Query Optimizations over Decentralized RDF Graphs

Ibrahim Abdelaziz *, Essam Mansour ◇, Mourad Ouzzani ◇, Ashraf Aboulnaga ◇, Panos Kalnis *

* *King Abdullah University of Science & Technology, Saudi Arabia* ◇ *Qatar Computing Research Institute, HBKU, Qatar*
{first}.{last}@kaust.edu.sa, {emansour,mouzzani,aaboulnaga}@qf.org.qa

***Abstract*—Applications in life sciences, decentralized social networks, Internet of Things, and statistical linked dataspaces integrate data from multiple decentralized RDF graphs via SPARQL queries. Several approaches have been proposed to optimize query processing over a small number of heterogeneous data sources by utilizing schema information. In the case of schema similarity and interlinks among sources, these approaches cause unnecessary data retrieval and communications, leading to poor scalability and response time.**

This paper addresses these limitations and presents *Lusail*, a system for scalable and efficient SPARQL query processing over decentralized graphs. Lusail achieves scalability and low query response time through optimizations at compile and run times. At compile time, we use a novel locality-aware query decomposition technique that maximizes the number of query triple patterns sent together to a source based on the actual location of the instances satisfying these triple patterns. At run time, we use selectivity-awareness and parallel query execution to reduce network latency and to increase parallelism by delaying the execution of subqueries expected to return large results. Lusail is evaluated using real and synthetic benchmarks, with data sizes up to billions of triples on an in-house cluster and a public cloud. Our experiments show that Lusail outperforms state-of-the-art systems by orders of magnitude in terms of scalability and response time.

I. Introduction

The Resource Description Framework (RDF) is extensively used to represent Web data. It uses a simple data model in the form of triples ⟨subject, predicate, object⟩. A key feature is the ability to link two entities from two different RDF datasets which are maintained by two local independent authorities, as shown in Figure 1. Through such links, *large decentralized graphs* can be easily created among a large number of RDF stores where each RDF store provides its own SPARQL endpoint.

Today, decentralized RDF graphs consist of more than 85 billions triples over more than 3400 datasets[1] in different domains, such as media, government, and life sciences [1]. In life sciences, Bio2RDF[2] has decentralized graphs of about 11 billions triples across 35 datasets. 270a[3] is a statistical data analysis project with 3.5 billion triples in 11 datasets. Internet of Things will connect billions of decentralized datasets [2]. Users of decentralized social networks [3], [4] store their data in their own RDF dataset and each user's dataset is connected with remote datasets of other users. Such networks have the potential to create large decentralized graphs across clusters of hundreds of independent datasets.

[1]http://stats.lod2.eu/
[2]http://bio2rdf.org/
[3]http://270a.info/

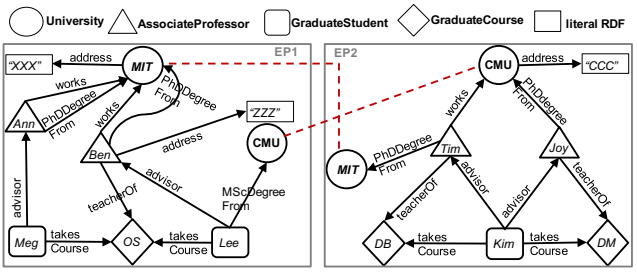


Fig. 1. Decentralized graphs for two universities managed by independent authorities providing SPARQL endpoints (EP). The red dotted line represents an interlink between endpoints, i.e., a vertex in an endpoint refers to another vertex in another endpoint. Thus, to get the address of the university from which Tim got his PhD, the interlink from EP2 to EP1 has to be traversed.

```
SELECT ?S ?P ?U ?A WHERE{
 ?S ub:advisor ?P .        ?S rdf:type ub:graduateStudent .
 ?P ub:teacherOf ?C .      ?P rdf:type ub:associateProfessor .
 ?S ub:takesCourse ?C .    ?C rdf:type ub:graduateCourse .
 ?P ub:PhDDegreeFrom ?U .  ?U ub:address ?A . }
```

Fig. 2. Q_a: a SPARQL query over decentralized RDF graphs across different universities. This query has to traverse the interlink between EP2 and EP1.

The emergence of these large decentralized graphs offers unique opportunities to integrate data from multiple RDF datasets via SPARQL queries. For example, Figure 2 shows a query (Q_a) on data from the LUBM benchmark [5] at two endpoints. Q_a returns all students who are taking courses with their advisors along with the URI and location of the advisors' alma mater. Q_a has three possible answers: (Kim, Joy, CMU, "CCCC"), (Kim, Tim, MIT, "XXX") and (Lee, Ben, MIT, "XXX"). Evaluating Q_a independently at each endpoint and concatenating their results will miss the results about Tim as EP2 does not have the address of MIT. We need to traverse the interlinks between the endpoints to return the full answer. Thus, Q_a has to be decomposed into subqueries to be submitted to the relevant endpoints and whose partial results has to be joined. Middleware to efficiently perform federated queries over multiple independent SPARQL endpoints is required.

Similar to federated query processing systems, processing a query over decentralized RDF graphs tries to push as much processing to the local SPARQL endpoint as possible. Schema information can be collected using SPARQL *ASK* queries, which check whether or not a triple pattern has a solution at an endpoint. Based on schema information, we may find that a specific RDF predicate has an answer at only one endpoint, e.g., the predicate *MScDegreeFrom* has an answer only at EP1. If a group of triple patterns can be answered by only one endpoint, it can be processed by this endpoint as one unit, in the form of an *exclusive groups* [6]. However, with RDF sources often utilizing similar ontologies and having interlinks, a given triple pattern may need to be answered by multiple endpoints, such as the triples in query Q_a of Figure 2. In this case, the triple pattern cannot be part of an exclusive group

and needs to be sent to all the endpoints as an individual triple pattern. This would lead to many queries being processed one triple pattern at a time. These triple patterns retrieve a large amount of data, which is then bound to other variables and sent to other endpoints to compute the query results.

In this paper, we explore the possibility of avoiding the processing of queries one triple pattern at a time by utilizing knowledge of the locations of the actual RDF triple instances matching a query variable. Knowing the actual location of these instances leads to two different cases of processing triple patterns: (*i*) local instances, e.g., the instances matching the variable `?S` in ⟨`?S`, ub:advisor, `?P`⟩ and ⟨`?S`, ub:takesCourse, `?C`⟩ are located in the same endpoint, and (*ii*) remote instances, e.g., one of the instances matching the variable `?U` in EP2, whose triples are ⟨`?P`, ub:PhDDegreeFrom, `?U`⟩ and ⟨`?U`, ub:address, `?A`⟩, is located in different endpoints. In the former case, the triple patterns can be processed as one unit by the same endpoint while in the latter, they have to be sent separately and then joined by a federated query processor. In a sense, exclusive groups [6] find groupings of triple patterns based on schema information, and we are proposing to find groupings based on instance information.

Existing federated SPARQL systems, such as SPLENDID [7], HiBISCuS [8], and FedX [6], cannot determine the location of triples in a general and accurate way. Hence, they retrieve unnecessary data from the data sources, leading to poor scalability and response time. We better highlight this problem through a simple experiment described in Section II. In addition, these systems retrieve triples from endpoints, bind them to other triple patterns, and then join the intermediate data with other endpoints using a *bound join* operation. This process limits the available parallelism since only one join step is processed at a time, and the federated query processor has to wait for the results of this join step before issuing the next join. There is a need to move less unnecessary data while retrieving this data in parallel from endpoints.

This paper addresses the above limitations and introduces Lusail (Section III), a system for scalable and efficient SPARQL query processing over decentralized RDF graphs. Lusail supports queries on an arbitrary set of endpoints without requiring prior knowledge of the data or the endpoints. Queries are processed in a two-tier strategy: (*i*) Locality-Aware DEcomposition (*LADE*) of the query into subqueries to maximize the computations at the endpoints and minimize intermediary results, and (*ii*) Selectivity-Aware and Parallel Execution (*SAPE*) to reduce network latency and increase parallelism. When two or more triple patterns have solutions at different endpoints, LADE investigates at each endpoint which of these triple patterns access remote data based on the actual location of the instances satisfying the triple patterns. Based on this check, LADE decomposes the query into a set of subqueries that will be executed independently at one or more endpoints. Afterward, SAPE dynamically decides to delay subqueries expected to return large results. It also chooses the join order among the subquery results based on their actual size and the highest degree of parallelism that could be achieved. SAPE uses a cost model for balancing between remote requests and local computations.

In summary, our contributions are as follows:

- A locality-aware decomposition method that dramatically reduces the number of remote requests and allows for better utilization of the endpoints. (Section IV)
- A cost model that uses lightweight runtime statistics. The cost model is used to decide the order of submitting subqueries and the tree execution plan for joining the results of these subqueries in a non-blocking fashion. This leads to a parallel execution that balances between remote requests and local computations. (Section V)
- Comprehensive experiments on real and synthetic benchmarks using two settings: a cluster of 20 machines and a federation of 18 virtual machines on a public cloud in 7 different geographic regions. We show that Lusail outperforms state-of-the-art systems by up to three orders of magnitude and scales-up to 256 endpoints, whereas other systems cannot scale beyond 4 endpoints. (Section VI).

