Lusail: A System for Querying Linked Data at Scale

Ibrahim Abdelaziz[*] Essam Mansour[◇] Mourad Ouzzani[◇] Ashraf Aboulnaga[◇] Panos Kalnis[*]
[*]King Abdullah University of Science & Technology [◇]Qatar Computing Research Institute, HBKU
{first}.{last}@kaust.edu.sa {emansour,mouzzani,aaboulnaga}@hbku.edu.qa

ABSTRACT

The RDF data model allows publishing interlinked RDF datasets, where each dataset is independently maintained and is queryable via a SPARQL endpoint. Many applications would benefit from querying the resulting large, decentralized, geo-distributed graph through a federated SPARQL query processor. A crucial factor for good performance in federated query processing is pushing as much computation as possible to the local endpoints. Surprisingly, existing federated SPARQL engines are not effective at this task since they rely only on schema information. Consequently, they cause unnecessary data retrieval and communication, leading to poor scalability and response time. This paper addresses these limitations and presents *Lusail*, a scalable and efficient federated SPARQL system for querying large RDF graphs that are geo-distributed on different endpoints. Lusail uses a novel query rewriting algorithm to push computation to the local endpoints by relying on information about the RDF instances and not only the schema. The query rewriting algorithm has the additional advantage of exposing parallelism in query processing, which Lusail exploits through advanced scheduling at query run time. Our experiments on billions of triples of real and synthetic data show that Lusail outperforms state-of-the-art systems by orders of magnitude in terms of scalability and response time.

PVLDB Reference Format:
Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. Lusail: A System for Querying Linked Data at Scale. *PVLDB*, 11(4): xxxx-yyyy, 2018.
DOI: 10.1145/3164135.3164144

1. INTRODUCTION

The Resource Description Framework (RDF) is extensively used to represent structured data on the Web. RDF uses a simple graph model in which data is represented in the form of ⟨`subject`, predicate, `object`⟩ triples. A key feature is the ability to link two entities from two different

Proceedings of the VLDB Endowment, Vol. 11, No. 4
Copyright 2017 VLDB Endowment 2150-8097/17/12... $ 10.00.
DOI: 10.1145/3164135.3164144

