PeNeLoop

Parallelizing Federated SPARQL Queries in Presence of Replicated Fragments

Minier, Thomas; Montoya, Gabriela; Skaf-Molli, Hala; Molli, Pascal

Published in:
Joint Proceedings of the 2nd RDF Stream Processing (RSP 2017) and the Querying the Web of Data (QuWeDa 2017) Workshops co-located with 14th ESWC 2017 (ESWC 2017), Portoroz, Slovenia, May 28th - to - 29th, 2017

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

PeNeLoop: Parallelizing Federated SPARQL Queries in Presence of Replicated Fragments

Thomas Minier[1], Gabriela Montoya[2], Hala Skaf-Molli[1], and Pascal Molli[1]

[1] LS2N – Nantes University, France
{thomas.minier1@etu.,hala.skaf@,pascal.molli@}univ-nantes.fr
[2] Department of Computer Science – Aalborg University, Denmark
gmontoya@cs.aau.dk

Abstract. Replicating data fragments in Linked Data improves data availability and performances of federated query engines. Existing replication aware federated query engines mainly focus on source selection and query decomposition in order to prune redundant sources and reduce intermediate results thanks to data locality. In this paper, we extend replication-aware federated query engines with a replication-aware parallel join operator: PeNeLoop. PeNeLoop exploits redundant sources to parallelize the join operator and reduce execution time. We implemented PeNeLoop in the federated query engine FedX with the replicated-aware source selection Fedra and we empirically evaluated the performance of FedX + Fedra + PeNeLoop. Experimental results suggest that FedX + Fedra + PeNeLoop outperforms FedX + Fedra in terms of execution time while preserving answer completeness.
Keywords: Linked Data · Parallel Query Processing · Fragment Replication · Federated SPARQL Queries Processing.

1 Introduction

Following the Linked Data principles, billions of RDF triples are made available through SPARQL endpoints. Even if federated SPARQL query engines [8,15,1] allow to execute SPARQL queries over multiple SPARQL endpoints, data-availability and reliability of SPARQL endpoints is still an issue [5].

Data replication is a common practice to overcome availability issues in distributed databases [13]. However, data replication in Linked Data is more challenging: the autonomy of data providers hosting SPARQL endpoints, and data consumers running federated query engines, prevent data replication to be designed. The fragmentation schema and the replication schema remain unknown until a data consumer defines a federation of SPARQL endpoints in a federated query engine.

Existing replication-aware [11,12] and duplicate-aware [14] federated query engines focus on source selection and query decomposition in order to prune redundant sources and use data-locality to reduce intermediate results. We point out that replicated data can also be used to parallelize query processing, and consequently reduce execution time.

In this paper, we extend replication-aware federated query engines with PeNeLoop, a replication-aware parallel join operator. More precisely, PeNeLoop solves the parallel join problem with fragment replication (PJP-FR). Given a SPARQL query and a set of data sources with replicated fragments, the problem is to use all data sources to reduce query execution time while preserving answer completeness and reducing data redundancy.

In contrast to *inter-operator parallelism* proposed in the state-of-the-art federated query engines [1,15], PeNeLoop introduces parallelization at the operator level in order to preserve properties ensured by replicated-aware source selection strategies [11] and replication-aware query decompositions [12].

PeNeLoop is based on Bound Join operator implemented in FedX [15]. Bound joins were originally designed to reduce the number of requests sent in a nested loop join [13]. PeNeLoop extends bound joins processing to use all relevant endpoints with replicated fragments and distribute join processing among them. The contributions of this work are as follows:

(i) We present PeNeLoop, a novel replication-aware parallel join operator that uses replicated fragments to reduce query execution time. PeNeLoop is the fist attempt to use replicated fragments to parallelize query processing in Linked Data
(ii) We extend the federated query engine FedX [15] and the source selection strategy Fedra [11] with PeNeLoop.
(iii) We experiment FedX, FedX+Fedra and FedX+Fedra+PeNeLoop in different setups. We show that FedX + Fedra + PeNeLoop outperforms FedX and FedX + Fedra in terms of execution time while preserving properties of Fedra in terms of reduced number of transferred tuples and answer completeness. The improvements are significative for queries with a large number of intermediate results.

The paper is organized as follows: Section 2 provides background and motivations. Section 3 presents the PeNeLoop approach and algorithm. Section 4 presents our experimental setup and describes our results. Section 5 summarizes related works. Finally, conclusions and future works are outlined in Section 6.