We discuss related work in Section VII and conclude the paper in Section VIII.

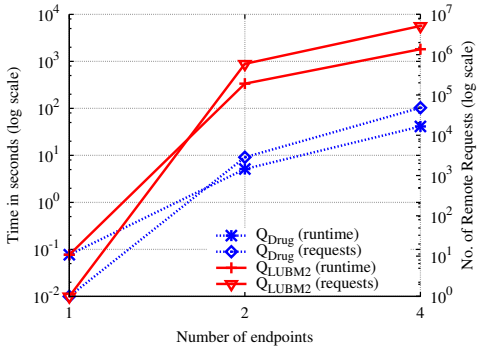


Fig. 3. Sensitivity of FedX to the number of endpoints with cached results from source selection. The trend of both runtime and no. of request is similar.

II. Limitations to Scalability

To highlight the need for Lusail, we examine the relationship between the size of intermediate results, the number of remote requests, and query response time. To this end, we conduct an experiment using FedX [6], which was shown in a recent survey [9] to outperform all existing systems. FedX starts processing queries by sending ASK requests to determine relevant sources and then determines join order optimization as well as exclusive groups. It uses block nested loops join to reduce the number of HTTP requests in the bound join. We use the QFed [10] and LUBM [5] benchmarks with two queries, the *Drug* query from QFed and *Q2* from LUBM (see the queries in [11])[4]. The *Drug* query finds all medicines that target asthma and optionally obtains information about them. It uses up to 4 datasets (in 4 endpoints). *Q2* finds graduate students who are registered in courses delivered by their advisor. Each endpoint in the LUBM benchmark corresponds to a university.

Figure II shows the response time and the number of remote requests, excluding requests made during source selection, for different numbers of endpoints. We run each query multiple times and enable FedX to cache the full results of source selection. We see a clear correlation between query response time and the number of HTTP requests, thus limiting scalability. Processing one triple pattern at a time while binding query variables to values from intermediate results cause a huge number of remote requests. Even with the optimizations

[4]The benchmark queries used in this paper are listed in [11].

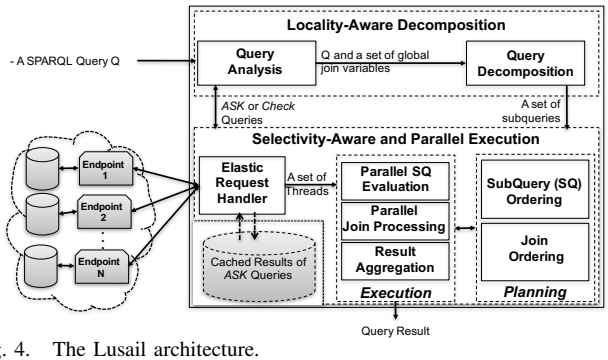


Fig. 4. The Lusail architecture.

employed by FedX, remote requests are still a bottleneck. In the bound join used by FedX, the number of remote requests depends on the size of the intermediate data. The bound join also reduces parallelism since only one join step is processed at a time. We thus need a way to avoid performing the bound join one triple pattern at a time.

One simple alternative is to evaluate each triple pattern independently against all relevant endpoints, without any binding. The data retrieved by these triple patterns can then be joined by the federated query processor. While significantly reducing the number of remote requests, this strategy substantially increases the amount of data retrieved from the endpoints, which makes it an inefficient strategy overall. Lusail avoids the bound join by using a locality-aware decomposition technique. This technique maximizes the number of triple patterns sent together as a subquery to an endpoint based on the actual location of the data instances satisfying these triple patterns. Maximizing the number of triple patterns per subquery reduces the probability of shipping irrelevant data.

III. LUSAIL ARCHITECTURE

The Lusail architecture, shown in Figure 4, includes two main components: the locality-aware decomposition (LADE) component, and the selectivity-aware planning and parallel execution (SAPE) component. Query decomposition starts by analyzing the query to identify the relevant endpoints (source selection). Like similar systems, we use a set of SPARQL *ASK* queries, one for each triple pattern. In contrast to other systems, LADE takes the additional step of checking, for each pair of triple patterns with a common (or join) variable, whether the pair can be evaluated as one unit by the relevant endpoints. The result of this check determines a group of triple patterns, i.e., a subquery, that can be sent together to an endpoint. To take advantage of previously submitted *ASK* queries, Lusail caches their results in a hash table.

SAPE takes as input the set of subqueries produced by LADE and schedules them for execution. Unlike other systems, each subquery is treated as an independent task, thus allowing Lusail to utilize more threads as needed. Such decision is made by the Elastic Request Handler. The ideal case is to have one thread per endpoint for handling requests and processing intermediate results. The Elastic Request Handler also allocates these threads to physical cores. Based on the number of relevant endpoints and cardinality estimates for the different triple patterns, our cost model delays subqueries that are expected to return large results. Lusail assigns a thread per endpoint to collect the result of the subquery. The result is seen as a relation, whose data is partitioned among different threads. Lusail decides the join order based on the number of partitions and the actual size of the result per subquery. Therefore, we can achieve a high degree of parallelism while minimizing the communication cost at two levels: (*i*) globally, by getting results from different endpoints simultaneously, and (*ii*) internally, by utilizing different threads in joining results.

IV. LOCALITY-AWARE DECOMPOSITION

Over decentralized RDF graphs, we may have a pair of triples where the data instances matching them are not located in the same endpoint, e.g., the triples having `?U` as a common variable in Figure 1. Thus, allowing these two triples to be in the same subquery will lead to incorrect result. LADE starts by analyzing which triples cannot be in the same subquery, and identifying the common variables in these triples as global join variables (GJV). Then, it decomposes the conjunction of triple patterns into subqueries. LADE aims to maximize the number of triple patterns sent together to each endpoint. We assume no prior knowledge of the data sources, such as schema, data distribution, or statistics. LADE relies solely on a set of check queries written in SPARQL.

A. Global Join Variables and Check Queries

A global join variable (v) is a variable that appears in at least two different triple patterns, where these triple patterns taken together cannot be solved by a single endpoint. A join between data located at two or more endpoints will be needed. Given two triple patterns, TP_i and TP_j, in a subquery, a GJV may appear in the triple patterns as: (*i*) *object* in TP_i and *subject* in TP_j, (*ii*) *object* in both patterns, or (*iii*) *subject* in both. Let v_i and v_j be the sets of instances of v that satisfy TP_i and TP_j, respectively.

Q_a in Figure 2 has four variables appearing in more than one triple pattern, namely `?S`, `?U`, `?P`, and `?C`. Figure 5 shows our analysis for the first three variables. In EP1 and EP2, all instances matching `?S` in ⟨`?S`, ub:advisor, `?P`⟩ are co-located with all instances matching `?S` in ⟨`?S`, ub:takesCourse, `?C`⟩. Thus, `?S` is not a GJV and hence the two corresponding triple patterns can be sent together in a single subquery (one single HTTP request) to each relevant endpoint. However, for the triples involving `?U`, ⟨`?P`, ub:PhDDegreeFrom, `?U`⟩ and ⟨`?U`, ub:address, `?A`⟩, we notice that in EP2 there is a professor, Tim, who got his PhD from another university. Thus, to get the address of that university, we need to perform a join between data fetched from EP1 and EP2. Therefore, `?U` is a GJV.

We now describe how LADE finds GJVs by determining the actual location of data instances depending on the roles they play, i.e., object or subject. We first discuss how to merge two triple patterns and then generalize to more than two. Two triple patterns TP_i and TP_j are put together in a single subquery under two conditions: (*i*) both triple patterns have the same list of relevant endpoints, and (*ii*) each relevant endpoint can fully answer both triple patterns without missing any result, i.e, all instances that match v in TP_i and TP_j are in the same endpoint.

Object and Subject. Consider the variable `?U` in Q_a (Figure 2). It appears as an object in TP_i: `?P ub:PhDDegreeFrom ?U` and as a subject in TP_j: `?U ub:address ?A`. Checking the location of the data instances v_i and v_j that match `?U` in each endpoint has two

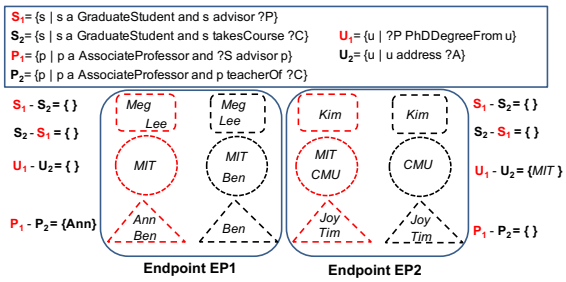


Fig. 5. Locality analysis of data instances in EP1 and EP2 that match ?S, ?U, and ?P in a pair of triples in Q_a. This figure is based on Figure 1.

```
1 SELECT ?P WHERE {
2 ?P rdf:type T
3 ?S < Predicate_i > ?P .
4 FILTER NOT EXISTS { SELECT ?P WHERE {
5    ?P < Predicate_j > ?C .
6 }} . } LIMIT 1
```

Fig. 6. A Lusail SPARQL check query to detect whether ?P is a global join variable or not. The check query returns zero or only one value.

cases: (*i*) remote instances, where v_i and v_j are located in different endpoints, i.e., all or some professors received their PhD from another university (in a different endpoint); e.g., EP2 in Figure 5, and (*ii*) local instances, where all v_i and v_j are located in the same endpoint, i.e., all professors teaching in a university A received their PhD from A (in the same endpoint); e.g., EP1 in Figure 5.