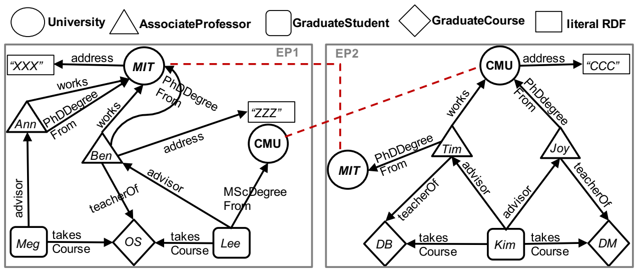


Figure 1: A decentralized graph for two universities managed by independent geo-distributed SPARQL endpoints (EP). The red dotted line represents an interlink between endpoints, i.e., a vertex in an endpoint referring to another vertex in another endpoint. Thus, to get the address of the university from which Tim got his PhD, the interlink from EP2 to EP1 must 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 . }
```

Figure 2: A SPARQL query, Q_a, over a decentralized RDF graph across different universities. This query has to traverse the interlink between EP2 and EP1.

RDF datasets which are maintained by two independent authorities, as shown in Figure 1. Through such links, *large decentralized graphs* are created among a large number of geo-distributed RDF stores where each RDF store can be queried through its own SPARQL endpoint. The Linked Open Data Cloud is one such decentralized RDF graph; it has more than 150 billion triples in around 10,000 datasets[1] from different domains, such as media, government, and life sciences [32].

Users can retrieve data from an individual dataset by issuing SPARQL queries against its SPARQL endpoint. However, it is often very useful to issue SPARQL queries that integrate data from multiple RDF datasets, which would require *federated query processing*. For example, Figure 2 shows a query (Q_a) on data from the LUBM benchmark [14] 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 answers: (Kim, Joy, CMU, "CCCC"), (Kim, Tim, MIT, "XXX"), and (Lee, Ben, MIT, "XXX"). One cannot simply evaluate

[1]`http://stats.lod2.eu/`

Q_a independently at each endpoint and concatenate their results as this will miss the results about Tim since EP2 does not have the address of MIT. Instead, we need a federated query processor that can automatically identify the endpoints that can answer each triple pattern, detect interlinks between endpoints, and automatically traverse them to compute the query answer. A federated query processor would decompose Q_a into subqueries, send each subquery to the relevant endpoint, and compute the answer to Q_a from the results of the subqueries. Computing such an answer typically requires the federated query processor to *join* the results obtained from the endpoints. For example, in Q_a this join would combine the address of MIT from EP1 with the information about Tim from EP2.

Conceptually, a query like Q_a can be processed by sending each of its triple patterns to all the endpoints, retrieving all matching triples from the endpoints, and joining all of these triples at the federated query processor to compute the query answer. This strategy is clearly inefficient since it sends triple patterns to endpoints even if they have no answers for them, retrieves triples that may not be relevant, and joins triples at the federated query processor even if they could be joined at the endpoints. Thus, this strategy would result in an unnecessarily large number of requests to the endpoints and unnecessarily large amounts of data retrieved from the endpoints and transferred over the network to the federated query processor. To avoid these unnecessary overheads, it is important for a federated query processor to push as much processing as possible to the endpoints.

Existing SPARQL federated query processing systems rely on schema information to push processing to the endpoints. For example, they use SPARQL ASK queries to check whether or not a triple pattern has an answer at an endpoint [34]. If a group of triple patterns can be answered exclusively by one endpoint, then it is possible to send this group to the endpoint as one unit, known as an *exclusive group*. Relying solely on schema information is not effective since RDF sources often utilize similar ontologies (e.g., EP1 and EP2 in Figure 2 have the same predicates), thus a triple pattern could be answerable by multiple endpoints and therefore cannot be part of an exclusive group. In this case, the triple pattern is sent to all the endpoints that can answer it and the values in the retrieved triples are bound to other triple patterns; the triple patterns with bound values are sent to the endpoints to retrieve further triples. This is known as a *bound join* operation, and effectively amounts to the query being processed one triple pattern at a time.

This strategy retrieves unnecessary data from the endpoints, since it retrieves all data matching a triple pattern even if this data is not useful for the rest of the query. Moreover, this process limits the available parallelism since only one join step can be processed at a time, and the federated query processor has to wait for the results of this join step before issuing the next join. To quantify the inefficiency of this approach, we note that our experiments on FedX [34], a federated SPARQL system that uses this approach and that was shown to outperform similar systems [30], show that increasing the number of endpoints from 1 to 4 can lead to 6 orders of magnitude increase in the number of requests sent to the endpoints, and 3 orders of magnitude increase in the running time (see [3] for details).

This paper addresses the limited ability of existing systems to push query processing to the local endpoints. We present Lusail, a scalable and efficient system for federated SPARQL query processing over decentralized RDF graphs. Lusail is the first system to decompose the federated query based on *instance* information not just schema information. That is, Lusail decomposes the query based on knowledge of the locations of the actual RDF triples matching triple patterns in the query. This knowledge helps us identify, for example, that the instances matching the variable `?S` in ⟨`?S`, ub:advisor, `?P`⟩ and ⟨`?S`, ub:takesCourse, `?C`⟩ in Q_a are always located in the same endpoint, so these triple patterns can be joined locally at the endpoint even though schema information tells us that both endpoints can answer both triple patterns. In contrast, the instances matching the variable `?U` in ⟨`?P`, ub:PhDDegreeFrom, `?U`⟩ and ⟨`?U`, ub:address, `?A`⟩ are sometimes located in different endpoints, so these triple patterns cannot be joined locally.

Lusail processes queries in a two-phase strategy: (*i*) Locality-Aware DEcomposition (LADE) of the query into subqueries to maximize the computation at the endpoints and minimize intermediate results, and (*ii*) Selectivity-Aware and Parallel Execution (SAPE) to reduce network latency and increase parallelism. Unlike prior approaches, the decomposition of LADE is based not only on schema information but also on instance information, i.e., the location of triples satisfying the triple patterns in the query. SAPE decides the order of executing the subqueries generated by LADE based on their result sizes and degree of parallelism.

We demonstrated Lusail in [20] and discussed the challenges of processing federated SPARQL queries at scale in a short paper [4]. In this paper, we describe the complete system. Our main contributions are:

- A locality-aware decomposition method that dramatically reduces the number of remote requests and allows for better utilization of the endpoints. We also provide a proof of correctness. (Section 3)
- A cost model that uses lightweight runtime statistics to decide the order of submitting subqueries and the 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 4)
- Our experiments on real data and synthetic benchmarks with billions of triples show that Lusail outperforms state-of-the-art systems by up to three orders of magnitude and scales up to 256 endpoints compared to 4 endpoints in existing systems. (Section 5)

We present the architecture of Lusail in Section 2, discuss related work in Section 6, and conclude in Section 7.

2. THE LUSAIL ARCHITECTURE

The Lusail architecture is shown in Figure 3. Lusail analyzes each query to identify the relevant endpoints and its correct decomposition that achieves high parallelism and minimal communication cost. After that, Lusail sends the subqueries to the relevant endpoints, joins their results, and sends the query answer back to the user.

Locality-Aware Decomposition (LADE): Query decomposition starts by identifying the relevant endpoints (source selection). Like similar systems [34, 31], we use a set of SPARQL ASK queries, one for each triple pattern.

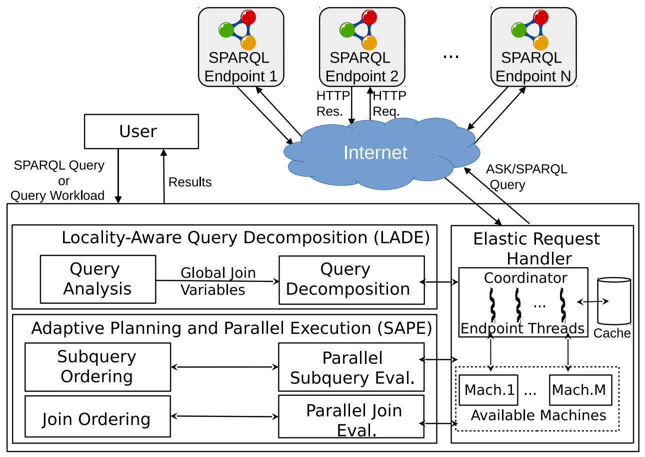


Figure 3: The Lusail system architecture.

Furthermore, 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. To do so, LADE utilizes the knowledge of the locations of the actual RDF triple instances matching a query variable. The result of this check determines a group of triple patterns, i.e., a subquery, that can be sent together to an endpoint. Based on this analysis, LADE decomposes the query into a set of independent subqueries. Lusail caches the results of both the source selection phase and the check queries that determine the triple patterns which *cannot* be executed locally at an endpoint.

Selectivity-Aware Planning and Parallel Execution (SAPE): SAPE takes as input the set of subqueries produced by LADE and schedules them for execution. This set of independent subqueries can be submitted concurrently for execution at each of the relevant endpoints, and Lusail can use one thread per endpoint to collect their results. SAPE uses cardinality estimates for the different triple patterns to delay subqueries that are expected to return large results. The results of these subqueries will then need to be joined by SAPE using a parallel join, where the join order is determined based on the actual sizes of the subquery results. SAPE achieves a high degree of parallelism while minimizing the communication cost by (i) obtaining results from different endpoints simultaneously, and (ii) utilizing different threads in joining the results.

Elastic Request Handler (ERH): Lusail utilizes multiple threads for evaluating the ASK queries from LADE or the subqueries from SAPE at the endpoints. ERH manages the allocation of threads from one or more machines to these tasks, where the number of available threads is determined by the number of physical cores.

3. LOCALITY-AWARE DECOMPOSITION

To push as much processing as possible to the endpoints, LADE maximizes the number of triple patterns in a given query that can be sent together to each endpoint. In a decentralized RDF graph, data instances matching a pair of triples may not be located in the same endpoint, e.g., the triples having `?U` as a common variable in Figure 1. Thus, putting this pair in the same subquery may miss results. LADE starts by analyzing which triples cannot be in the same subquery, and identifying the common variables

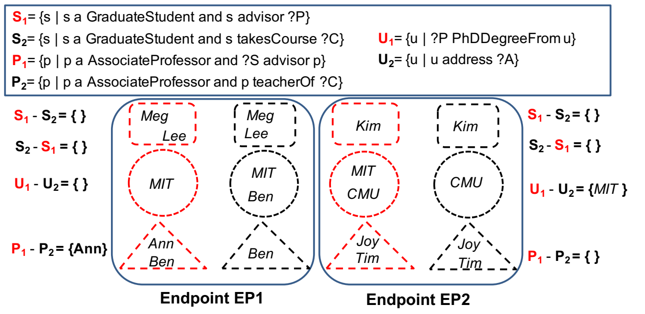

S1= {s | s a GraduateStudent and s advisor ?P}
S2= {s | s a GraduateStudent and s takesCourse ?C}
P1= {p | p a AssociateProfessor and ?S advisor p}
P2= {p | p a AssociateProfessor and p teacherOf ?C}
U1= {u | ?P PhDDegreeFrom u}
U2= {u | u address ?A}
S1 - S2 = { }
S2 - S1 = { }
U1 - U2 = { }
P1 - P2 = {Ann}
Meg Lee
Meg Lee
MIT
MIT Ben
Ann Ben
Ben
Endpoint EP1
Kim
Kim
MIT CMU
CMU
Joy Tim
Joy Tim
Endpoint EP2
S1 - S2 = { }
S2 - S1 = { }
U1 - U2 = {MIT }
P1 - P2 = { }

Figure 4: Locality analysis of data instances in EP1 and EP2 from Figure 1 that match `?S`, `?U`, and `?P` in a pair of triple patterns in Q_a.

in these triples as *global join variables (GJV)*. Then, it decomposes the conjunction of triple patterns into subqueries. 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.

In this section, we only discuss how Lusail evaluates conjunctive SPARQL queries. However, Lusail also supports queries with joins on variable predicates as well as UNION, FILTER, LIMIT, and OPTIONAL statements (see [3] for more details). For example, Lusail determines where to add the FILTER and OPTIONAL clauses during query decomposition and during the global join evaluation.

3.1 Detecting Global Join Variables

A global join variable (v) is a variable that appears in at least two different triple patterns such that these triple patterns, when taken together, cannot be solved by a single endpoint. A global join between data coming from 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 (Figure 2) has four variables appearing in more than one triple pattern, namely `?S`, `?U`, `?P`, and `?C`. Figure 4 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 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 detects 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 (see Algorithm 1). 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`