2 Background and Motivations

For replicating data, we follow the approach of replicated fragments introduced in [11,12]. Data consumers replicate fragments composed of RDF triples that satisfy a given triple pattern. Figure 1a shows a fragment from DBpedia which contains RDF triples that match the triple pattern `?film dbo:director ?director`. Fragments are described using a 2-tuple `fd` that indicates the authoritative source of the fragment, e.g. DBpedia, and the triple pattern met by the fragment's triples.

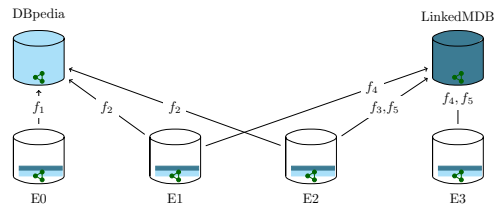
Figure 1b shows a federation with four SPARQL endpoints: E_0, E_1, E_2 and E_3. These endpoints expose replicated fragments from DBpedia and LinkedMDB. Figure 1c describes a federated SPARQL query $Q1$ executed against this

(a) Fragment description

triples(f): { dbr:A_Knight's_Tale
dbo:director dbr:Brian_Helgeland,
dbr:A_Thousand_Clowns
dbo:director dbr:Fred_Coe,
dbr:Alfie_(1966_film)
dbo:director dbr:Lewis_Gilbert,
dbr:A_Moody_Christmas
dbo:director dbr:Trent_O'Donnell,
dbr:A_Movie dbo:director
dbr:Bruce_Conner, $\cdots$ }

fd(f): <dbpedia, ?film dbo:director ?director>

(b) Replicated fragments



fd(f_1): <dbpedia, ?director dbo:nationality ?nat>
fd(f_2): <dbpedia, ?film dbo:director ?director>
fd(f_3): <linkedmdb, ?movie owl:sameAs ?film>
fd(f_4): <linkedmdb, ?movie linkedmdb:genre ?genre>
fd(f_5): <linkedmdb, ?genre linkedmdb:film_genre_name ?name>

(c) Federated SPARQL query $Q1$ and its relevant fragments and endpoints

```
select distinct *
where {
  ?director dbo:nationality ?nat.                (tp1)
  ?film db:director ?director.                   (tp2)
  ?movie owl:sameAs ?film.                       (tp3)
  ?movie linkedmdb:genre ?genre.                 (tp4)
  ?genre linkedmdb:film_genre_name ?gname.       (tp5)
}
```

Triple pattern	Relevant fragment	Relevant endpoint
tp_1	f_1	E_0
tp_2	f_2	E_1, E_2
tp_3	f_3	E_2
tp_4	f_4	E_1, E_3
tp_5	f_5	E_2, E_3

Fig. 1: A federation with replicated fragments

federation and its relevant fragments. For instance, the triple pattern tp_4 has relevant fragment f_4 that has been replicated at E_1 and E_3.

The logical plan of Q1 produced by FedX [15] is presented in Figure 2a. As FedX is not replication-aware, *i.e.*, it does not know that the evaluation of tp_2 at E_1 or E_2 will produce the same results, query execution following this plan will retrieve redundant data from endpoints and increase significantly the query execution time.

The Fedra [11] replication-aware source selection prunes redundant sources in order to minimize intermediate results. Fedra selects E_2 for tp_2, tp_3 and tp_5, E_1 for tp_4 and E_0 for tp_1. Next, Fedra lets FedX builds the logical plan of Figure 2b that minimizes intermediate results.

As pointed in Figure 2b, Fedra has removed E_3 from selected sources of tp_4. However, it also removes an opportunity of parallelization. Indeed, it is possible to use both endpoints to perform in parallel half of the join of $\bowtie_2$ with E_1 and the other half with E_3, as they mirror each other [3].

Such parallelization can be obtained with a replication-aware query decomposer or with intra-operator [13] parallelism. In this paper, we focus on the second approach because it can be easily embedded in current federated query

[3] Note that joins $\bowtie_1$ and $\bowtie_3$ cannot be parallelized in this way, because $\bowtie_1$ is a local join performed at E_2, and tp_1 has only one relevant source.