We check the relative complement (i.e., set difference) of v_i and v_j in all relevant endpoints by sending a SPARQL query to each endpoint. If at least one of these endpoints has instances in v_i but not in v_j, then v is a GJV. At each endpoint, we check for each data instance appearing as an object in TP_i whether this instance appears locally as a subject in TP_j. Once a common variable is found to be a GJV, the triple patterns cannot be combined in the same subquery even for those endpoints that return an empty result for the difference in the instances, e.g., the pair of triples where ?U is a common variable in Figure 5. This allows us to have simple plans and better parallel execution.

The set difference operator ($-$) is implemented using *FILTER NOT EXISTS* (Figure 6) where TP_i: ⟨?S, Predicate$_i$, ?P⟩, and TP_j: ⟨?P, Predicate$_j$, ?C⟩. If there is a triple pattern setting a type for v (⟨?P, rdf:type, T⟩), we use it to limit the check to only the relevant values of v. Since Lusail needs to only know whether the result is an empty set, we use *LIMIT 1*. Note that our goal is to check whether there are data instances matching the predicate j regardless of the actual values. So, if the query specifies a constant in the triple pattern mentioned in the filter of the check query (i.e., 'XXX'), Lusail will replace it with a variable in the check query.

Objects/Subjects Only. If a variable appears only as *object*, respectively *subject*, in both triple patterns TP_i and TP_j, Lusail checks in each relevant endpoint that $v_i - v_j$ and $v_j - v_i$ are both empty. As shown in Figure 5, the variable ?S appears as subject in both ⟨?S, ub:advisor, ?P⟩ and ⟨?S, ub:takesCourse, ?C⟩. Having two empty sets in the same endpoint means that (*i*) any graduate student ?S having an advisor ?P should take a course ?C and (*ii*) any graduate student ?S taking a course ?C should have an advisor ?P, all located in the same endpoint.

Algorithm 1: Detecting Global Join Variables

Input: Input query (Q), Set of relevant sources ($Sources$)
Result: List of Global Join Variables (V)

```
1  Triples ← Q.getTriplePatterns();
2  vars ← getJoinEntities (Triples);
3  chkQueries ← ∅;
4  V ← ∅;
5  foreach var_i in vars do
6      pairWiseTriples ←getPairTriples (var_i.Triples);
7      joinVar ← False;
8      foreach pair_i in pairWiseTriples do
9          if pair_i[0].sources ≠ pair_i[1].sources then
10             V.addJoinVar (var_i.varName, pair_i);
11             joinVar ← True;
12     if joinVar is True then continue ;
13     if var_i is subject only || var_i is object only then
14         chkQueries ← formulatePairWiseQuery (var_i,
           pairWiseTriples);
15     if var_i is subject and object then
16         chkQueries ← formulateSubjObjQuery (var_i,
           var_i.subjTriples, var_i.objTriples);
17 if chkQueries is not empty then
18     ReqHandler ← initializeRequestHandler (thrdPoolSize,
       Sources);
19     foreach chkQry_i in chkQueries do
20         //each chkQry_i is attached with its relevant sources;
21         RES = ReqHandler.executechkQAtRelSrcs (chkQry_i);
22         if RES is not empty then
23             V.addJoinVar (chkQry_i.varName,
               chkQry_i.triples);
24 return V;
```

Correctness and False Positives. In some cases, LADE may detect a join variable as being global while the triple patterns sharing this variable could be solved together locally at the endpoints. Consider for example the variable ?P in ⟨?S, ub:advisor, ?P⟩ and ⟨?P, ub:teacherOf, ?C⟩. As shown in EP1 in Figure 5, there is an advisor (Ann) who has joined MIT but is not a teacher of any course yet. ?P will be considered as a GJV. It is clearly safe to send both triple patterns together in the same subquery since it does not need to access data in remote endpoints. Such "false positives" may lead Lusail to generate a query plan with unnecessary GJVs, i.e., more remote requests and more join computations at the Lusail server rather than at the endpoints. This, however, does not affect the correctness of the results.

B. Detecting Global Join Variables

Algorithm 1 is used to detect GJVs. It receives a query and a list of relevant endpoints and outputs a set of GJVs (V) along with the triple patterns that caused each variable to be a GJV. It assumes that source selection is already done using ASK requests or the Lusail cache. The algorithm starts by retrieving the set of query variables and triple patterns. Each variable is associated with its subject and object patterns (line 2). The algorithm iterates over the variables to detect GJVs.

If the variable joins triple patterns from different sources, then it is a GJV (lines 8-11). There is no need to check the other conditions. Otherwise, Algorithm 1 formulates a set of check queries as discussed in Section IV-A. For the *only object* or *only subject* cases, Algorithm 1 formulates check queries for all possible pairwise combinations of the triple patterns associated with the variable (lines 13-14). For the *object and subject* case, the check query is a combination of object triples and subject triples (lines 15-16).

The algorithm utilizes the *elastic request handler* (Figure 4)

Algorithm 2: Query Decomposition

```
   Input: Input query (Q), set of Join Vars (V)
   Result: Set of independent subqueries
1  subqueries ← ∅;
2  if V is empty then return subqueries.add (Q);
3  Triples ← Q.getTriplePatterns();
4  VisitedTriples ← ∅;
5  Nodes ← ∅;
6  foreach jvar_i in V do
7      Nodes.push (jvar_i);
8      while Nodes is not empty do
9          vrtx ← Nodes.pop ();
10         edges ← vrtx.edges ();
11         if subqueries is empty then
12             foreach edge_i in edges do
13                 if visited (edge_i, VisitedTriples) then continue ;
14                 sq ← createSubquery(edge_i);
15                 subqueries.add (sq);
16                 Nodes.push (edge_i.destNode);
17                 VisitedTriples.add (edge_i);
18             continue;
19         parentSq ← getParentSubquery (vrtx, subqueries);
20         foreach edge_i in edges do
21             if visited (edge_i, VisitedTriples) then continue ;
22             if canBeAddedToSubQ (parentSq, edge_i, V) then
23                 parentSq ← addToSubquery (parentSq, edge_i);
24             else
25                 sq ← createSubquery(edge_i);
26                 subqueries.add (sq);
27             Nodes.push (edge_i.destNode);
28             VisitedTriples.add (edge_i);
29     if VisitedTriples ≡ Triples then break ;
30 mergeSubQ (subqueries);
31 return subqueries;
```

to execute check queries. It initializes the handler with the size of the thread pool and the set of endpoints (line 18). Then, it iterates over all check queries and executes each at the relevant endpoints (lines 19-23). If the query returns any results, then the corresponding variable is a GJV (lines 22-23). The algorithm returns the set of GJVs along with the *triple patterns* that caused each variable to be a GJV.

Assuming that $|V|$ is the number of variables appearing in more than one triple pattern in the query and $|T|$ is the number of triples, the maximum number of check queries, C_Q, is bound by O($|V|*|T|^2$). The number of triple patterns in real-world SPARQL queries is usually small [12] and hence the number of GJVs is also small. Therefore, C_Q will be typically small as well. Thus, LADE creates a maximum of $N * C_Q$ HTTP requests. A request is made to evaluate each check query with respect to its relevant endpoints (N). These check queries are lightweight and have a minimal overhead (see Section VI-B). Note that the number of HTTP requests in the state-of-the-art algorithms are dominated by the intermediate data, which is a huge overhead compared to $N * C_Q$.

C. Query Decomposition

Algorithm 2 decomposes a query into multiple subqueries to be sent to different endpoints. If query Q is disjoint, i.e., with no GJVs, the algorithm returns a set containing only Q (line 2). Otherwise, LADE utilizes the set of GJVs and the source selection information to decompose Q.

Algorithm 2 has two phases: branching (lines 6-29) and merging (lines 30). In the branching phase, the algorithm starts by building a query tree rooted at one of the GJVs (line 7). It then traverses the query tree to create subqueries. If all the triple patterns have been assigned to one of the subqueries

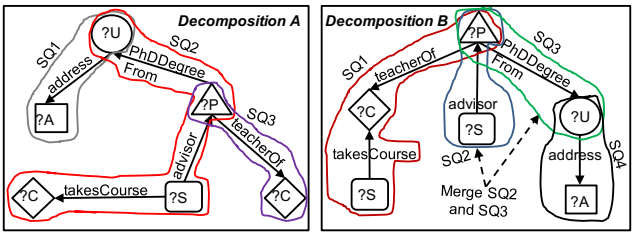


Fig. 7. Two possible decompositions of Q_a, where the GJVs are ?U and ?P. Any pair of predicates, which causes a variable to be a GJV, cannot be in the same subquery.