```
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
```

Figure 5: 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.

`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 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 4, 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 4.

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 one or more 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 triple patterns where `?U` is common (Figure 4). This allows us to have simple plans and better parallel execution.

Set difference ($-$) is implemented using *FILTER NOT EXISTS* (Figure 5) 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*.

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 4, 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 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 above. For the *object only* and *subject only* 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).

Algorithm 1: Detecting Global Join Variables

```
Input: Input query ($Q$), Set of relevant sources ($Sources$)
Result: List of Global Join Variables ($V$)
1  $Triples \leftarrow Q$.getTriplePatterns();
2  $vars \leftarrow$ getJoinEntities ($Triples$);
3  $chkQueries \leftarrow \emptyset$;
4  $V \leftarrow \emptyset$;
5  foreach $var_i$ in $vars$ do
6      $pairWiseTriples \leftarrow$getPairTriples ($var_i.Triples$);
7      $joinVar \leftarrow$ False;
8      foreach $pair_i$ in $pairWiseTriples$ do
9          if $pair_i[0].sources \neq pair_i[1].sources$ then
10             $V$.addJoinVar ($var_i$.varName, $pair_i$);
11             $joinVar \leftarrow$ True;
12     if $joinVar$ is True then continue ;
13     if $var_i$ is subject only || $var_i$ is object only then
14         $chkQueries \leftarrow$ formulatePairWiseQuery ($var_i$, $pairWiseTriples$);
15     if $var_i$ is subject and object then
16         $chkQueries \leftarrow$ formulateSubjObjQuery ($var_i$, $var_i$.subjTriples, $var_i$.objTriples);
17 if $chkQueries$ is not empty then
18     $ReqHandler \leftarrow$ 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 $V$;
```

The algorithm uses the elastic request handler (Figure 3) 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.

Let $|V|$ be the number of variables appearing in more than one triple pattern in the query and $|T|$ be the number of triples. Since check queries are formed for pairs of triples, the maximum number of check queries, C_Q, is bound by $O(|V| * |T|^2)$. Assuming N relevant endpoints, LADE creates a maximum of $N * C_Q$ requests. Since the number of triple patterns in real-world SPARQL queries is usually small [12], the number of GJVs is also small. Therefore, $N * C_Q$ will be typically small. In addition, the check queries are lightweight and have minimal overhead (see Section 5.4).

3.2 Query Decomposition

Algorithm 2 decomposes a query Q into multiple subqueries to be sent to different endpoints. If Q has no GJVs, the algorithm returns Q (line 3). Otherwise, LADE uses the set of GJVs and the source selection information to decompose Q. It iterates over all join variables in any order using the current join variable as a root. It tries to find the best decomposition that leads to a set of subqueries with minimal execution cost (cost estimation is discussed in Section 4). The algorithm has two phases: branching (lines 9-30) and merging (lines 32).

In the branching phase, we build a query tree with the current join variable as its root (line 9). An initial set of subqueries is created at the root, one subquery per child

Algorithm 2: Query Decomposition