(a) FedX Left-Linear plan for $Q1$ **(b)** FedX + Fedra Left-Linear plan for $Q1$

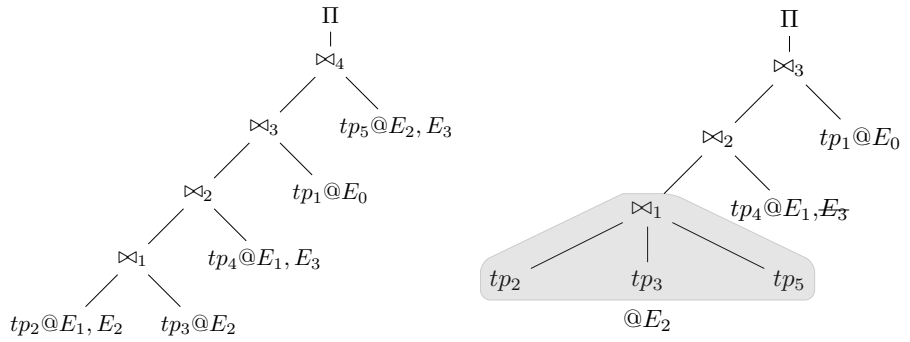


Fig. 2: Logical plans generated by FedX and FedX + Fedra for $Q1$

engines. Consequently, the challenge is to build replication-aware parallel operators to speed-up query execution.

Parallel Join Problem with Fragment Replication (PJP-FR)

Given S_1 and S_2 two disjoint sets of replicated data sources. A set of replicated data sources is a set of endpoints that replicate the same fragments. Given a join $\bowtie_i$ between O_1 and O_2 with relevant sources respectively, S_1 and S_2. The parallel join problem with fragment replication is to distribute the execution of join $\bowtie_i$ among endpoints of S_1 and S_2 in order to minimize the execution time while guaranteeing complete query answers.

3 PeNeLoop : A Replication-Aware Nested Loop Join Operator

PeNeLoop is a solution for parallel join problem with fragment replication with the following assumptions: (i) we focus on nested loop join (NLJ), (ii) we do not consider the load of different endpoints, (iii) we consider that replicated fragments are synchronized, (iv) replicated sources are determined by a replication-aware source selection algorithm as Fedra before pruning.

3.1 NLJ Processing

During a NLJ processing, the query engine iteratively evaluates each triple pattern, starting with a single pattern and substituting the set of mappings produced by the pattern's execution in the next evaluation step. Even if a NLJ is more efficient when the first evaluated triple patten is more selective than the others, it still produces many remote requests in a distributed setting. In FedX [15], the Bound Join (BJ) operator is proposed to minimize the number of join steps and the number of requests sent in nested loop joins. A BJ consists of a nested

loop join where sets of mappings are grouped in *blocks*, *i.e.*, as a single subquery using SPARQL UNION constructs. The subquery is then sent to the relevant endpoint in a single remote request. This technique acts as a distributed semijoin and allows to reduce the number of requests by a factor equivalent to the size of the *block*.

PeNeLoop proposes to parallelize the BJ operator itself. Instead of sending all blocks to the same endpoint, PeNeLoop uses the knowledge about replicated sources to further parallelize the bound join operator. When processing a join in a basic graph pattern (BGP), if the current triple pattern has N relevant sources that replicate the same fragment, PeNeLoop sends each block to a different endpoint in a Round Robin fashion, *i.e.*, the block b_i is sent to the endpoint E_k, $k = i \mod N$. Therefore, PeNeLoop does not increase the number of remote calls while increasing the parallelization during join processing.

3.2 PeNeLoop Algorithm

Algorithm 1: PeNeLoop

Input: $tp = <s, p, o>$: a triple pattern, $E = \{E_0, \ldots, E_{m-1}\}$: relevant endpoints of tp, $NextOp$: next operator in the pipeline, b: maximum number of mappings per block

Data: M_i: a set of mappings produced by the previous operator in the pipeline, $B = \{M_1, \ldots, M_n\}$: block of sets of mappings waiting to be sent

Init: $B = \{\}$, $k = 0$

```
1  SendBlock(block, tp):
2  |  Q = GroupedSubquery(block, tp)
3  |  SendQuery(Q) to E_k
4  |  B = {}
5  |  k = (k + 1) mod Size(E)
6  ↯ onMappings(M_i):
7  |  B = B ∪ {M_i}
8  |  if Size(B) ⩾ b then
9  |  |  SendBlock(B, tp)
10 |  end
11 ↯ onResults(R):
12 |  Send(R) to NextOp
13 ↯ onEnd():
14 |  if Size(B) ⩾ 0 then
15 |  |  SendBlock(B, tp)
16 |  end
17 |  Close()
```

PeNeLoop is defined as part of a pipelining approach allowing for intermediate results to be processed by the next operator as soon as they are ready, providing higher throughput than a blocking model.