(lines 29), then the merging phase starts (lines 30). Otherwise, the algorithm creates another query tree rooted at the next GJV.

An initial set of subqueries is created at the root of the query tree, one subquery per child (lines 11-18). Each subquery is expanded through depth first traversal (lines 19-28). A triple pattern can be included in a subquery (lines 22-23) if both the subquery and the triple pattern have the same relevant sources, and the triple pattern with any of the subquery triple patterns did not cause a query variable to be a GJV. If one of the conditions is invalid, a new subquery is created from the current triple pattern and added to the set of subqueries (lines 24-26). In both cases, the edge destination node is added to the nodes stack and the edge itself is marked as visited (lines 27-28).

The function $mergeSubQ$ (line 30) loops through the set of subqueries and merges a pair of subqueries together if they have common variables, the same relevant sources, and no triple patterns from both subqueries caused a variable to be a GJV in one of them. Afterwards, the algorithm continues checking the subsequent subqueries.

Decomposing the query into subqueries is similar to a depth first traversal. Thus, its runtime complexity is $O(|V| + |T|)$ where $|V|$ is the total number of query variables and $|T|$ is the number of triple patterns. The outer loop can repeat the traversal for few more times if there are triple patterns that were not visited yet. However the algorithm complexity is still bound by $O(|V| + |T|)$.

Result Completeness. In Lusail, all triple patterns of a given query will be sent to all relevant endpoints. The optimization introduced by LADE assigns these triple patterns to different subqueries based on the concept of locality. The risk of missing results will be in the case of a subquery containing a set of triple patterns that will miss some results. This can only happen if the subquery contains triple patterns that access predicates through interlinks, e.g., a subquery that contains ⟨?P, ub:PhDDegreeFrom, ?U⟩ and ⟨?U, ub:address, ?A⟩ will miss the tuple (Kim, Tim, MIT, "XXX") when submitted to EP2. Since LADE puts triple patterns into the same subquery only if the data instances matching them are located in the same endpoint, such cases cannot happen. Figure 7 shows two possible decompositions for Q_a (Figure 2), which has two GJVs, namely ?U and ?P. The generated set of subqueries may change depending on the order in which variables are selected during query decomposition (line 6 in Algorithm 2). All decompositions (i.e., all sets of subqueries) will produce the same result set and will not miss any result triples, as we just explained, but some decompositions may generate more intermediate results and thus place a higher processing load on Lusail. In this paper, we rely on SAPE to order the subqueries at run time in an efficient way, and this works well

in our experiments (details in Sections V and VI). Choosing the best decomposition at compile time to minimize intermediate results is left for future work.

Generic SPARQL Queries. So far, we only discussed how Lusail evaluates conjunctive SPARQL queries. However, Lusail also supports queries with joins on variable predicates, UNION, FILTER, LIMIT, and OPTIONAL statements, see the listed quereis in [11]. In a nutshell, Lusail determines where to add clauses, such as FILTER, LIMIT, and OPTIONAL, in the subqueries or during the global join evaluation. For example, FILTER statements with a single variable are pushed with relevant subqueries and thus handled by the endpoints. In case of multi-variable filters, if both variables exist exclusively in the same subquery, then they are handled by the endpoints. Otherwise, Lusail considers the filter clause during the join evaluation phase.

V. Selectivity-Aware Query Execution

SAPE is responsible for choosing: (*i*) a good execution order for the subqueries that would balance between the communication cost and the degree of parallelism and (*ii*) a good join order for the subquery results. Lusail also supports multi-query optimization and multi-machine execution [11].

Figure 8 gives an overview of our Selectivity-Aware Planning and parallel Execution (SAPE) algorithm. SAPE estimates the cardinality of the different subqueries and accordingly delays subqueries expected to return large results. Non-delayed subqueries are evaluated concurrently while the delayed ones are evaluated serially using bound joins. The objective is to maximize the degree of parallelism and to minimize the communication cost in terms of the number of HTTP requests and subquery results. To avoid an increase in the number of HTTP requests due to bound joins, we simply group these bindings into blocks and submit one HTTP request per block.

A. Subquery Ordering and Cost Model

LADE outputs a set of independent subqueries that can be submitted concurrently for execution at each of its relevant endpoints. The results of these subqueries will then need to be joined at the global level. There are two extreme approaches to execute these subqueries.

The simplest approach is to simultaneously submit the subqueries to the relevant endpoints and wait for their results to start the joining phase. For example, the subqueries of Figure 7 will be executed concurrently and after receiving all their results, a join phase is started. Notice that the subquery ⟨`?U`, address, `?A`⟩ is so generic and executing it independently will retrieve all entities with addresses regardless of whether these entities match `?U` in the remaining subqueries, see Figure 1. Such case is evident in subqueries that touch most of the endpoints or retrieve large amounts of intermediate results.

Such subqueries affect the query evaluation time by overwhelming the network, the endpoints, and Lusail, with irrelevant data. Examples include: (*i*) generic subqueries that are relevant to the majority of the endpoints, e.g., common predicates such as *owl:sameAs*, *rdf:type*, *rdfs:label* and *rdfs:seeAlso*. (*ii*) Simple subqueries that have one triple pattern with two or three variables, e.g., ⟨`?s`, ?p, `?o`⟩ or ⟨`?s`, owl:sameAs, `?o`⟩, and (*iii*) optional subqueries, e.g., the drug query in [11].

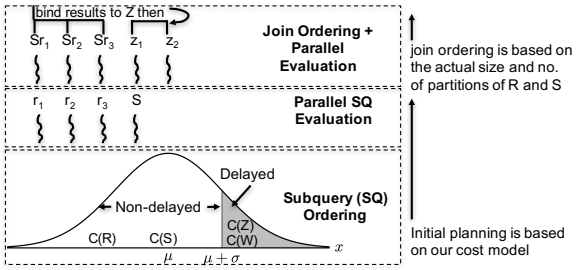


Fig. 8. Query Evaluation in Lusail

At the other extreme, we submit the most selective subquery first and then use the actual bindings of the variables that have been already obtained to submit the next most selective subquery and so on. While limiting the amount of intermediate results to be retrieved from the endpoints, this approach suffers from long communication delays and almost no concurrency except submitting the same subquery to multiple endpoints.

To overcome the shortcomings of these two approaches, we need to balance between the degree of parallelism, i.e., the number of subqueries that can be submitted concurrently, and the communication cost which is dominated by the size of intermediate results. Thus, we need to detect the subqueries expected to return significantly less results if some of their variables are bound to the already obtained results. The idea is to cluster subqueries based on their estimated cardinality and the number of endpoints they access while taking into account the variability in these values. We introduce the concept of *delayed subqueries*. A delayed subquery is evaluated using the actual bindings of the variables that have been already obtained. We thus follow a two-phase subquery evaluation: (*i*) concurrently submit non-delayed subqueries for evaluation at the endpoints, and (*ii*) use the variable bindings obtained from the first phase to evaluate the delayed subqueries.

We introduce a cost model to determine delayed and non-delayed subqueries. SAPE assumes that subquery cardinalities follow a normal distribution, i.e., most subqueries return results whose sizes are within one standard deviation of the mean. SAPE calculates the average μ and standard deviation σ values for all the cardinalities and for all the numbers of relevant endpoints per subquery. The outliers, e.g., subqueries returning extremely huge results (very low selectivity), or accessing a large number of endpoints, compared to other subqueries, misleadingly increase the standard deviation. This may lead to consider some subqueries, which are better to be delayed, as non-delayed. We therefore apply the Chauvenet's criterion [13] for detecting and rejecting outliers before computing μ and σ. Any subquery sq_i with cardinality $C(sq_i) > \mu_C + \sigma_C$ is delayed as shown in Figure 8. We apply the same concept for the number of relevant endpoints per subquery. With this heuristic, only subqueries (including the outliers) whose results are expected to be significantly bigger than the results of the majority of the subqueries will be delayed.

The cardinality of a subquery is estimated based on the cardinality of its triple patterns. It is collected during the query analysis phase using a simple SELECT COUNT query, one per triple pattern. Whenever a filter clause is available for a subject and/or object, it is pushed with the statistics query to obtain better cardinality estimations. Note that cardinality statistics per predicate are usually collected by RDF engines for their runtime query optimization [14], [15], [16]. Therefore, this

Algorithm 3: Subqueries Evaluation

Input: Subqueries list ($subQs$), relevant sources ($srcs$)
Result: The final query results ($qResult$)

1 $ReqHandler \leftarrow$ `initializeRequestHandler` ($srcs$);
2 **if** $subQs$.`size ()=1` **then**
3 $ReqHandler$.`executeSubQAtRelSrcs` ($subQs$[0]);
4 **return** `concatEndptsResults` ($ReqHandler$);
5 $foundBindings \leftarrow$ Empty;
6 **foreach** *sq in* $subQs.nonDelayed$ **do**
7 $ReqHandler$.`executeSubQAtRelSrcs` (sq);
8 $sqsRes \leftarrow$`joinSubqsResults` ($ReqHandler.threads$);
9 `updateFoundBindings` ($subqRes$, $foundBindings$);
10 **while** $subQs.delayed$ *is not empty* **do**
11 $sq \leftarrow$ `getMostSelectiveSubq` ($subQs$, $foundBindings$);
12 $boundSubQs \leftarrow$ `formulateBoundSubqs` (sq, $foundBindings$);
13 sq.`relSrcs` $\leftarrow$ `refineRelSrs` (sq.`relSrcs`, $foundBindings$);
14 $sqRes \leftarrow$ Empty;
15 **foreach** $boundSubq_i$ *in* $boundSubQs$ **do**
16 $sqRes = sqRes \cup ReqHandler$.`executeSubQAtRelSrcs` ($boundSubq_i$);
17 `updateFoundBindings` ($subqRes$, $foundBindings$);
18 $subQs.delayed$.`remove` (sq);
19 **return** `joinSubqsResults` ($ReqHandler$);

count query is lightweight as it asks for a simple triple-by-triple statistics.