```
   Input: Input query ($Q$), set of GJVs ($V$)
   Result: Set of independent subqueries
1  $bestDecomposition \leftarrow \emptyset$;
2  $minDecompCost \leftarrow infinity$;
3  if $V$ is empty then return $subqueries$.add ($Q$);
4  $Triples \leftarrow Q$.getTriplePatterns();
5  foreach $jvar_i$ in $V$ do
6      $visitedTriples \leftarrow \emptyset$;
7      $nodes \leftarrow \emptyset$;
8      $subqueries \leftarrow \emptyset$;
9      $nodes$.push ($jvar_i$);
10     while $nodes$ is not empty do
11         $vrtx \leftarrow nodes$.pop ();
12         $edges \leftarrow vrtx$.edges ();
13         if $subqueries$ is empty then
14             foreach $edge_i$ in $edges$ do
15                 if visited ($edge_i$, $visitedTriples$) then
                   continue ;
16                 $sq \leftarrow$ createSubquery($edge_i$);
17                 $subqueries$.add ($sq$);
18                 $nodes$.push ($edge_i$.destNode);
19                 $visitedTriples$.add ($edge_i$);
20             continue;
21         $parentSq \leftarrow$ getParentSubquery ($vrtx$, $subqueries$);
22         foreach $edge_i$ in $edges$ do
23             if visited ($edge_i$, $visitedTriples$) then continue ;
24             if canBeAddedToSubQ ($parentSq$, $edge_i$, $V$) then
25                 $parentSq \leftarrow$ addToSubquery ($parentSq$, $edge_i$);
26             else
27                 $sq \leftarrow$ createSubquery($edge_i$);
28                 $subqueries$.add ($sq$);
29             $nodes$.push ($edge_i$.destNode);
30             $visitedTriples$.add ($edge_i$);
31     if $visitedTriples \equiv Triples$ then
32         $subqueries \leftarrow$ mergeSubQ ($subqueries$);
33         $cost \leftarrow$ estimateCost ($subqueries$);
34         if $cost < minDecompCost$ then
35             $bestDecomposition \leftarrow subqueries$;
36             $minDecompCost \leftarrow cost$;
37 return $bestDecomposition$;
```

(lines 13-20) and each subquery is expanded through depth first traversal (lines 21-30). A triple pattern is added (lines 24-25) if both the subquery and the triple pattern have the same relevant sources, and the addition of the pattern does 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 27-28). In both cases, the edge destination node is added to the nodes stack and marked as visited (lines 29-30).

The merging phase (line 32) starts once all triple patterns are assigned to one of the subqueries (line 31). The function *mergeSubQ* (line 32) loops through the set of subqueries and merges a pair of subqueries if they have common variables, the same relevant sources, and no pair of triple patterns from both subqueries has a common variable that is global. If the estimated cost (line 33) of the current decomposition is less than other decompositions, the algorithm updates the minimum cost and selects the current decomposition as the current best. The algorithm continues to check other possible decompositions using the remaining join variables.

The algorithm returns the best subquery decomposition (line 37). For simplicity, the pseudo-code of the algorithm assumes a connected query graph. Lusail also supports queries with multiple disconnected subgraphs, in which case it executes each subquery independently and creates a special join variable that connects these subqueries, if possible.

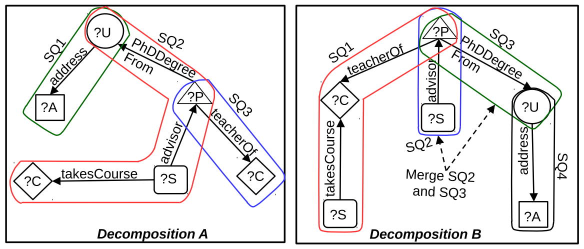


Figure 6: Two possible decompositions of Q_a, where the GJVs are `?U` and `?P`. Any pair of predicates, which causes a variable to be a GJV, cannot be in the same subquery.

Figure 6 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 5 in Algorithm 2). However, all decompositions produce the same result set and do not miss any triple (details in Section 3.3), but some decompositions may generate more intermediate results and thus cost more. To avoid a costly decomposition, LADE enumerates all possible decompositions and chooses at compile time the best decomposition expected to minimize the intermediate results.

The outer loop (line 5 in Algorithm 2) iterates over the set of GJVs to generate all possible query decompositions. Each iteration performs a depth first traversal, whose complexity is $O(|V|+|T|)$ where $|V|$ is the total number of query variables and $|T|$ is the number of triple patterns. Since the number of iterations is the number of GJVs, which is small, the algorithm complexity is still bound by $O(|V|+|T|)$.

3.3 Result Completeness

Missing Results. The optimization introduced by LADE assigns triple patterns to different subqueries based on the concept of locality. Results could be missed in two cases. **Case 1**: A subquery contains a set of triple patterns where a GJV is considered to be local. 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 cause Q_a to miss the result (Kim, Tim, MIT, "XXX") when the subquery is submitted to EP2. However, such a case cannot happen since LADE puts triple patterns into the same subquery only if the data instances matching them are located in the same endpoint. Lemma 1 formalizes this argument. **Case 2**: A subject or object may be present in more than one endpoint, e.g., EP1 has ⟨`a1`, p, `b`⟩, ⟨`b`, q, `c1`⟩ and EP2 has ⟨`a2`, p, `b`⟩, ⟨`b`, q, `c2`⟩. Having the pair of triple patterns, ⟨`?x`, p, `?y`⟩ and ⟨`?y`, q, `?z`⟩, in the same subquery does not miss the local triples matching the query. Lusail first detects $?y$ as a local join variable and then performs the join between the results of the same subquery from different endpoints at the Lusail server (see Section 4.2).

LEMMA 1. *Any local join variable detected by LADE is a true local join variable.*

PROOF: Let v be the join variable and $TP(v) = \{tp_1, tp_2, ...tp_k\}$ be the set of triple patterns in which v appears. v can appear in $TP(v)$ as subject only, object only, or subject and object.

Subject only: In this case, $\forall_{tp_i \in TP(v)}\ tp_i.subj = v$. Let B_i and B_j be the set of bindings of v from triples tp_i and tp_j, respectively. LADE decides that v is a local join variable iff: $\forall_{0<l<t}\ \forall_{0<i,j<k, i \neq j}\ B_i(ep_l) - B_j(ep_l) = \phi$ and $B_j(ep_l) - B_i(ep_l) = \phi$ where k=$|TP(v)|$ and t is the number of relevant endpoints. At each relevant endpoint, $B_i - B_j = \phi$ means that each endpoint can fully evaluate $tp_i \bowtie tp_j$ locally. This means that v is a true local join variable and there is no need to join tp_i and tp_j across endpoints. The same applies for $B_j - B_i$.
Object only: In this case, $\forall_{tp_i \in TP(v)}\ tp_i.obj = v$. The same analysis of the subject only case applies.
Subject/Object: Let $TPS(v) = \{tps_1, ...tps_s\}$ and $TPO = \{tpo_1, ...tpo_o\}$ be the set of triples in which v appears as subject and object, respectively. $\forall_{tps_i \in TPS(v)}\ tps_i.subj = v$ and $\forall_{tpo_i \in TPO(v)}\ tpo_i.obj = v$. Let B_i and B_j be the set of bindings of v using triple tps_i and tpo_j, respectively. LADE decides that v is a local join variable iff: $\forall_{0<l<t}\ \forall_{0<i<s,\ 0<j<o}\ B_i(ep_l) - B_j(ep_l) = \phi$. At each relevant endpoint, $B_i - B_j = \phi$ means that each endpoint can fully evaluate $tps_i \bowtie tpo_j$ locally. It also means that v is a true local join variable and there is no need to join tps_i and tpo_j across endpoints. □

Extraneous computations. 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. For example, the variable `?P` in ⟨`?S`, ub:advisor, `?P`⟩ and ⟨`?P`, ub:teacherOf, `?C`⟩. As shown in EP1 (Figure 4), there is an advisor (Ann) who works at MIT but who is not a teacher of any course. `?P` will be considered as a GJV based on our checks. However, it is clearly safe to send both triple patterns in the same subquery since there is no need to access data in remote endpoints. Adding more checks to avoid such cases would be too expensive since it would require accessing all other relevant endpoints. Such cases may lead to query plans with unnecessary GJVs, i.e., more remote requests and more join computations at global level rather than at the endpoints. Lemma 2 shows that assuming that a join variable is global, while it is not, does not affect the correctness of the results.

LEMMA 2. *Any local join variable v can be selected as a global join variable without affecting the result correctness.*

PROOF: Let $TP(v) = \{tp_1, ..., tp_k\}$ be the set of triple patterns in which v appears. If v is a local join variable, each relevant endpoint can evaluate $TP(v)$ as a single subquery. The set of bindings of the local join variable v is simply the union of all bindings from all relevant endpoints, i.e. $B_l(v, TP(v)) = \cup_{0<i<t} B_l(v, TP(v), ep_i)$ where t is the number of relevant endpoints. Assume now that v is considered a global join variable. In this case, each endpoint will evaluate each triple pattern independently and the results are joined at the global level. Let $B_g(v, tp_j) = \cup_{0<i<t} B_g(v, tp_j, ep_i)$ be the set of bindings of the global variable v for triple pattern tp_j. Then, the global bindings of v is $B_g(v, TP(v)) = B_g(v, tp_1) \bowtie B_g(v, tp_2) ... \bowtie B_g(v, tp_k)$. Since v should be a local join variable, then the join between different endpoints is always empty. This means that $B_g(v, TP(v)) = \cup_{0<i<t} B_g(v, tp_1, ep_i) \bowtie B_g(v, tp_2, ep_i) ... \bowtie B_g(v, tp_k, ep_i)$ which is equivalent to evaluating all triples in $TP(v)$ as a single subquery and taking the union across the relevant endpoints. Consequently, $B_g(v, TP(v)) = B_l(v, TP(v))$. □

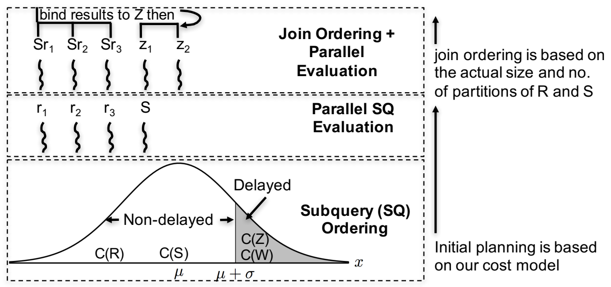

bind results to Z then
Sr1
Sr2
Sr3
z1
z2
Join Ordering + Parallel Evaluation
join ordering is based on the actual size and no. of partitions of R and S
r1
r2
r3
S
Parallel SQ Evaluation
Delayed
Non-delayed
Subquery (SQ) Ordering
C(R)
C(S)
C(Z)
C(W)
x
μ
μ + σ
Initial planning is based on our cost model

Figure 7: Query evaluation in Lusail

4. SELECTIVITY-AWARE EXECUTION

The Selectivity-Aware Planning and parallel Execution (SAPE) algorithm 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. An overview of SAPE is shown in Figure 7. 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 of SAPE is to maximize the degree of parallelism while minimizing the communication cost in terms of the number of requests to endpoints and size of subquery results.

4.1 Subquery Ordering and Cost Model

LADE outputs a set of independent subqueries that can be submitted concurrently for execution at each of the 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 join. For example, the subqueries of Figure 6 would be executed concurrently and after receiving all their results, a join phase would start. Notice that the subquery ⟨`?U`, address, `?A`⟩ is so generic that executing it independently will retrieve all entities with addresses regardless of whether these entities match `?U` in the remaining subqueries (see Figure 1). These subqueries, which touch most of the endpoints or retrieve large amounts of intermediate results, affect 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 RDF 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.

At the other extreme, we can submit the most selective subquery first and use the actual bindings of the variables obtained to submit the next most selective subquery with its variables bound to the values retrieved by the first subquery, and then submit the next most selective subquery in a similar fashion, and so on. While limiting the amount of intermediate results to be retrieved from the endpoints, this approach offers no parallelism beyond submitting the same subquery to multiple endpoints.

Our objective is to balance between the degree of parallelism, i.e., the number of subqueries submitted concurrently, and the communication cost, which is dominated by the size of intermediate results. Our only constraint is that Lusail should avoid collecting expensive statistics during pre-processing or at runtime. Therefore, Lusail uses only lightweight per-triple statistics during query evaluation. To fulfill our objective, we detect the subqueries expected to return substantially fewer results if some of their variables are bound to the results already obtained. 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. To this end, we introduce the concept of *delayed subqueries*, which are 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 to 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, so most subqueries return results whose sizes are within one standard deviation of the mean. SAPE calculates the mean μ and standard deviation σ values for all the cardinalities and all the numbers of relevant endpoints per subquery. Outliers, e.g., subqueries returning extremely large results (very low selectivity) or accessing a large number of endpoints compared to other subqueries, misleadingly increase the standard deviation. This may lead SAPE to consider some subqueries that are better to be delayed as non-delayed. We, therefore, apply the Chauvenet's criterion [7] 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 7. We apply the same concept for the number of relevant endpoints per subquery. With this heuristic, only subqueries (including outliers) whose results are expected to be significantly larger than the majority of subqueries will be delayed.

The cardinality of a subquery is estimated based on the cardinality of its triple patterns, which 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 estimates. Note that cardinality statistics per predicate are usually collected by RDF engines for their runtime query optimization [10, 23, 17] and it may be possible to use them to provide the required estimates. We leave this as future work.

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, denoted $C(sq_i, v)$, 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:

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

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 aggregateEndptResults ($ReqHandler$);
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 ($ReqHandler$);
```