Algorithm 1 describes the PeNeLoop algorithm using an event driven paradigm. Sets of mappings M_i are produced by the previous operator in the pipeline and sent in continuous to PeNeLoop operator. When a set M_i arrives (Line 6), it is stored in the next block B. When B reaches its maximum size b (Line 8), PeNeLoop generates a subquery in a Bound Join fashion using B and tp

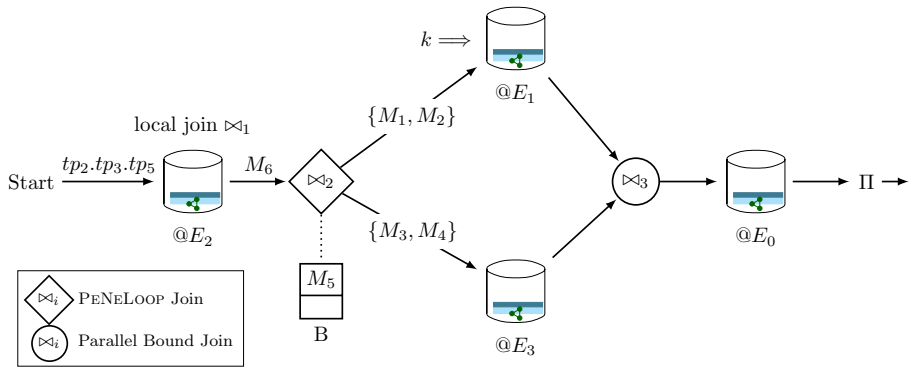


Fig. 3: Join processing of $Q1$ with PeNeLoop

(Line 2). Then, the subquery is sent to the endpoint E_k (Line 3), B is cleared and the next endpoint is selected using our Round Robin approach (Line 5).

When results, *i.e.*, new sets of mappings, arrive from the requested endpoints (Line 11), they are sent to the next operator in the pipeline. Finally, when the previous operator has completed its work and will not produce any more data (Line 13), PeNeLoop sends the last non-empty block and then close the operator.

In the following, we illustrate PeNeLoop processing for the query $Q1$ (Figure 1c) using the query plan generated by FedX + Fedra (Figure 2b). For simplicity, we fix $b = 2$.

Figure 3 illustrates a snapshot of the pipeline during the evaluation of the triple pattern tp_4 of the query $Q1$. We focus on processing of join $\bowtie_2$, performed using PeNeLoop. Two blocks $\{M_1, M_2\}$ and $\{M_3, M_4\}$ have been already sent to E_1 and E_3, respectively. A set of mappings M_5 arrived from the join $\bowtie_1$ and was placed in the next block. When another set of mappings M_6 arrives, the block will be full and sent to the next endpoint E_1. Join $\bowtie_2$ ends when no more mappings are produced by join $\bowtie_1$.

4 Experimental Study

The goal of the experimental study is to evaluate the execution time reduction obtained with the parallelization enabled by PeNeLoop. Moreover, such reduction is obtained without degrading the reduced number of transferred tuples and the answer completeness granted by Fedra. We compare the performance of the federated query engine FedX alone, FedX with the addition of Fedra (FedX + Fedra) and FedX with both Fedra and PeNeLoop (FedX + Fedra + PeNeLoop).

We expect to see that FedX + Fedra + PeNeLoop exhibits lower query execution time than FedX and FedX + Fedra, while maintaining the same number of transferred tuples and answer completeness.

Dataset and Queries: We use one instance of the Waterloo SPARQL Diversity Test Suite (WatDiv) synthetic dataset [2,3] with 10^5 triples. We generate 50,000 queries from 500 templates. Next, we unbound subjects and objects of each query. 100 queries with at least one join are then randomly picked to be executed against our federations. Generated queries are STAR, PATH and SNOWFLAKE shaped queries, we use the DISTINCT modifier.

Federations: We setup three federations with respectively 10, 20 and 30 SPARQL endpoints, and generate three versions of each of these federations by randomizing the fragmentation schema. Every schema is distinct from the others. Fragments are created from the 100 random queries and are replicated exactly three times to provide opportunities of parallelization.

To measure the number of transferred tuples, the federated query engine accesses SPARQL endpoints through a proxy. All the federation endpoints are deployed on the same machine, and to simulate the network latency, the proxies were configured to add a delay of 30ms to each request.

Hardware configuration: One machine with Intel Xeon E5-2680 v2 2.80GHz and 128GB of RAM hosts the SPARQL endpoints and performs the queries. Each SPARQL endpoint is deployed using Jena Fuseki 1.1.1[4]. Fuseki is configured to handle incoming queries on only one executing thread to increase the stress load and study the effect of the parallelization done by the engine. Endpoints have no limitations in term of memory used.