We need to estimate the cardinality of the variables in the projection list of each subquery. The cardinality of a variable v in a subquery sq_i is denoted as $C(sq_i, v)$ and represents the number of bindings of v. If two triple patterns TP_i and TP_j join on a variable v, then the number of bindings of v at endpoint ep_k after the join will be:

$$C(sq_i, v, ep_k) = min(C(TP_i, ep_k), ..., C(TP_j, ep_k))$$

Therefore, we use the minimum cardinality of the predicates, in which v is a common variable, as an upper bound of the cardinality of v per endpoint. Thus, the total cardinality of v in the subquery sq_i is the sum of its cardinalities in all the relevant endpoints ep, estimated as:

$$C(sq_i, v) = \sum_{ep \in srcs(sq_i)} C(sq_i, v, ep)$$

The cardinality of a subquery sq_i, denoted as $C(sq_i)$, is the maximum cardinality of the subquery projected variables.

B. Evaluation of Subqueries

Different orders of delayed subquery evaluation can result in different computation and communication costs. Our query planner tries to find an order of subqueries that has the minimal cost. Given a set of non-delayed subqueries, SAPE evaluates them concurrently and builds a hashmap that contains the bindings of each variable. As a result, SAPE knows the exact number of bindings of each subquery variable. Then, we refine the cardinality of the delayed subqueries based on the cardinality of variables they can join with. The first delayed subquery to be evaluated is the one with the lowest cardinality.

Given a set of binding values, SAPE divides them into fixed size blocks and creates a subquery for each block. The data block size is a SAPE configuration parameter. Once the first subquery is selected, it is evaluated at the corresponding endpoints and its results are used to update the bindings hashmap. SAPE continues to select the next subquery to be evaluated till all subqueries are executed.

Algorithm 3 describes our selectivity-aware evaluation technique for subqueries. The input is a set of independent subqueries with their delay decisions. Each subquery contains its triple patterns, the relevant endpoints (sources), the projection variables, and whether the subquery is optional. The algorithm initializes the request handler which creates a thread per relevant endpoint (line 1). If there is only one subquery; i.e the query is disjoint, the entire query can be evaluated independently at each endpoint and no global join is required. The algorithm evaluates the whole query at all relevant endpoints independently (line 3). Then, it simply concatenates the results obtained from relevant endpoints and returns the final query result (line 4).

If the query is not disjoint, SAPE iterates over all input subqueries and evaluates each subquery at its relevant endpoints (lines 6-19). In the first phase, non-delayed subqueries are evaluated and their results are collected concurrently (lines 6-7). This step is non-blocking, i.e, each thread is assigned all relevant subqueries at the same time. Whenever possible, the results of non-delayed subqueries are joined together (join evaluation is discussed next). This step is necessary to obtain a reduced set of found bindings. In the second phase, SAPE evaluates the delayed subqueries using the found bindings (lines 10-18). SAPE selects the next delayed subquery to be the one with the smallest estimated cardinality (line 11). SAPE formulates a set of modified subqueries from the subquery itself using the found bindings (line 12). It appends a data block to the subquery using the SPARQL VALUES construct, which allows multiple values to be specified in the data block. If the subquery contains triple patterns of the form $\langle$`?s`, ?p, `?o`$\rangle$, the source selection process is repeated using the found bindings to reduce the number of relevant endpoints (line 13). Without this refinement, such subqueries are relevant to all endpoints. We empirically verified that the source selection refinement step on irrelevant endpoints using ASK queries costs significantly less than evaluating the delayed subquery with the found bindings. Finally, the bound subqueries are evaluated and their results are merged (lines 15-16). SAPE updates the set of found bindings using the current subquery results (line 17). After that, the evaluated subquey is removed from the delayed subqueries list (line 18). SAPE continues to evaluate the other subqueries until no more delayed subqueries are left.

Join Evaluation. At this stage, each endpoint thread maintains a set of relevant subqueries and their corresponding results. This information is encapsulated in the request handler object which is then passed to the threads performing the joins (line 19). Each subquery corresponds to a relation (R) for which we know the true cardinality and is partitioned among a set of threads. The join evaluation algorithm has four main steps: (*i*) for each subquery, it collects aggregate statistics (relation size and number of partitions) from all threads. (*ii*) it then uses a cost-based query optimizer based on the Dynamic Programming (DP) enumeration algorithm [17]. It considers both the inter-operator parallelism where the same operator is being executed concurrently by multiple threads and the selection of the join which leads to the smallest cardinality. Using an in-memory hash join algorithm, joining the subplan at state S with another relation R has two phases; hashing and probing. Assuming that S is the smaller relation, the join cost is estimated as follows:

$$JoinCost(S, R) = \underbrace{\frac{1}{S.threads}|S|}_{hashing} + \underbrace{\frac{1}{R.threads}C(R, v)}_{probing}$$

All threads that have the smaller relation build a hash table

for their part of S. Then the threads that maintain R will evaluate the join by probing the built hash tables with the found bindings of the join variables. (*iii*) Given the devised join order, SAPE joins the different subqueries together to produce the query answer, and (*v*) finally, SAPE aggregates the joined results from the individual threads and returns the result.

VI. Experimental Study

A. Evaluation Setup

Compared Systems: We evaluate Lusail extensively against: one index-free system, FedX [6], and two index-based systems, SPLENDID [7] and HiBISCuS [8]. In a recent study [9], FedX performed better than other systems on the majority of queries and datasets. HiBISCuS [8] is an add-on to improve performance. In our experiments, we use it on top of FedX. SPLENDID showed competitive performance to FedX on several queries in [9] and LargeRDFBench[5]. Similarly to Lusail, both FedX and SPLENDID support multiple-threads.

Computing Infrastructure: We used two settings for our experiments: two local clusters, *84-cores* and *480-cores*, and the public cloud. The *84-cores* cluster is a Linux cluster of 21 machines, each with 4 cores and 16GB RAM, connected by 1Gb Ethernet. The *480-cores* cluster is a Linux cluster of 20 machines, each with 24 cores and 148GB RAM, connected by 10Gb Ethernet. We use the *84-cores* cluster in all experiments except those that need 256 endpoints for the LUBM dataset. For the public cloud, we use 18 virtual machines on the Azure cloud to form a real federation.

Datasets: We use several real and synthetic datasets. Table I shows their statistics. QFed [10] is a federated benchmark of four different real datasets. Although the total number of triples used in QFed is only 1.2 million, there are interlinks between the four datasets, which makes federated query evaluation challenging. LargeRDFBench is a recent federated benchmark of 13 different real datasets with more than 1 billion triples in total. We also used the synthetic LUBM benchmark to generate data for 256 universities, each with around 138K triples. It includes links between the different universities through students and professors.

Queries: QFed [10] has different categories of queries. Each query has a label *C* followed by the number of entities for each class, and a label *P* followed by the number of predicates linking different datasets. LUBM comes with its benchmark queries. We only used the queries that access multiple endpoints. Queries *Q1*, *Q2*, and *Q3* correspond to *Q2*, *Q9*, and *Q13* in the benchmark while *Q4* is a variation of *Q9*, it retrieves extra information from remote universities. LargeRDFBench has three categories: simple *S*, complex *C*, and large *B*. LargeRDFBench subsumes the FedBench benchmark[6], as the simple category use the 14 FedBench queries. The complex category contains 10 queries with a high number of triple patterns and advanced SPARQL clauses. The last category has 8 large queries that generate large amounts of intermediate data and final results. We exclude three queries (*C5* , *B5* and *B6*) as they contain two disjoint subgraphs joined by a filter variable. Neither Lusail nor its competitors support this kind of queries. We show query samples from each benchmark in [11].