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

While the proposed cost model is simple, it provides accurate cardinality estimates. To measure estimation accuracy, we compared the estimated vs. actual cardinality of subqueries with more than one triple pattern using the q-error metric [22]. Let a be the actual cardinality and e be an estimate of a. The q-error is defined as $\max(e/a, a/e)$. Using LargeRDFBench queries [29], the median q-error of Lusail in our experiments is 1.09, close to the optimal value of 1.

4.2 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 minimum 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.

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 until all subqueries are executed. When executing a subquery with its variables bound to values from the bindings hashmap, SAPE groups values from the hashmap into blocks and submits a subquery for each block (as opposed to a subquery for each value).

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, the algorithm evaluates the whole query at all relevant endpoints independently (line 3). Then, it aggregates the results obtained from relevant endpoints, joins the partial results from different endpoints, if necessary, and returns the final query answer (line 4).

If there is more than one subquery, 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. This reduces the number of found bindings used in delayed subqueries. In the second phase, SAPE evaluates the delayed subqueries using the found bindings from the first phase (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 ⟨`?s`, ?p, `?o`⟩, 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 subquery 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. 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 [21]. The DP algorithm starts with a join tree of size 1, i.e., a single relation, where the join cost is initially zero. It then builds larger join trees by considering the rest of the relations, pruning expensive partial plans as early as possible. At each DP step, SAPE joins the current subplan with another relation (R) leading to a new state S' with cost: $Cost(S') = min(Cost(S'), Cost(S) + JoinCost(S, R))$. Since the expanded state S' can be reached using different orders, we associate each state with the minimum cost found. 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 with the smaller relation build a hash table for their part of S. The threads that maintain R evaluate the join by probing these 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. (iv) Finally, SAPE aggregates the joined results from the individual threads and returns the result.

Table 1: 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

5. EXPERIMENTAL STUDY

5.1 Evaluation Setup

Compared Systems. We evaluate Lusail[2] against one index-free system, FedX [34], and two index-based systems, SPLENDID [13] and HiBISCuS [31]. [30] has shown that FedX outperformed other systems on the majority of queries and datasets. HiBISCuS [31] is an add-on to improve performance; we use it on top of FedX. SPLENDID showed competitive performance to FedX on several queries in [30] and LargeRDFBench[3]. 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 1Gbps Ethernet. The *480-cores* cluster is a Linux cluster of 20 machines, each with 24 cores and 148GB RAM, connected by 10Gbps 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 used several real and synthetic datasets. Table 1 shows their statistics. QFed [26] 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 [14] to generate data for 256 universities, each with around 138K triples. It includes links between the different universities through students and professors.