Implementations: FedX + Fedra implementation[5] (in Java) has been modified to preserve the multiple sources that provide the same relevant fragments. Additionally, FedX join processing has been modified to remove some redundant synchronization barriers imposed by FedX on the first join of a plan, *i.e.*, the right operand can start execution before the left one has finished its evaluation, and to use PeNeLoop operator when possible[6]. Every configuration of this experimental study has received the same modifications. Proxies used to measure results are implemented in Java 1.7, using the Apache HttpComponents Client library 4.3.5[7].

Evaluation Metrics: *i*) *Execution Time (ET):* is the elapsed time since the query is posed until the complete answer is produced. We used a timeout of 1800 seconds. *ii*) *Number of parallelized queries (NPQ):* is the number of queries where at least one join has been parallelized by PeNeLoop. This metric is only used in FedX + Fedra + PeNeLoop. Queries marked as *improved* have a lower execution time (*ET*) with FedX + Fedra + PeNeLoop than with FedX + Fedra. *iii*) *Number of Transferred Tuples (NTT):* is the number of transferred tuples from all the endpoints to the query engine during a query evaluation. *iv*) *Completeness (C):* is the ratio between the answers produced by the query execution engine and the answers produced by the evaluation of the

[4] `http://jena.apache.org/`, January 2015.
[5] `https://github.com/gmontoya/fedra`, June 2016.
[6] Implementation available at: `https://github.com/Callidon/peneloop-fedx`
[7] `https://hc.apache.org/`, October 2014.

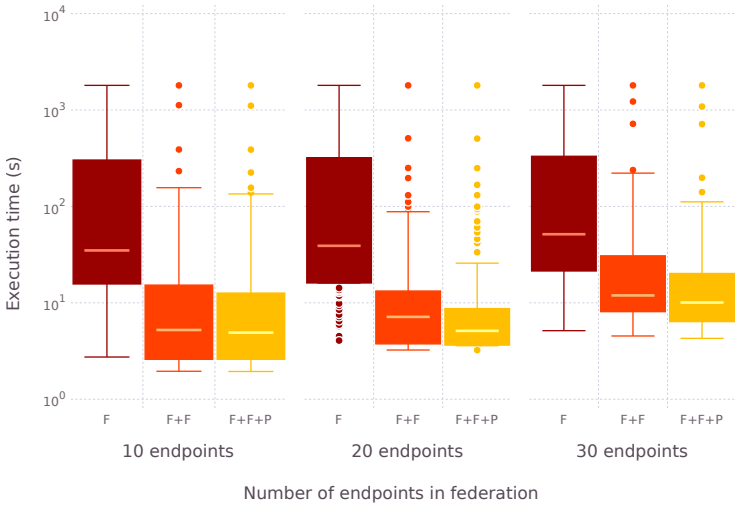


Fig. 4: Average execution time with FedX (F), FedX + Fedra (F+F) and FedX + Fedra + PeNeLoop (F+F+P).

query over the set of all triples available in the federation; values range between 0.0 and 1.0.

Results presented for *ET*, *NTT* and *C* correspond to the average over the three versions generated for each size of federation. Queries that failed to deliver an answer due to a query engine internal error are excluded from the final results.

Statistical Analysis: The Wilcoxon signed rank test [17] for paired non-uniform data is used to study the significance of the improvements on performance obtained when the join execution benefits from replicated fragments.[8]

4.1 Execution time

Figure 4 summarizes the execution time (*ET*) for the three federations. Execution time (*ET*) with FedX + Fedra + PeNeLoop is better for all federations than with FedX and FedX + Fedra. As queries have unbounded subjects and unbounded objects, they generated more intermediate results during joins, which allow PeNeLoop to distribute more bindings between relevant sources. Figure 5 presents the execution time for queries with a large number of intermediate results (at least 1000 tuples). This represents 562 queries out of 865 for all federations. PeNeLoop is even more efficient for queries with a large number of intermediate results. This is an important result because generally the number of the intermediate results impacts negatively the query execution time.

[8] The Wilcoxon signed rank test was computed using the R project (`http://www.r-project.org/`)