[5]https://github.com/AKSW/LargeRDFBench

[6]A popular benchmark for federated SPARQL, https://code.google.com/p/fbench/

TABLE I. Datasets used in experiments

Benchmark	Endpoint	Triples
QFed	DailyMed	164,276
	Diseasome	91,182
	DrugBank	766,920
	Sider	193,249
	Total Triples	1,215,627
LargeRDFBench	LinkedTCGA-M	415,030,327
	LinkedTCGA-E	344,576,146
	LinkedTCGA-A	35,329,868
	ChEBI	4,772,706
	DBPedia-Subset	42,849,609
	DrugBank	517,023
	Geo Names	107,950,085
	Jamendo	1,049,647
	KEGG	1,090,830
	Linked MDB	6,147,996
	New York Times	335,198
	Semantic Web Dog Food	103,595
	Affymetrix	44,207,146
	Total Triples	1,003,960,176
LUBM	256 Universities	35,306,161

Endpoints: We use Jena Fuseki 1.1.1 as the SPARQL engine at the endpoints for LUBM and QFed. Since Jena run out of memory while indexing the endpoints of LargeRDFBench, we used a Virtuoso 7.1 instance for each of the 13 endpoints in LargeRDFBench. The standard, unmodified installation of each SPARQL engine is run at the endpoints and used by all federated systems in our experiments.

Data Preprocessing Cost: Index-based systems such as SPLENDID and HiBISCuS require a preprocessing phase that generates summaries about the data schemas and collects statistics that are used during query optimization. In real applications, endpoints might not allow collecting these statistics. Moreover, it is a time consuming process dominated by the dataset size. For example, SPLENDID needs 25 and 3,513 seconds to pre-process QFed and LargeRDFBench respectively. In contrast, Lusail and FedX do not require any preprocessing. Hence, index-free methods are preferred in a large scale and dynamic environment, since endpoints can join and leave the federation at no cost.

In the rest of this section, we first analyze the effect of using different threshold values for delayed queries detection. Then, we profile the different costs of Lusail's query processing. After that, we present the results of our evaluation on the local clusters and then on a geo-distributed settings where the endpoints are located on a public cloud. We also present an experiment on independently deployed endpoints. For more experiments on multi-query optimization, multi-machine execution, and how each of LADE and SAPE contribute to the performance of Lusail the reader is referred to [11][7].

B. Delayed SubQueries and Profiling Lusail

Delayed Subqueries: This experiment evaluates different values for the threshold used to delay subqueries, these are μ, $\mu + \sigma$, and $\mu + 2\sigma$, in addition to delaying only subqueries with outlier estimated cardinalities. We used the Chauvenet criterion [13] for outlier detection. We deployed the LargeRDFBench endpoints on 13 D4 instances (8 Cores, 28 GB memory) on Microsoft Azure distributed among 7 regions in the USA and Europe. We report the total time for evaluating

[7]While these are not part of the core contributions of this paper, they are supported features by Lusail.

the queries of each category in LargeRDFBench. Figure 9 shows that for *simple* and *complex* queries, $\mu+2\sigma$ and *outliers* allowed most subqueries to be evaluated concurrently and only few of them to be delayed. Hence, they missed the opportunity to delay some subqueries that could reduce the communication cost and hence the cost of joining the fetched data. Thus, $\mu + 2\sigma$ and *outliers* achieved significantly worse than μ and $\mu+\sigma$. For *large* queries, delaying several subqueries limits the parallelism. Thus, μ achieved significantly worse than others as fewer subqueries were evaluated concurrently while the rest were delayed. As shown, $\mu + \sigma$ consistently performs well in all the three categories and hence we use it in our system.

Profiling Lusail: Lusail has three phases: source selection, query analysis using LADE, and query execution using SAPE. In this experiment, we profile these phases while varying the query complexity and data size. We use LargeRDFBench queries with different complexities, simple (*S10*), complex (*C4*), and large (*B1*). Lusail is deployed on a single machine of the *84-cores* cluster. The results are shown in Figure 10(a). Source selection and query analysis require a small amount of time compared to query execution, especially for *C4* and *B1*. As expected, the total response time is dominated by the query execution phase. Lusail's query analysis phase is lightweight, requiring less time than the source selection phase in *S10* and *C4*. *B1* requires performing a union operation between two triple patterns and retrieves its data from the endpoints with the largest data sizes. Hence, the query analysis phase takes slightly more time than the source selection phase.

The cost of query processing in Lusail also depends on the number of endpoints and the sizes of the datasets. Therefore, we profiled Lusail while varying the number of endpoints, which also increases the data size. LUBM allows us to increase both endpoints and data size in a systematic way by adding more universities. We deployed 256 university endpoints on the *480-cores* cluster. Lusail is deployed on one machine in the same cluster.

Figures 10(b) and 10(c) show the time required for each phase of *Q3* and *Q4*, respectively. Both queries join data from different endpoints to produce the final result. Lusail's query analysis is lightweight, especially for *Q3* since it has only two triple patterns. For *Q3*, Lusail detects the GJVs using the source selection information, i.e., it does not need to communicate with the endpoints. Source selection time is substantial for these queries and increases slightly as the number of endpoints increases. Query execution time is the dominant factor as the number of endpoints increases. The figure shows the total query response time with and without caching the results of *ASK* and *check* queries. The cache helps, especially for the more complex *Q4* and when the number of endpoints is large.

Summary. These experiments demonstrate that Lusail's query analysis does not add significant overhead to the total query response time, and that Lusail's cache helps. In all subsequent experiments, Lusail as well as its competitors are allowed to cache the results of the source selection phase. Furthermore, we run each query three times and report their average.

C. Query Performance on a Local Cluster

We compare Lusail to FedX, HiBISCuS, and SPLENDID. They are all deployed on one machine of the *84-cores* cluster.

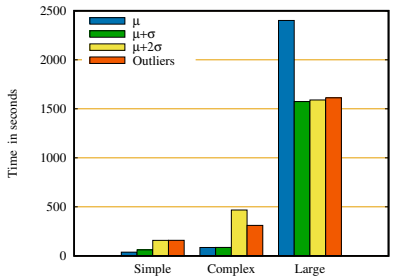


Fig. 9. Evaluating different threshold values for delayed subquery detection.

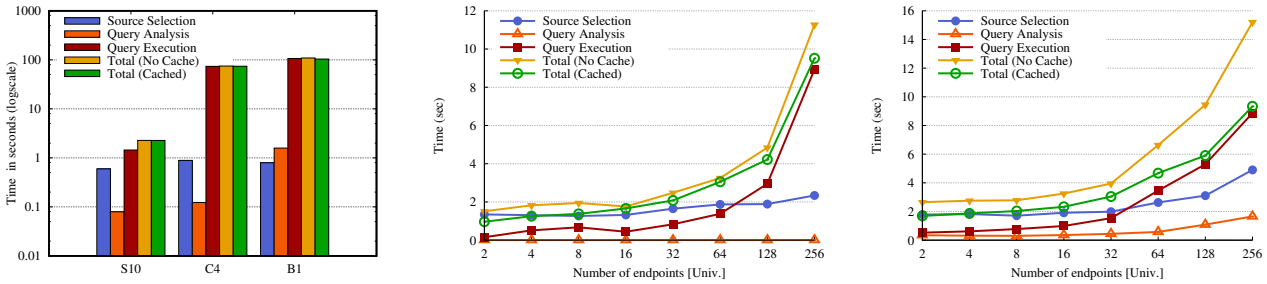
The endpoints are also deployed on the same cluster.

QFed Dataset: Figure 11 shows the query performance of Lusail compared to FedX and HiBISCuS. SPLENDID timed out in all QFed queries except *C2P2* which is answered in 56 seconds. Lusail achieves better performance than FedX and HiBISCuS for all queries. Queries with filter, namely *C2P2BF*, *C2P2BOF*, *C2P2F* and *C2P2OF*, have high selectivity, i.e., less intermediate data. Hence, most of these queries are answered within a few seconds. Lusail is up to six times faster than other systems for these queries. Using big literal object (*C2P2B*, *C2P2BO*) increases the volume of communicated data. Hence, FedX and HiBISCuS timed out after one hour in *C2P2BO*, while FedX took significant time to evaluate *C2P2B*, on which HiBISCuS timed out. This is due to the large size of communicated data and the number of remote requests. Lusail successfully answers both these queries in less than 2 seconds.

LUBM Dataset: This experiment utilizes up to four university datasets[8] from the LUBM benchmark, each in a different endpoint. Figures 12(a) and 12(b) show the results using two and four endpoints, respectively. The datasets at the endpoints have the same schema. Therefore, FedX and HiBISCuS cannot create exclusive groups. Instead, a subquery is created per triple pattern and is sent to all endpoints. Bound joins are then formulated using all the results retrieved from the different endpoints. This leads to a huge amount of remote requests. Lusail utilizes the schema as well as the location of data instances accessed by the query to formulate the subqueries. Therefore, Lusail discovered that both *Q1* and *Q2* are disjoint queries and their final results can be formulated by sending the whole query to each endpoint independently.

Q3 and *Q4* need to join data from different endpoints. *Q3* finds graduate students who received their undergraduate degree from *university0*. This limits the size of intermediate data and the number of endpoints. Hence, FedX and HiBISCuS evaluated this query on two and four different endpoints. Lusail decomposed this query into two subqueries: the first subquery (those who obtained an undergraduate degree from *university0*) is sent to the relevant endpoint. The second subquery contains only ⟨`?x`, rdf:type, `ub:GraduateStudent`⟩, which is relevant to all endpoints. Hence, Lusail decided to delay its evaluation and managed to outperform the other systems on four endpoints. Lusail decomposes *Q4* into two subqueries, with the second subquery delayed until the results of the first subquery are ready. Lusail decides the join order of the subquery results based on the cardinality of each result and the number of threads hosting this result (in order to achieve a high degree of parallelism). The figures illustrate that Lusail is up

[8]The competitors do not scale to more than four while Lusail (Figures 10(b) and 10(c)) scales to 256 universities.



