Query Optimizations Over Decentralized RDF Graphs

Ibrahim Abdelaziz* Essam Mansour◇ Mourad Ouzzani◇ Ashraf Aboulnaga◇ Panos Kalnis*
* King Abdullah University of Science & Technology
◇ Qatar Computing Research Institute, HBKU
{ibrahim.abdelaziz,panos.kalnis}@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 communication, 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 various 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. We evaluate Lusail using real and synthetic benchmarks, with data sizes up to billions of triples on an in-house cluster and a public cloud. We show that Lusail outperforms state-of-the-art systems by orders of magnitude in terms of scalability and response time.**

I. Introduction

The RDF (Resource Description Framework) data model is extensively used to represent Web data. RDF data consists of triples of the form ⟨`subject`, predicate, `object`⟩. A key feature is the ability to link two entities from two different RDF datasets maintained by two local independent authorities. Through such links, *large decentralized graphs* can be 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 various 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. The 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.

The emergence of these large decentralized graphs offers unique opportunities to integrate data from multiple RDF datasets via SPARQL queries. Consider the example query (Q) shown below; it finds US presidents, their parties, and their associated news articles. Q integrates results from two endpoints; NYTimes and DBpedia. Evaluating Q independently at each endpoint and concatenating their results will not retrieve anything as none of the endpoints has all the data required to solve the query. We need to traverse the interlinks between the endpoints to return the full answer. Q has to be decomposed into subqueries to be submitted to the relevant endpoints and whose partial results have to be joined. Thus, middleware to efficiently perform federated queries over multiple independent SPARQL endpoints is needed.

```
Q: SELECT ?p ?party ?page WHERE {
  ?p a dbpedia-yago:PresidentsOfUSA .
  ?p <http://dbpedia.org/ontology/party> ?party.
  ?x <http://www.w3.org/2002/07/owl#sameAs> ?p .
  ?x <http://data.nytimes.com/elements/topicPage> ?page.
}
```

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 *topicPage* has an answer only at NYTimes. 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* [5]. 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 triple *sameAs* in query Q. 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 a new approach that would avoid 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, and (*ii*) remote instances. 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 [5] find groupings of triple patterns

[1] http://stats.lod2.eu/
[2] http://bio2rdf.org/

based on schema information, and we are proposing to find groupings based on instance information.

Existing federated SPARQL systems, such as SPLENDID [6], HiBISCuS [7], and FedX [5], cannot determine the location of data instances matching triple patterns. Hence, they retrieve unnecessary data, leading to poor scalability and response time. 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 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 the endpoints.

This paper addresses the above limitations and introduces Lusail, 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 accesses 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 delays 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.
- A cost model that uses lightweight runtime statistics. The cost model is used 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.
- Experiments on real and synthetic benchmarks. We show that Lusail outperforms state-of-the-art systems by up to three orders of magnitude.

II. LUSAIL QUERY OPTIMIZATIONS

This section gives an outline about Lusail query optimization which is performed at (i) compile time to maximize the number of triple patterns per subquery, and at (ii) run time to minimize the intermediate data to be shipped by deciding the optimal query execution order.