Queries. QFed [26] 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

[2]`https://github.com/Lusail/lusail`

[3]`https://github.com/AKSW/LargeRDFBench`

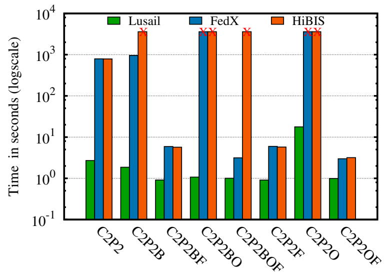
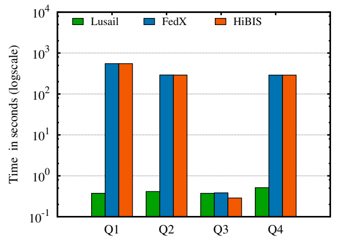
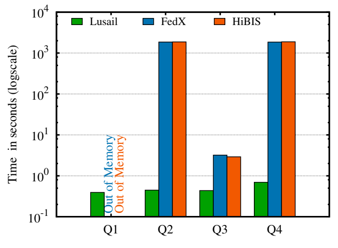


Figure 8: Queries with Filter have high selectivity while Big literal queries have bigger intermediate data.



(a) Two Endpoints



(b) Four Endpoints

Figure 9: 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.

benchmark queries. We only used the queries that access multiple endpoints. Queries *Q1*, *Q2*, and *Q3* in our experiments 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 (big) *B*. LargeRDFBench subsumes the FedBench benchmark [33]. The complex category contains 10 queries with a high number of triple patterns and advanced SPARQL clauses. The large category has 8 queries with large intermediate results.

Endpoints. We used Jena Fuseki 1.1.1 as the SPARQL engine at the endpoints for LUBM and QFed. Since Jena runs out of memory while indexing LargeRDFBench endpoints, we used a Virtuoso 7.1 instance for each of the 13 endpoints in LargeRDFBench. The standard, unmodified installation of each SPARQL engine was 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 present the results of our evaluation on a local cluster and on a geo-distributed settings, in Sections 5.2 and 5.3, respectively. We analyze the different costs of Lusail's query processing and its sensitivity to the threshold for delayed queries in Section 5.4.

In all subsequent experiments, all systems are allowed to cache the results of source selection. Each query is run three times and we report the average of the last two. We set a time limit of one hour per query before aborting.

5.2 Lusail on a Local Cluster

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

QFed Dataset. Figure 8 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 answered both queries in less than 2 seconds.

LUBM Dataset. This experiment utilizes up to four university datasets[4] from the LUBM benchmark, each in a different endpoint. Figures 9(a) and 9(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 number of remote requests. Lusail utilizes the schema as well as the location of data instances accessed by the query to formulate the subqueries. Thus, Lusail discovered that both *Q1* and *Q2* have only one subquery 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. FedX and HiBISCuS do not utilize such filtering so they sent the query to all endpoints. Lusail decomposed the query into two subqueries: the first subquery (students 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 decomposed *Q4* into two subqueries, with the second subquery delayed until the results of the first subquery are ready. The figures illustrate that Lusail is up 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 and took 52 seconds, significantly slower than all other systems, so it is not included in the figures.

LargeRDFBench Dataset. Figure 10 shows the response times of the different systems on the LargeRDFBench

[4]The other systems do not scale beyond four endpoints while Lusail scales to 256 endpoints (Figures 12(b) and 12(c)).

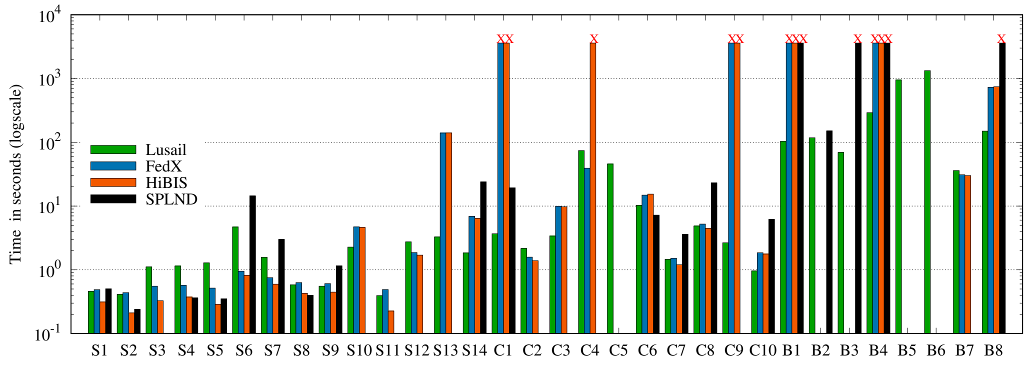
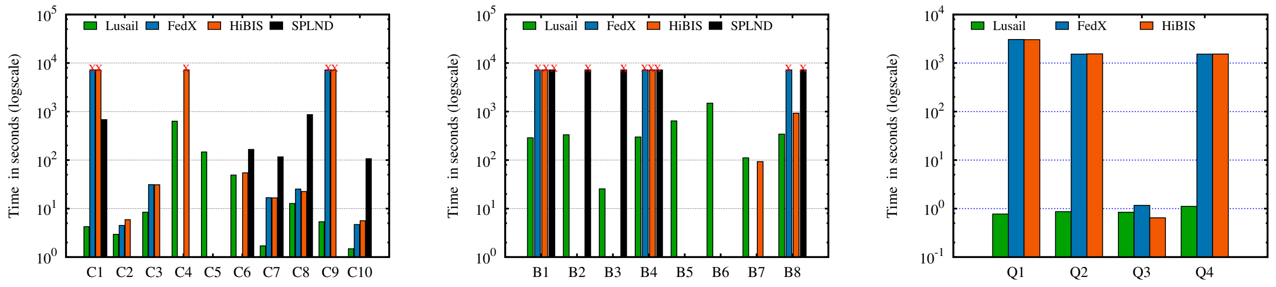


Figure 10: LargeRDFBench: Most of the simple queries do not access large intermediate data, unlike the complex and large queries. X corresponds to time out while missing bars correspond to runtime errors.



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

Figure 11: 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.

queries. The performance of Lusail and FedX is comparable for most of the simple queries. The preprocessing performed by the index-based systems, HiBISCuS and SPLENDID, sometimes results in better performance on the simple queries, but not always. For example, HiBISCuS is much slower than Lusail for *S13* and *S14*, and SPLENDID has the worst performance in *S6*, *S7*, *S9*, and *S14*. Lusail is the fastest system for *S13* and *S14* since these two queries return relatively large intermediate results. However, Lusail is generally not faster than the index-based systems on the simple queries since they do not generate large intermediate results and they access datasets with different schemas, so Lusail's optimizations do not improve performance.