(a) Varying Query Complexity (b) Varying Data Size (LUBM): Q3 (c) Varying Data Size (LUBM): Q4

Fig. 10. Profiling Lusail by varying query complexity, the number of endpoints, and the data size.

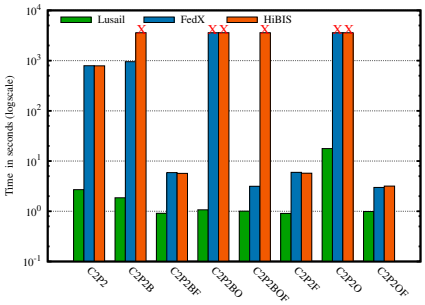
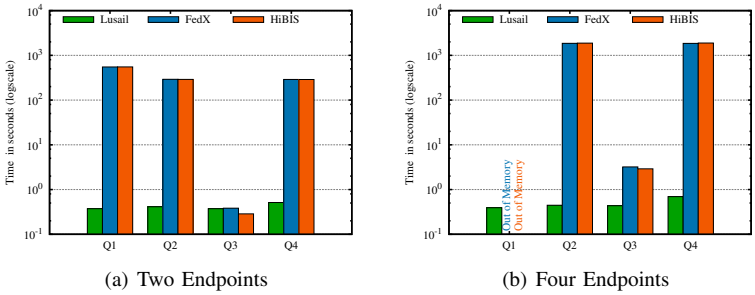


Fig. 11. Queries with Filter have high selectivity while Big literal queries have bigger intermediate data.



(a) Two Endpoints (b) Four Endpoints

Fig. 12. FedX and HiBISCuS evaluate queries one triple pattern at a time in a bound join. Lusail decomposes these queries based on the location of data instances.

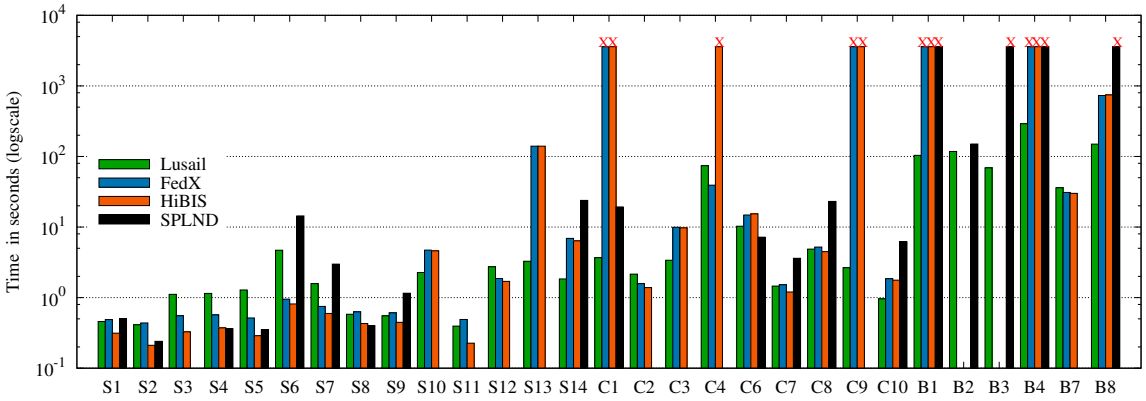


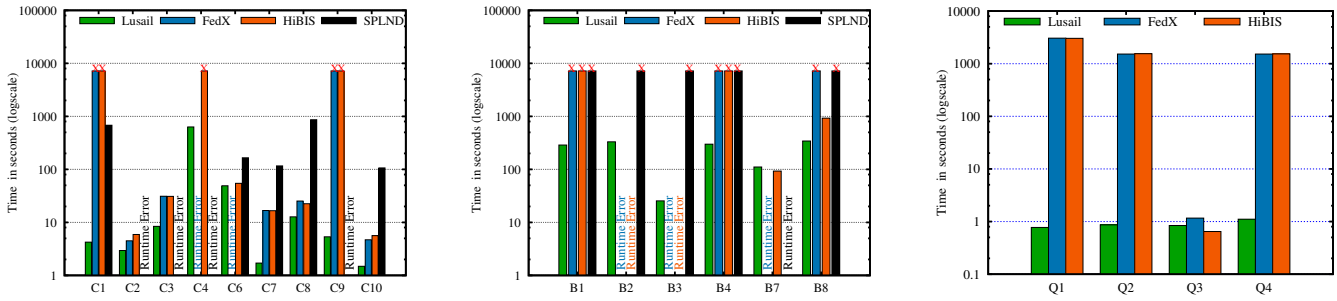
Fig. 13. LargeRDFBench: Most of the simple queries do not access large intermediate data, unlike the complex and large queries. X corresponds to time out while missing bars correspond to runtime errors.

to three orders of magnitude faster than FedX and HiBISCuS for queries *Q1*, *Q2*, and *Q4*. FedX and HiBISCuS ran out of memory for *Q1* on four endpoints. SPLENDID managed to run only *Q3* on four endpoints within 52 seconds, which is significantly larger than all other systems.

LargeRDFBench Dataset: Figure 13 shows the response times for each system on each query category in LargeRDFBench. The performances of Lusail and FedX are comparable for most of the simple queries. The collected statistics by HiBISCuS and SPLENDID do not guarantee better response times in all cases even for simple queries. For example, HiBISCuS is much slower than Lusail for *S13* and *S14*, and SPLENDID has the worst performance in *S6*, *S7*, *S9*, and *S14*. In queries *S13* and *S14*, Lusail is significantly faster than FedX and HiBISCuS since these two queries return relatively large intermediate results. In general, the simple queries do not access huge intermediate data and target datasets of different schema. Hence, Lusail's optimizations do not result in performance improvement especially against index-based systems. However, index-based systems require preprocessing and cannot add endpoints on demand.

The complex and large queries have on average a larger number of triple patterns per query and access a larger amount of intermediate data. Lusail achieves significantly better performance than other systems for most of the complex queries (Figure 13). Both FedX and HiBISCuS could not finish the evaluation of *C1*, and *C9* within an hour. SPLENDID evaluated only 5 out of the 10 complex queries. *C2* is a selective query returning 4 results, which explains why all systems have comparable performance. FedX achieved the best performance for *C4* followed by Lusail, while HiBISCuS could not evaluate the query within one hour. *C4* contains a LIMIT clause of 50 results. Lusail current implementation uses a naive approach for the LIMIT clause. It computes all the final results and returns only the top 50 results. FedX cuts short the query execution once the first 50 results are obtained, hence FedX outperformed Lusail in *C4*. SPLENDID achieved the best performance in only *C6* on which other systems have comparable performance.

Lusail is by far the best system for all big queries. FedX and HiBISCuS timed out in two queries and returned no results on another two queries. SPLENDID succeeded on only one



(a) LargeRDFBench: Complex Queries (b) LargeRDFBench: Large Queries (c) LUBM Queries: Two Endpoints

Fig. 14. Geo-distributed federation: endpoints are deployed in 7 different regions of the Azure cloud. Communication cost affects all systems, but Lusail can execute all queries and outperforms other systems.

query (*B2*) and timed out in the rest.

Summary. Lusail is the only system that successfully executes all queries of LargeRDFBench, often showing orders of magnitude better performance. It is never significantly outperformed by other systems. In contrast, other systems time out or throw runtime errors in addition to their performance being highly variable and highly unpredictable.

D. Evaluation on a Geo-distributed Settings

In this section, we evaluate Lusail by simulating a real scenario on the cloud as well as using real endpoints.

Using the Microsoft Azure cloud: We simulate a real setting where Lusail and endpoints are deployed on 7 different regions in the USA and Europe of the Azure cloud. We used 18 D4 instances (8 Cores, 28 GB memory), 13 for the LargeRDFBench endpoints and four for the LUBM and QFed datasets, interchangeably. Lusail and its competitors are deployed on a D5_V2 instance (16 Cores, 56 GB memory) in Central USA, while none of the 18 instances is located in Central USA.

The communication cost imposed a clear overhead. For QFed, neither FedX nor HiBISCuS were able evaluate most of the queries. FedX finished only *C2P2BF* in 23 seconds, compared to 1.9 seconds for Lusail, while HiBISCuS finished only *C2P2* in 4,477 seconds, compared to 9.5 seconds for Lusail. Figures 14(a) and 14(b) show the query response times of both complex and large queries on LargeRDFBench. We omit the simple queries since they exhibit the same behavior. The high communication overhead affected the runtime of all systems. For complex queries, FedX timed out on two queries and gave runtime errors in two others. HiBISCuS timed out on three queries but did reasonably well in the rest. SPLENDID was able evaluate only five out of the ten complex queries. Lusail outperformed all other systems in almost all complex queries, in some cases by up to two orders of magnitude (*C1* and *C9*). Large queries show the same behavior. Lusail is the only system that returns results (no time out or runtime errors).

