs and Git principles and functionality to RDF data. Their main goal is to provide versioning to Linked Data, they do not consider any correctness criterion when networks of collaborators copy fragments from each other, and they do not allow fragmentation, *i.e.,* the full repository needs to be copied each time.

[10,25] use *eventual consistency* as correctness criterion. However, this requires that all updates to be eventually delivered to all participants, which is not compatible with fragmentation nor with socially-generated collaboration network. [19] proposes a partial replication of RDF graph for mobile devices using the same principles of SVN with a limited lifetime of local replica checkout-commit cycle. Therefore, it is not possible to ensure synchronization of partial copies with the source since a data consumer has to checkout a new partial graph after committing changing to the data provider.

[9] formalizes an OWL-based syndication framework that uses description logic reasoning to match published information with subscription requests. Similar to us, they execute queries incrementally in response to changes in the published data. However, in their model consumers do not update data, and connection between consumers and publishers depends on *syndication brokers*.

Provenance models for Linked Data using annotations have been studied in [26,6] and efficiently implemented in [24], showing several advantages with respect to named graphs or reification. The model in [6] is based on provenance polynomials and is more general than the one we used, however, as basic fragments are a restricted class of queries, the M monoid suffices.

Collaborative Data Sharing Systems (CDSS) like Orchestra [11] use Z-Relations and provenance to solve the data exchange problem in relational databases. CDSS have two requirements that we do not consider: support for full relational algebra in the fragment definition and strong consistency. However, the price to

pay is limited scalability and the need of a global ordering on the synchronization operation, that becomes blocking [20]. Both drawbacks are not compatible with Linked Data requirements of scalability and participant's autonomy. Col-Graph uses the same tools to solve the different fragment synchronization problem, with an opposite trade-off: scalability and autonomy of participants in exchange of weaker consistency and limited expressiveness of fragment definitions.

7 Conclusions and Future Work

In this paper, we proposed to extend Linked Data federation to data consumers in order to improve its availability, scalability and data quality. Data consumers materialize fragments of data from data providers and put them at disposal of other consumers and clients. Adequate federated query engines can use these fragments to balance the query load among federation members. Fragments can be updated to fix errors, and these updates can be consumed by other members (including the original sources) to collaboratively improve data quality.

We defined a consistency criterion for networks of collaborators that copy fragments from each other and designed an algorithm based on annotated RDF-triples to synchronize them consistently. We analyzed the complexity of our algorithm in time, space and traffic, and determined that the main factors that affect performance are the probability of concurrent insertion, the connectivity of the collaboration network and the fragment overlapping.

We evaluated experimentally the incurred overhead using a 50k real dataset on our open source implementation, finding that in space, concurrency and connectivity represent approximately 6% of overhead each, and that it grows sub-linearly; in time, our algorithm is faster than the reevaluation up to 30% of updated triples without taking in account latency. We also found that our algorithm performs better in socially generated networks than in random ones.

Future works include a large scale evaluation of Col-Graph focused on the effect of fragment overlapping, and using real dataset dynamics. We also plan to benchmark replication-aware federated query engines [13,17] on collaboration networks using Col-Graph to quantify the availability boost of our solution, and extend our model to handle dynamics on the fragment definitions themselves.

Acknowledgements: Some of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. This work is supported by the French National Research agency (ANR) through the KolFlow project (code: ANR-10-CONTINT-025).

References

1. Acosta, M., Zaveri, A., Simperl, E., Kontokostas, D., Auer, S., Lehmann, J.: Crowdsourcing linked data quality assessment. In: ISWC (2013)
2. Berners-Lee, T., O'Hara, K.: The read-write linked data web. Philosophical Transactions of the Royal Society (2013)

3. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: Sparql web-querying infrastructure: Ready for action? In: ISWC (2013)
4. Cassidy, S., Ballantine, J.: Version control for rdf triple stores. In: ICSOFT (2007)
5. Erdös, P., Rényi, A.: On the evolution of random graphs. Magyar Tud. Akad. Mat. Kutató Int. Közl 5 (1960)
6. Geerts, F., Karvounarakis, G., Christophides, V., Fundulaki, I.: Algebraic structures for capturing the provenance of sparql queries. In: ICDT (2013)
7. Green, T.J., Ives, Z.G., Tannen, V.: Reconcilable differences. Theory of Computer Systems 49(2) (2011)
8. Gupta, A., Jagadish, H., Mumick, I.S.: Data integration using self-maintainable views. In: EDBT (1996)
9. Halaschek-Wiener, C., Kolovski, V.: Syndication on the web using a description logic approach. J. Web Sem. 6(3) (2008)
10. Ibáñez, L.D., Skaf-Molli, H., Molli, P., Corby, O.: Live linked data: Synchronizing semantic stores with commutative replicated data types. International Journal of Metadata, Semantics and Ontologies 8(2) (2013)
11. Karvounarakis, G., Green, T.J., Ives, Z.G., Tannen, V.: Collaborative data sharing via update exchange and provenance. ACM TODS (August 2013)
12. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. ACM TKDD 1(1) (march 2007)
13. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Fedra: Query Processing for SPARQL Federations with Divergence. Tech. rep., Université de Nantes (May 2014)
14. Morsey, M., Lehmann, J., Auer, S., Stadler, C., Hellmann, S.: Dbpedia and the live extraction of structured data from wikipedia. Program: electronic library and information systems 46(2), 157–181 (2012)
15. Ognyanov, D., Kiryakov, A.: Tracking changes in rdf(s) repositories. In: EKAW (2002)
16. Passant, A., Mendes, P.N.: sparqlpush: Proactive notification of data updates in rdf stores using pubsubhubbub. In: Sixth Workshop on Scripting and Development for the Semantic Web (SFSW) (2010)
17. Saleem, M., Ngomo, A.C.N., Parreira, J.X., Deus, H.F., Hauswirth, M.: Daw: Duplicate-aware federated query processing over the web of data. In: ISWC (2013)
18. Sande, M.V., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., de Walle, R.V.: R&wbase:git for triples. In: LDOW (2013)
19. Schandl, B.: Replication and versioning of partial rdf graphs. In: ESWC (2010)
20. Taylor, N.E., Ives, Z.G.: Reliable storage and querying for collaborative data sharing systems. In: ICDE (2010)
21. Verborgh, R., Sande, M.V., Colpaert, P., Coppens, S., Mannens, E., de Walle, R.V.: Web-scale querying through linked data fragments. In: LDOW (2014)
22. W3C: SPARQL 1.1 Federated Query (march 2013), `http://www.w3.org/TR/sparql11-federated-query/`
23. W3C: SPARQL 1.1 Update (Mar 2013), `http://www.w3.org/TR/sparql11-update/`
24. Wylot, M., Cudre-Mauroux, P., Groth, P.: Tripleprov: Efficient processing of lineage queries in a native rdf store. In: WWW (2014)
25. Zarzour, H., Sellami, M.: srce: a collaborative editing of scalable semantic stores on p2p networks. Int. J. of Computer Applications in Technology 48(1) (2013)
26. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated semantic web data. Web Semant. 11 (Mar 2012)