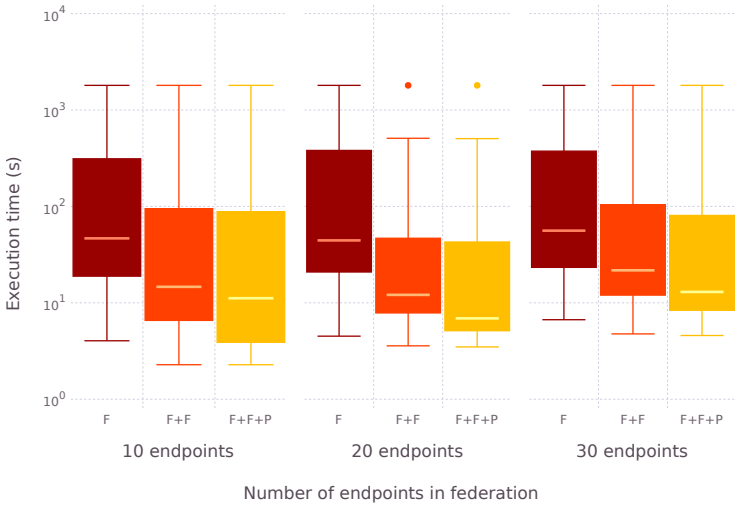


Fig. 5: Average execution time with FedX (F), FedX + Fedra (F+F) and FedX + Fedra + PeNeLoop (F+F+P) for queries with at least 1000 intermediate results.

Both FedX + Fedra and FedX + Fedra + PeNeLoop benefit from the reduction of transferred tuples granted by Fedra, which reduce the number of mappings that PeNeLoop can distribute.

To confirm that PeNeLoop reduces the execution time of FedX + Fedra, a Wilcoxon signed rank test was run for results of Figure 4 with the hypotheses:
H0: PeNeLoop does not change the engine query execution time.
H1: PeNeLoop reduces FedX + Fedra's query execution time.

We obtain p-values no greater than 1.639×10^{-4} for each federation. These low p-values allow for rejecting the null hypothesis that the execution time of FedX + Fedra and FedX + Fedra + PeNeLoop are the same. Additionally, it supports the acceptance of the alternative hypothesis that FedX + Fedra + PeNeLoop has a lower execution time.

4.2 Number of Parallelized Queries

Figure 6 presents the number of parallelized queries (*NPQ*) in FedX + Fedra + PeNeLoop for the three versions of each federation. PeNeLoop increases query parallelization during join processing, especially in larger federations where fragments are more scattered across endpoints. In most cases, queries parallelized by PeNeLoop are improved, *i.e.*, they exhibit a lower execution time compared to FedX + Fedra. Parallelized queries with unimproved execution time are those that do not have a large number of intermediate results. Parallelization of such

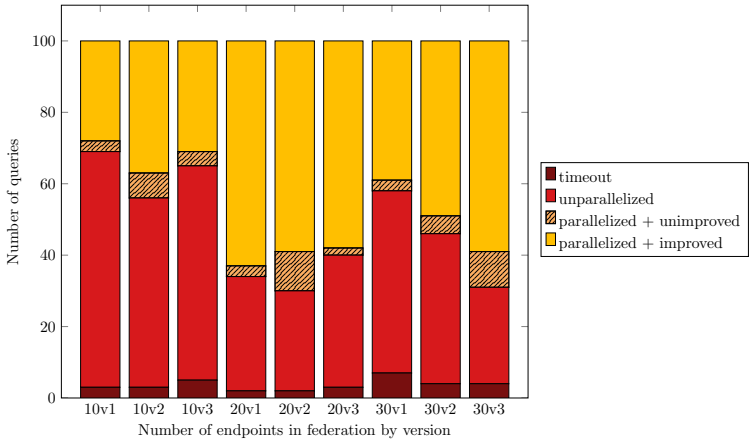


Fig. 6: Number of parallelized queries with FedX + Fedra + PeNeLoop.

queries does not improve query performance, as their joins were not originally costly to evaluate.

As pointed in Figure 6, the number of parallelized queries is not constant within different versions the same federation, because the replication schema directly influences query parallelization. When this schema is not designed, as in Linked Open Data, PeNeLoop creates parallelization where locality cannot be used by Fedra to optimize the query execution plan.

4.3 Number of transferred tuples

Figure 7 summarizes the number of transferred tuples (NTT) in different federations. FedX + Fedra + PeNeLoop transfers the same amount of tuples as FedX + Fedra. This demonstrates that PeNeLoop does not deteriorate the reduction of transferred tuples provided by Fedra. Moreover, modifications performed on FedX to remove some synchronisation barriers do not introduce any difference between FedX + Fedra and FedX + Fedra + PeNeLoop in terms of number of transferred tuples and do not impact FedX + Fedra performance.

4.4 Completeness

Figure 8 presents results concerning answer completeness (C) for the different federations. In all cases, FedX + Fedra + PeNeLoop is able to produce the same answers as FedX + Fedra for all queries.

As observed with the number of transferred tuples (NTT), our modification for FedX does not reduce the completeness of FedX and FedX + Fedra, which support our claim that this modification does not impact negatively FedX + Fedra.