A. Locality-Aware Decomposition

In decentralized RDF graphs, we may have a pair of triples where the data instances matching them are not located in the same endpoint. Consider an example from the LUBM benchmark[3], where each endpoint represents a university and different universities are connected through graduate students and professors. Now consider the query shown in Figure 1, Q_a returns all students who are taking courses with their advisors along with the URI and location of the advisors' alma mater. Since a professor may have graduated from one university and be teaching in another university, the variable $?U$ is considered a global join variable.

```
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. 1. Q_a: A SPARQL query over decentralized RDF graphs across different universities.

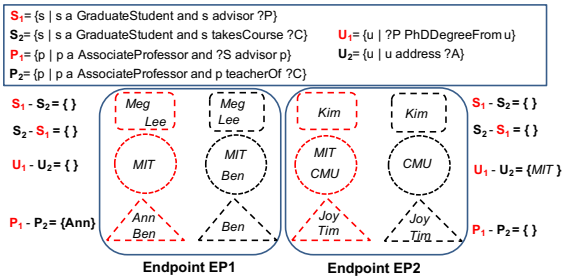


Fig. 2. Sets of data instances match different triple patterns in Q_a.

LADE aims to detect global join variables (GJVs) and consequently decompose the query into independent subqueries. A global join variable (v) is a variable that appears in at least two different triple patterns, where these triple patterns 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.

LADE aims to maximize the number of triple patterns sent together to each endpoint. We first discuss how to merge two triple patterns and then generalize to more than two. Two triple patterns TP_i and TP_j with a common variable are put together in a single subquery under two conditions: (*i*) both triple patterns have the same list of relevant endpoints, and (*ii*) the same 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.

Consider the variable `?U` in Q_a (Figure 1). It appears as an object in TP_i: ⟨`?P`, ub:PhDDegreeFrom, `?U`⟩ and as a subject in TP_j: ⟨`?U`, ub:address, `?A`⟩. Let v_i and v_j be the sets of instances of v that satisfy TP_i and TP_j, respectively. 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). As shown in EP2 in Figure 2, one of the instances matching the variable `?U` in EP2, whose triples are ⟨`?P`, ub:PhDDegreeFrom, `?U`⟩, i.e. the instance MIT matches ?U in this case, while the instance MIT in EP2 does not match ⟨`?U`, ub:address, `?A`⟩; this means the instance MIT may be located in different endpoints. (*ii*) local instances, where all v_i and v_j are located in the same endpoint. For example, all

[3]http://swat.cse.lehigh.edu/projects/lubm

students instance (Meg and Lee) in EP1 in Figure 2 match the variable ?S in ⟨?S, ub:advisor, ?P⟩ and ⟨?S, ub:takesCourse, ?C⟩ are located in the same endpoint.

LADE detects GJVs using the relative complement of v_i in v_j in all relevant endpoints by sending the following SPARQL query to each endpoint.