The complex and large (or big) queries have a larger number of triple patterns per query, on average, and access a larger amount of intermediate data. Lusail achieves significantly better performance than other systems for most of the complex queries (Figure 10). *C5* contains two disjoint subgraphs joined by a filter variable, a query not supported by Lusail's competitors. Both FedX and HiBISCuS could not finish on *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's current implementation uses a simple 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, so it outperformed Lusail on *C4*. SPLENDID achieved the best performance only on *C6*, and other systems have comparable performance on this query.

Lusail is superior for all large queries. These queries generate large intermediate results, which explains the high response time of Lusail. Similar to *C5*, *B5* and *B6* contain two disjoint subgraphs joined by a filter variable, which is not supported by systems other than Lusail. For the remaining queries, FedX and HiBISCuS timed out on two queries and returned no results on another two. SPLENDID succeeded only on *B2* and timed out on the rest.

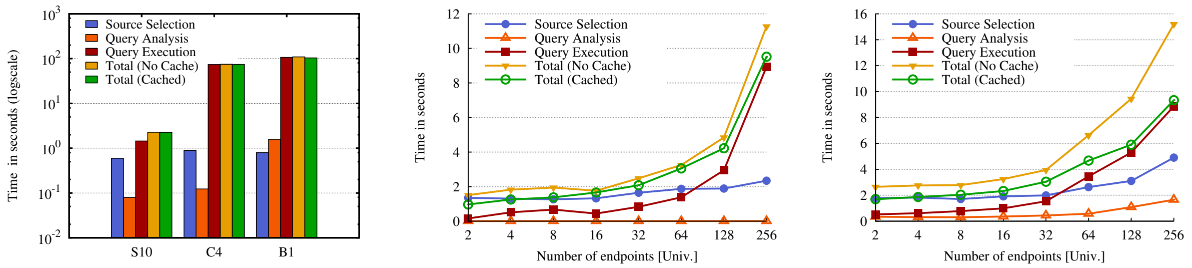
Summary. Lusail is the only system that successfully executes all queries of LargeRDFBench, often showing orders of magnitude better performance than other systems. In contrast, the other systems time out or fail to execute on some queries, in addition to their performance being highly variable and unpredictable.

5.3 Lusail in a Geo-Distributed Setting

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

Using the MS Azure cloud. We create a real geo-distributed setting by deploying SPARQL endpoints in 7 regions of the Azure cloud in the US and Europe. We used 17 D4 Azure VMs (8 Cores, 28 GB memory), 13 for the LargeRDFBench endpoints and four for the LUBM and QFed endpoints, interchangeably. Lusail and its competitors are deployed on one D5_V2 instance (16 Cores, 56 GB memory) in Central US. None of the 17 VMs is located in Central US.

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 11(a) and 11(b) show the query response times of both complex and large queries on LargeRDFBench. We



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

Figure 12: Profiling Lusail by varying query complexity, the number of endpoints, and the data size.

Table 2: Query runtimes (sec) on real endpoints. ZR: zero results error, RE: runtime exception.

	Bio2RDF					**LargeRDFBench**					
	R1	**R2**	**R3**	**R4**	**R5**	**S3**	**S4**	**S7**	**S10**	**S14**	**C9**
Lusail	12.3	8.1	35.6	28.7	13.9	1.9	2.1	1.9	3.3	8.9	2.3
FedX	128.1	721.5	RE	ZR	RE	0.5	0.5	21.6	14.8	453	TO

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 on 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 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, with no time out or runtime errors.

Figure 11(c) shows results on two endpoints of the LUBM dataset. Lusail's query response times increased slightly compared to the local cluster (Figure 9(a)). All queries finished in around 1 second. In contrast, both FedX and HiBISCuS required more than 1,000 seconds; an order of magnitude degradation compared to 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 to evaluate only *Q3* and ran out of memory or timed out on the rest.

Real Endpoints. In this experiment, we use Lusail and FedX to query real independently deployed endpoints. Specifically, we use the Bio2RDF endpoints[5] and a subset of the LargeRDFBench endpoints[6]. We extracted five representative queries from the Bio2RDF query log: *R1*, *R2*, *R3*, *R4*, and *R5* (queries shown in [3]). For LargeRDFBench, we evaluated six queries: *S3*, *S4*, *S7*, *S10*, *S14*, and *C9*. We use a single machine of the *84-cores* cluster to run Lusail and FedX. We show the results in Table 2. For *S3* and *S4*, which are simple and selective queries, FedX outperforms Lusail, as it does when running on a local cluster (Figure 10). FedX was unable to execute four of the other queries, and is one or two orders of magnitude slower than Lusail on the queries that it does execute. This demonstrates that Lusail is capable of answering queries accessing real independently deployed endpoints with good performance.

5.4 Analyzing Lusail

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 12(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 sets of 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. In all cases, query analysis does not add significant overhead.

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 12(b) and 12(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 figures show 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.

Delayed Subqueries. This experiment evaluates different threshold values for identifying subqueries to delay, namely μ, $\mu + \sigma$, and $\mu + 2\sigma$, in addition to delaying only subqueries with outlier estimated cardinalities. We used the Chauvenet criterion [7] for outlier detection. In this experiment, we use our LargeRDFBench deployment in Microsoft Azure. Figure 13 reports the total time for evaluating the queries of each category in LargeRDFBench. For simple and complex queries, $\mu + 2\sigma$ and *Outliers* allowed most subqueries to be evaluated concurrently and delayed only a few of them. Hence, these thresholds missed the opportunity to delay some subqueries that could reduce the communication cost and the cost of joining the fetched data. Thus, $\mu + 2\sigma$ and *Outliers* performed significantly worse than μ and $\mu + \sigma$,

[5]`http://bio2rdf.org/`

[6]`http://manager.costfed.aksw.org/costfed-web`

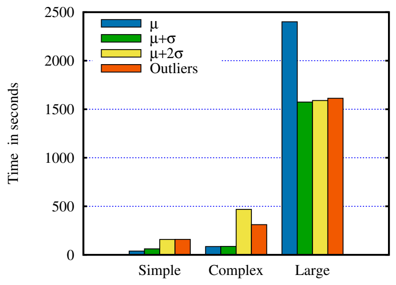


Figure 13: Evaluating different threshold values for identifying subqueries to delay.

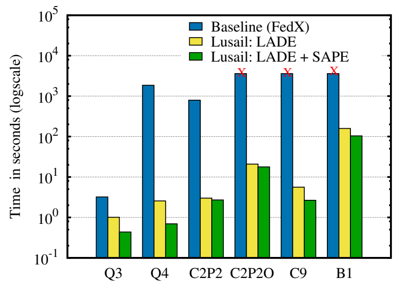


Figure 14: The effect of LADE and SAPE.

as seen in the figure. For large queries, delaying too many subqueries limits parallelism. Thus, μ performed significantly worse than others since too few 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.