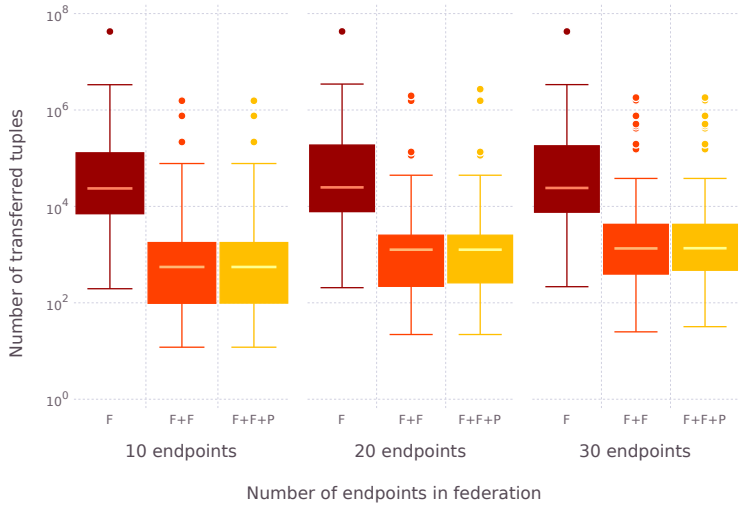


Fig. 7: Average number of transferred tuples with FedX (F), FedX + Fedra (F+F) and FedX + Fedra + PeNeLoop (F+F+P).

4.5 Synthesis

Experimental study results confirm that PeNeLoop can further increase the performance of join processing in presence of replicated fragments. Execution time in average is lower with FedX + Fedra + PeNeLoop than with FedX or FedX + Fedra, and the reduced number of transferred tuples granted by Fedra is maintained. Answer completeness is not degraded. PeNeLoop is able to parallelize a significant number of queries in presence of replicated fragments and shows to be more efficient on larger federations. Query performance are significantly improved for queries with a large number of intermediate results, and the time to evaluate joins is reduced by taking advantage of parallel processing.

5 Related Work

Fedra [11] is a replication-aware source selection that uses data locality produced by replicated fragments to enhance federated query engines performances. Fedra uses Union and BGP reductions to prune data sources and finds as many sub-queries that can be executed against the same endpoint as possible, leading to evaluation of local joins and a reduced number of transferred tuples. PeNeLoop uses replicated fragments differently. As seen in Section 2, Fedra prunes redundant endpoints that cannot be used to creates localities, whereas PeNeLoop uses these endpoints to create more opportunities of parallelization.

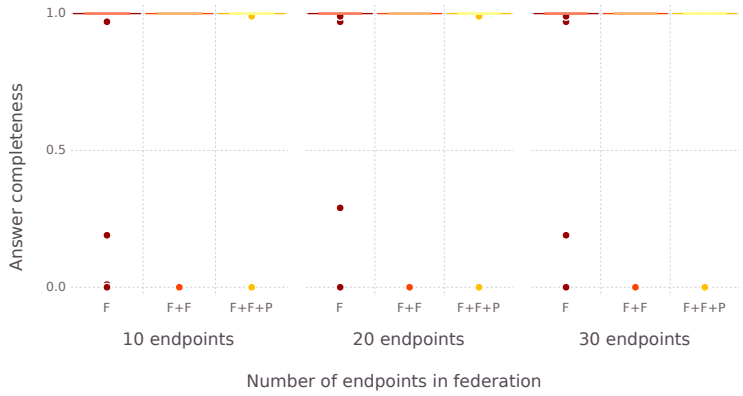


Fig. 8: Average completeness with FedX (F), FedX + Fedra (F+F) and FedX + Fedra + PeNeLoop (F+F+P).

LILAC [12] is a replication-aware decomposer. Compared to Fedra, LILAC is able to reduce intermediate results by allocating a triple pattern to several endpoints. As for Fedra, PeNeLoop can reuse source selection performed by LILAC to introduce intra-operator parallelism.

Other existing sources selection techniques reduce the number of selected sources by a federated SPARQL query engine. BBQ [9] and DAW [14] use sketches to estimate the overlapping among sources, but they only operate on duplicated sources and not on replication itself. They do not provide information about replicated fragments that allow PeNeLoop to efficiently parallelize join processing.

Parallel join processing in distributed database systems has been the subject of significant investigation. Parallel nested loop algorithms have been investigated in [4,6], but they do not use replication for parallelization. Instead, replication is mostly used for fault tolerance and to locate data closer to their access points [10,13], improving query performance by reducing communication time. PeNeLoop does not use localities created by data redundancy, but opportunities of parallelization created by this redundancy.