```
SELECT ?P WHERE {
  ?P rdf:type T .
  ?S<Predicatei >?P.
  FILTER NOT EXISTS { SELECT ?P WHERE {
  ?P < Predicatej > ?C . }} . } LIMIT 1
```

If at least one of these endpoints has instances in v_i but not in v_j, then v is a global join variable. 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 global join variable, 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. This allows us to have simple plans and better parallel execution. 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. Thus, if the query specifies a constant in the triple pattern mentioned in the filter of the check query (i.e. ?Z), Lusail will replace it with a variable in the check query.

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. For example, the variable ?X appears as subject in both ?X ub:advisor ?Y and ?X ub:takesCourse ?Z. Having two empty sets in the same endpoint means that (*i*) any graduate student ?X having an advisor ?Y should take a course ?Z and (*ii*) any graduate student ?X taking a course ?Z should have an advisor ?Y, all located in the same endpoint.

Using the detected GJVs, Lusail decomposes the query into a set of independent subqueries. Lusail utilizes these join variables and the source selection information to decompose the query. The query decomposition algorithm has two phases: branching phase and merging phase. For each GJV, Lusail constructs a tree rooted at the current join variable. An initial set of subqueries is created at the root of the query tree, one subquery per child. Each subquery is expanded through depth first traversal. A triple pattern can be included with a subquery 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 global join variable. If one of the conditions is invalid, a new subquery is created from the current triple pattern and added to the set of subqueries. In both cases, the edge destination node is added to the nodes stack and the edge itself is marked as visited. The merging phase 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 global join variable. Figure 3 shows a possible decomposition for Q_a.

B. Adaptive Query Execution

LADE outputs a set of independent subqueries that can be submitted concurrently for execution at each of its relevant endpoints, such as subqueries SQ1, SQ2, and SQ3 in Figure 3.

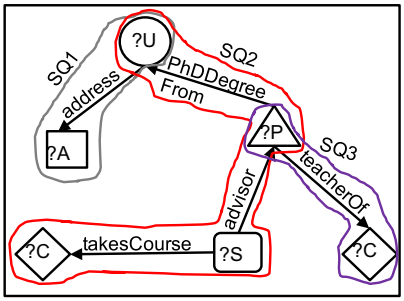


Fig. 3. A possible decomposition of Q_a, where the GJVs are ?U and ?P. Any pair of predicates, which causes a variable to be a GJV, cannot be in the same subquery.

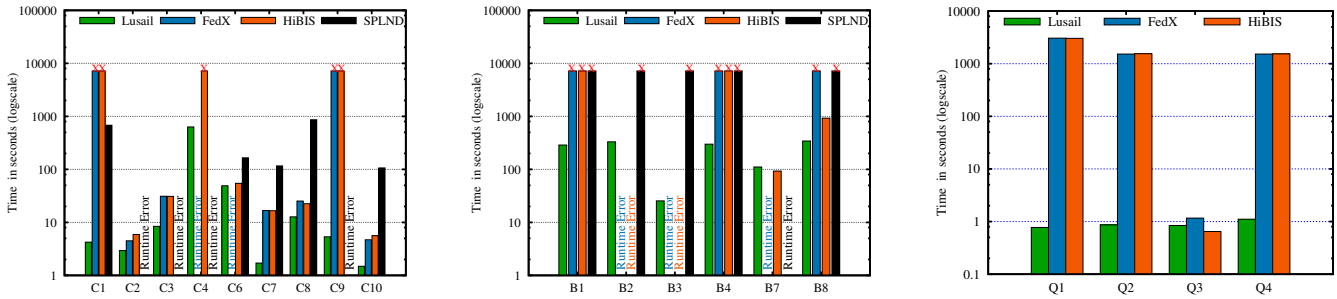
The results of these subqueries will then need to be joined at the global level. If the query is disjoint (i.e. can be evaluated independently at each endpoint and no join is required), SAPE evaluates the whole input query as-is at all relevant endpoints and returns the result. Otherwise, SAPE iterates over all the decomposed subqueries and evaluates each subquery at its relevant endpoint.

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. SAPE estimates the cardinality of the different subqueries and accordingly delays subqueries expected to return large results. The cardinality of a subquery is estimated based on the cardinality of its triple patterns. It is collected during the query decomposition 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 [8], [9]. SAPE assumes that subquery cardinalities follow a normal distribution. It calculates the average μ and standard deviation σ values for all the cardinalities and for all the numbers of relevant endpoints per subquery. Any subquery sq_i with cardinality $C(sq_i) > \mu_C + \sigma_C$ is delayed.

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 the size of 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. 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. SAPE selects the next delayed subquery to evaluate to be the one with the smallest estimated cardinality until no more delayed subqueries are left.

III. Evaluation

We evaluate Lusail by simulating a real scenario on the cloud as well as using real endpoints. We use several real



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

Fig. 4. 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.

datasets, namely QFed [4] and LargeRDFBench[5], and the LUBM synthetic dataset. QFed is a federated benchmark of four different real datasets. LargeRDFBench is a recent federated benchmark of 13 different real datasets with more than 1 billion triples in total.

Simulation on the Microsoft Azure cloud: 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.

Figures 4(a) and 4(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. 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). Thus, we can see that none of the previous systems is as robust or as fast as Lusail.

Figure 4(c) shows results on two endpoints of the LUBM dataset. All queries finished in around 1 second. In contrast, both FedX and HiBISCuS require more than 1,000 seconds. 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: We use Lusail and FedX to query Bio2RDF[6] endpoints. We extracted three representative queries from the Bio2RDF query log: R1, R2 and R3. 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. 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.

[4]https://github.com/nurainir/QFed

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

[6]http://bio2rdf.org/

IV. 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 the 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 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] 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.

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

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

[8] O. Erling and I. Mikhailov, "RDF support in the Virtuoso DBMS," in *Networked Knowledge-Networked Media*. Springer, 2009.

[9] 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.