Effect of LADE and SAPE: This experiment measures the gain obtained through LADE and SAPE compared to FedX as a baseline. FedX and Lusail are each deployed on a single machine of the *84-cores* cluster, and the endpoints are on the same cluster. We only report results for two queries from each benchmark. We observed similar behavior in most of the queries with medium and high complexity. Figure 14 shows the total response time for each query using FedX, Lusail with LADE alone, and Lusail with LADE and SAPE. FedX takes a significant amount of time for query execution due to its static query decomposition and bound join evaluation. It could not process three queries out of the six within the time limit of one hour. In these three queries, FedX sent a large number of requests to the endpoints and spent the hour waiting for them to finish. LADE decomposition shifts some of the computation of intermediate results from Lusail to the endpoints, which enables Lusail to outperform FedX by up to three orders of magnitude. Using SAPE execution in addition to LADE always improves performance compared to using LADE alone.

In [3], we show additional experiments to demonstrate that Lusail reduces the memory footprint and communication costs compared to FedX.

6. RELATED WORK

Distributed RDF systems [1, 17, 15, 36, 19, 18] deal with data stored in a single endpoint, where the data is replicated and/or partitioned among different servers in the same cluster [2, 25]. 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 [5], SPLENDID [13], and HiBISCuS [31], is based on collected information and statistics about the data hosted by each endpoint. The cost of adding a new endpoint is proportional to the size of the data. Index-free systems, such as FedX [34] and Lusail, do not assume any prior knowledge of the datasets. FedX [34] and Lusail utilize SPARQL ASK queries to find the relevant endpoints and cache the results of these queries for future use. Thus, the startup cost and the cost of adding a new endpoint is small. Federated SPARQL systems usually divide the query into exclusive groups of triple patterns, where each group has a solution at only one endpoint. This decomposition is typically based on the schema and not the data instances. In contrast, Lusail decomposes the query based on checking the data instances, thereby shifting more of the computation of intermediate results to the endpoints.

APlug [28] automatically tunes the execution of a bag of independent tasks. Unlike APlug, the execution of the independent tasks in Lusail, i.e., the subquery ordering, is followed by joining their results. Moreover, Lusail considers the communication cost of executing each subquery to determine the best ordering which balances between the communication cost and the degree of parallelism.

Several efforts, such as Ariadne [6], InfoMaster [9], Garlic [27], and Disco [35], have focused on web-based data integration over heterogeneous information sources [24]. In general, a wrapper is run at each data source to translate between the supported languages and data models. Moreover, systems, such as Piazza [16], coDB [11] and HePToX [8], 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.

7. CONCLUSION

Lusail optimizes federated SPARQL query processing through a locality-aware decomposition (LADE) at compile time followed by selectivity-aware and parallel query execution (SAPE) at run time. The LADE decomposition is based not only on the schema but also on the actual location of data instances satisfying the query triple patterns. This decomposition increases parallelism and minimizes the retrieval of unnecessary data. SAPE query execution orders queries at run time by delaying subqueries expected to return large results, and chooses join orders that achieve a high degree of parallelism. Lusail outperforms state-of-the-art systems by orders of magnitude and scales to more than 250 endpoints with data sizes up to billions of triples.

As future work, we plan to investigate keyword search as a means for querying federated RDF systems, and to develop methods for returning fast and early results during federated query execution. Both extensions aim to facilitate interactive data discovery and exploration on linked web data.

8. REFERENCES

[1] I. Abdelaziz, M. R. Al-Harbi, S. Salihoglu, and P. Kalnis. Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(12):3374–3388, 2017.

[2] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB*, 10(13), 2017.

[3] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga, and P. Kalnis. Lusail: A System for Querying Linked Data at Scale (Extended Version). *CoRR*, 2017.

[4] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga, and P. Kalnis. Query optimizations over decentralized RDF graphs. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 139–142, 2017.

[5] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proceedings of International Semantic Web Conference (ISWC)*, pages 18–34, 2011.

[6] J. L. Ambite and C. A. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proceedings of International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 3–10, 1998.

[7] L. Bol'shev and M. Ubaidullaeva. Chauvenet's test in the classical theory of errors. *Theory of Probability & Its Applications*, 19(4):683–692, 1975.

[8] 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*, 19(2):231–256, 2010.

[9] O. M. Duschka and M. R. Genesereth. Query planning in InfoMaster. In *Proceedings ACM Symposium on Applied Computing (SAC)*, pages 109–111, 1997.

[10] O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.

[11] E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. Queries and updates in the coDB peer to peer database system. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 1277–1280, 2004.

[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 Proceedings World Wide Web Conference (WWW)*, 2011.

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

[14] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics*, 3(2-3), 2005.

[15] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *Proceedings of International Conference on Management of Data (SIGMOD)*, 2014.

[16] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic Web applications. In *Proceedings of International Conference on World Wide Web*, WWW, pages 556–567, 2003.

[17] 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*, pages 1–26, 2016.

[18] J. Huang, D. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[19] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB*, 6(14), 2013.

[20] E. Mansour, I. Abdelaziz, M. Ouzzani, A. Aboulnaga, and P. Kalnis. A demonstration of Lusail: Querying linked data at scale. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 1603–1606, 2017.

[21] 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 *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 930–941, 2006.

[22] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.

[23] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[24] M. Ouzzani and A. Bouguettaya. Query processing and optimization on the web. *Distributed and Parallel Databases*, 15(3):187–218, 2004.

[25] M. T. Özsu. A survey of RDF Data Management Systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.

[26] N. A. Rakhmawati, M. Saleem, S. Lalithsena, and S. Decker. QFed: Query set for federated SPARQL query benchmark. In *Proceedings of International Conference on Info. Integration and Web-based App. (iiWAS))*, pages 207–211, 2014.

[27] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, volume 97, pages 25–29, 1997.

[28] M. Sahli, E. Mansour, T. Alturkestani, and P. Kalnis. Automatic tuning of bag-of-tasks application. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 843–854, 2015.

[29] M. Saleem, A. Hasnainb, and A.-C. Ngonga Ngomo. LargeRDFBench: A billion triples benchmark for sparql endpoint federation (submitted). In *Journal of Web Semantics (JWS)*, 2017.

[30] 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*, 7(5):493–518, 2015.

[31] M. Saleem and A. N. Ngomo. HiBISCuS: Hypergraph-based source selection for SPARQL

endpoint federation. In *Proceedings Extended Semantic Web Conference (ESWC)*, pages 176–191, 2014.

[32] M. Schmachtenberg, C. Bizer, and H. Paulheim. Adoption of the linked data best practices in different topical domains. In *Proceedings of International Semantic Web Conference (ISWC)*, pages 245–260, 2014.

[33] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. FedBench: A benchmark suite for federated semantic data query processing. In *Proceedings of International Semantic Web Conference (ISWC)*, pages 585–600, 2011.

[34] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *Proceedings of International Semantic Web Conference (ISWC)*, pages 601–616, 2011.

[35] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of Disco. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, pages 449–457, 1996.

[36] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4), 2013.