Parallel join processing has been also studied in federated query engines. For instance, [1,15,7] propose parallel architectures for executing queries concurrently at different data sources. Anapsid [1] takes advantage of bushy query execution plans to create *inter-operator parallelism*. FedX [15] implements *bound joins* in a distributed and highly parallelized environment where different subqueries can be executed at the endpoints concurrently. PeNeLoop creates intra-operator parallelism and proposes a more advanced parallel join processing using replication. Similar to FedX, subqueries are executed concurrently, but they are distributed between endpoints, increasing parallelization.

To our knowledge, none of existing federated query engines propose to take advantage of replicated data for join processing or propose a replication-aware parallel join operator.

6 Conclusions and Future Works

In this paper, we extended a replication-aware federated query engine with a new replication-aware parallel join operator PeNeLoop. PeNeLoop provides intra-operator parallelism relying on replicated data. In this way, PeNeLoop preserves properties of source-selection and query decomposition replication-aware federated query engines. We implemented PeNeLoop in FedX. Evaluation results demonstrates that PeNeLoop improves significantly query performance.

PeNeLoop is the first attempt to use replicated data to parallelize query processing in Linked Open Data and opens several perspectives. First, we made the assumption that the load of the endpoints is uniform during query execution. We can leverage this hypothesis by making PeNeLoop adaptive to the performances of endpoints. Second, we focused on a Nested Loop Join operator, we can also parallelize others operators such as Symmetric Hash-Join [18] used in Anapsid. Finally, we focused on SPARQL endpoints, and we think that parallel query processing in presence of replicated fragments can also be applied to the Triple Pattern Fragment approach [16].

Acknowledgments. This work is partially supported through the FaBuLA project, part of the AtlanSTIC 2020 program.

References

1. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: Anapsid: an adaptive query processing engine for sparql endpoints. In: International Semantic Web Conference. pp. 18–34. Springer (2011)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of rdf data management systems. In: International Semantic Web Conference. pp. 197–212. Springer (2014)
3. Aluç, G., Ozsu, M., Daudjee, K., Hartig, O.: chameleon-db: a workload-aware robust rdf data management system. university of waterloo. Tech. rep., Tech. Rep. CS-2013-10 (2013)
4. Bitton, D., Boral, H., DeWitt, D.J., Wilkinson, W.K.: Parallel algorithms for the execution of relational database operations. ACM Transactions on Database Systems (TODS) 8(3), 324–353 (1983)
5. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P.Y.: Sparql web-querying infrastructure: Ready for action? In: International Semantic Web Conference. pp. 277–293. Springer (2013)
6. DeWitt, D.J., Naughton, J.F., Burger, J.: Nested loops revisited. In: Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on. pp. 230–242. IEEE (1993)

7. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: Proceedings of the Second International Conference on Consuming Linked Data - Volume 782. pp. 13–24. COLD'11, CEUR-WS.org, Aachen, Germany, Germany (2010), http://dl.acm.org/citation.cfm?id=2887352.2887354
8. Görlitz, O., Staab, S.: Federated Data Management and Query Optimization for Linked Open Data, vol. 331, pp. 109–137. Springer, Heidelberg (2011)
9. Hose, K., Schenkel, R.: Towards benefit-based rdf source selection for sparql queries. In: Proceedings of the 4th International Workshop on Semantic Web Information Management. p. 2. ACM (2012)
10. Kossmann, D.: The state of the art in distributed query processing. ACM Computing Surveys (CSUR) 32(4), 422–469 (2000)
11. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Federated sparql queries processing with replicated fragments. In: International Semantic Web Conference. pp. 36–51. Springer International Publishing (2015)
12. Montoya, G., Skaf-Molli, H., Molli, P., Vidal, M.E.: Decomposing federated queries in presence of replicated fragments. Web Semantics: Science, Services and Agents on the World Wide Web 42, 1 – 18 (2017), //www.sciencedirect.com/science/article/pii/S1570826816300580
13. Özsu, M.T., Valduriez, P.: Principles of distributed database systems. Springer Science & Business Media (2011)
14. 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: International Semantic Web Conference. pp. 574–590. Springer (2013)
15. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. In: International Semantic Web Conference. pp. 601–616. Springer (2011)
16. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: A low-cost knowledge graph interface for the web. Web Semantics: Science, Services and Agents on the World Wide Web 37, 184–206 (2016)
17. Wilcoxon, F.: Individual comparisons by ranking methods. In: Breakthroughs in Statistics, pp. 196–202. Springer (1992)
18. Wilschut, A.N., Apers, P.M.: Dataflow query execution in a parallel main-memory environment. Distributed and Parallel Databases 1(1), 103–128 (1993)