Figure 14(c) shows results on two endpoints of the LUBM dataset. Lusail's query response times increased slightly compared to the local cluster (Figure 12(a)). All queries are finished in around 1 second. In contrast, both FedX and HiBISCuS require more than 1,000 seconds; an order of magnitude compared to their performance on the local cluster. This shows their sensitivity to the communication overhead since they tend to communicate large volumes of data. With four endpoints, FedX and HiBISCuS were able evaluate only Q3 and ran out of memory or timed out in the rest.

Real Endpoints: In this experiment, we use Lusail and FedX to query Bio2RDF[9] endpoints. We extracted three representative queries from a real workload: R1, R2 and R3[10], from the Bio2RDF query log. R1 accesses three endpoints; DrugBank, HGNC and MGI. R2 joins data from PharmGKB and OMIM while R3 integrates data from DrugBank and OMIM. This experiment uses a single machine of our *84-cores* cluster as a mediator. FedX was not able to evaluate these queries as it throws several runtime exceptions. On the other hand, Lusail successfully evaluated queries R1, R2 and R3 in 12, 8 and 35 seconds, respectively. This demonstrates that Lusail is capable of solving queries accessing real, independently deployed, endpoints and provides good performance.

VII. Related Work

Distributed (i.e., parallel) RDF systems [16], [18], [19], [20], [21] deal with data stored in a single endpoint, where the data is replicated and/or partitioned among different servers in the same cluster. The goal of these systems is to speed up query execution for RDF data at one endpoint. In contrast, federated RDF systems have no control over the data; the data is accessible through independent, remote SPARQL endpoints.

Federated SPARQL systems can be classified into index-based and index-free. The source selection in index-based systems, such as ANAPSID [22], SPLENDID [7], DARQ [23], and HiBISCuS [8], is based on collected information and statistics about the data hosted by each endpoint. Therefore, the cost of adding a new endpoint is proportional to the size of the data. Index-free systems, such as Fedx [6] and Lusail, do not assume any prior knowledge of the datasets. Fedx [6] and Lusail utilize SPARQL ASK queries to find the relevant endpoints and cache their results for future usage. Thus, the startup cost and the cost of adding a new endpoint is small.

Federated SPARQL systems usually divides the query into exclusive groups of triple patterns, where each group has a solution at only one endpoint. They do not check whether the data instances matching a group of triples are located in the same endpoint. Thus, they fail to create groups for triple patterns having solutions in different endpoints. Lusail detects whether the data instances are located in disjoint groups, i.e., each endpoint can solve the subquery locally, or distributed

[9]http://bio2rdf.org/

[10]The queries are shown in [11]

groups, i.e., some instances are located in remote endpoints. This allows us shift most of the computation on intermediate data to the endpoints.

Several efforts, such as Ariadne [24], InfoMaster [25], Garlic [26], and Disco [27], have focused on web-based data integration over heterogeneous information sources [28]. In general, a wrapper is run at each data source to translate between the supported languages and data models. Moreover, systems, such as Piazza [29], coDB [30] and HePToX [31], are peer-to-peer systems that interconnect a network of heterogeneous data sources. Since Lusail works with SPARQL endpoints, it does not need wrappers, and it takes advantage of the capabilities of SPARQL (e.g., ASK). Moreover, while these systems utilize source descriptions (schema), Lusail does not assume any prior knowledge about the datasets. In terms of query decomposition, these systems also aim at dividing a query into exclusive subqueries based on the known schema, where each subquery is submitted to only one data source. In contrast, Lusail's decomposition benefits from the actual location of data matching the query to maximize the local computation and increase parallelism.

VIII. Conclusion

Lusail optimizes federated SPARQL query processing through a locality-aware decomposition at compile time followed by selectivity-aware and parallel query execution at run time. Lusail's decomposition is based not on the schema but rather on the actual location of data instances satisfying the query triple patterns. This decomposition increases parallelism in query execution and minimizes the retrieval of unnecessary data from the endpoints. Selectivity-aware and parallel query execution orders queries at run time by delaying subqueries expected to return large results, and chooses join orders of the results of subqueries that achieve a high degree of parallelism. To the best of our knowledge, our locality- and selectivity-aware optimizations are the first to be carried out on processing federated queries over decentralized RDF graphs. Hence, Lusail outperforms state-of-the-art systems by orders of magnitude and scale to more than 250 endpoints with data sizes up to billions of triples.

References

[1] M. Schmachtenberg, C. Bizer, and H. Paulheim, "Adoption of the linked data best practices in different topical domains," in *Proc. of International Semantic Web Conference (ISWC)*, 2014.

[2] M. I. Ali, N. Ono, M. Kaysar, K. Griffin, and A. Mileo, "A semantic processing framework for IoT-enabled communication systems," in *Proc. of International Semantic Web Conference (ISWC)*, 2015.

[3] E. Mansour, A. V. Sambra, S. Hawke, M. Zereba, S. Capadisli, A. Ghanem, A. Aboulnaga, and T. Berners-Lee, "A demonstration of the solid platform for social web applications," in *Proc. World Wide Web Conf. (WWW)*, 2016.

[4] S. Tramp, P. Frischmuth, T. Ermilov, S. Shekarpour, and S. Auer, "An architecture of a distributed semantic social network," *Semantic Web*, vol. 5, no. 1, 2014.

[5] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semantics*, vol. 3, no. 2-3, 2005.

[6] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "FedX: Optimization techniques for federated query processing on linked data," in *Proc. Int. Semantic Web Conf. (ISWC)*, 2011.

[7] O. Görlitz and S. Staab, "SPLENDID: SPARQL endpoint federation exploiting VOID descriptions," in *Proc. Workshop on Consuming Linked Data (COLD) at (ISWC)*, 2011.

[8] M. Saleem and A. N. Ngomo, "HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation," in *Proc. Extended Semantic Web Conf. (ESWC)*, 2014.

[9] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo, "A fine-grained evaluation of SPARQL endpoint federation systems," *Semantic Web Journal*, vol. 7, no. 5, 2015.

[10] N. A. Rakhmawati, M. Saleem, S. Lalithsena, and S. Decker, "QFed: Query set for federated SPARQL query benchmark," in *Proc. Int. Conf. on Information Integration and Web-based Applications (iiWAS))*, 2014.

[11] [Online]. Available: http://ds.qcri.org/publications/Lusail_extraFeatures.pdf

[12] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world SPARQL queries," in *1st International Workshop on Usage Analysis and the Web of Data (USEWOD) at Proc. World Wide Web Conf. (WWW)*, 2011.

[13] L. Bol'shev and M. Ubaidullaeva, "Chauvenet's test in the classical theory of errors," *Theory of Probability & Its Applications*, vol. 19, no. 4, 1975.

[14] O. Erling and I. Mikhailov, "Rdf support in the virtuoso dbms," in *Networked Knowledge-Networked Media*. Springer, 2009.

[15] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, no. 1, 2010.

[16] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning," *The VLDB Journal*, 2016.

[17] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[18] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald, "TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing," in *Proc. Int. Conf. on Management of Data (SIGMOD)*, 2014.

[19] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," *PVLDB*, vol. 6, no. 4, 2013.

[20] K. Lee and L. Liu, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *PVLDB*, vol. 6, no. 14, 2013.

[21] J. Huang, D. Abadi, and K. Ren, "Scalable SPARQL Querying of Large RDF Graphs," *PVLDB*, vol. 4, no. 11, 2011.

[22] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus, "ANAPSID: an adaptive query processing engine for SPARQL endpoints," in *Proc. Int. Semantic Web Conf. (ISWC)*, 2011.

[23] B. Quilitz and U. Leser, "Querying distributed RDF data sources with SPARQL," in *Proc. European Semantic Web Conf. on The Semantic Web: Research and Applications (ESWC)*, 2008.

[24] J. L. Ambite and C. A. Knoblock, "Flexible and scalable query planning in distributed and heterogeneous environments." in *Proc. Int. Conf. on Artificial Intelligence Planning Systems (AIPS)*, 1998.

[25] O. M. Duschka and M. R. Genesereth, "Query planning in infomaster," in *Proc. ACM Symposium on Applied Computing (SAC)*, 1997.

[26] M. T. Roth and P. M. Schwarz, "Don't scrap it, wrap it! a wrapper architecture for legacy data sources." in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, vol. 97, 1997.

[27] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling heterogeneous databases and the design of disco," in *Proc. Int. Conf. on Distributed Computing Systems (ICDCS)*, 1996.

[28] M. Ouzzani and A. Bouguettaya, "Query processing and optimization on the web," *Distributed and Parallel Databases*, vol. 15, no. 3, 2004.

[29] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov, "Piazza: Data management infrastructure for semantic web applications," in *Proc. Int. Conf. on World Wide Web*, ser. WWW, 2003.

[30] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu, "Queries and updates in the codb peer to peer database system," in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

[31] A. Bonifati, E. Chang, T. Ho, L. V. Lakshmanan, R. Pottinger, and Y. Chung, "Schema mapping and query translation in heterogeneous p2p xml databases," *The VLDB Journal*, vol. 19, no. 2, 2010.