Cluster-Based Join for Geographically Distributed Big RDF Data

Fan Yang*, Adina Crainiceanu†, Zhiyuan Chen*, Don Needham†

*University of Maryland, Baltimore County
{fyang1, zhchen}@umbc.edu

†United States Naval Academy
{adina, needham}@usna.edu

***Abstract*—Federated RDF systems allow users to retrieve data from multiple independent sources without needing to have all the data in the same triple store. The performance of these systems can be poor for large and geographically distributed RDF data where network transfer costs are high. This paper introduces CBTP, a novel join algorithm that takes advantage of network topology to decrease the cost of processing SPARQL queries in a geographically distributed environment. Federation members are grouped in clusters, based on the network communication cost between the members, and the bulk of the join processing is pushed to the clusters. We use an *overlap list* to efficiently compute join results from triples in different clusters. We implement our algorithms in OpenRDF Sesame federated framework and use Apache Rya triple store instances as federation members. Experimental evaluation results show the advantages of our approach over existing techniques.**

***Keywords*-RDF; SPARQL; Federated Queries; Join; Cluster**

I. Introduction

The Resource Description Framework (RDF) [1] provides a set of specifications for describing resources and relationships and for making semantic queries using the SPARQL language [2]. Billions of RDF triples are now available on the internet. Centralized RDF systems process data locally and provide fast, efficient access to the data contained within their system [3]–[9]. However, in practice, data is often distributed at many different geographic locations. Federated systems transparently map multiple autonomous databases as one entity. A shortcoming of this approach is performance, particularly when queries need to be executed over geographically distributed nodes with large communication overhead.

This paper focuses on improving the performance of Basic Graph Pattern (BGP) SPARQL queries in geographically distributed environments. All SPARQL queries are either BGP queries or contain a BGP query as part of the query. We take advantage of the locality of data as well as the heterogeneity of the communication cost between nodes in federated systems. We organize the nodes into clusters such that the communication cost within a cluster is relatively low. We introduce a Cluster-Based Two-Phase algorithm, CBTP, for federated join. In the first phase, we compute most join results locally, within a cluster, thereby avoiding the high communication cost between clusters. In the second phase, we use a join index-like structure called an *overlap list* to efficiently compute join results from triples in different clusters. Our approach seeks to query and join only the relevant information.

```
Q1:
SELECT ?x, ?y, ?z, ?w
WHERE
{ ?x hasNewPier ?y .
  ?x hasNewShipWrek ?w .
  ?z isAnchoredIn ?y .
}
```

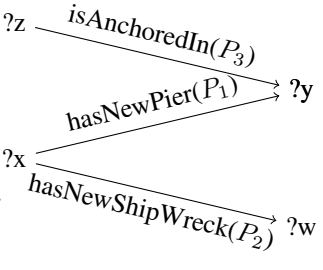


Fig. 1: Query 1

Consider a use case where multiple sources send information to merchant mariners. The mariners need updates on ports, pier facilities, port traffic, and safety of navigation hazards to derive any relevant impact to their own ship and route plan. One possible query they could issue is to retrieve all ships anchored in ports that have a recently built pier, and a recent shipwreck in the area, as shown in Fig. 1.

Suppose there are multiple data sources in Europe, North America, and Asia, and each source stores information about facilities and traffic for ports in nearby regions. Furthermore, most of the shipwreck data is stored at the source near the shipwreck location, but for a few of them (e.g., when there is no source nearby) such data is stored at a source in England. Suppose the coordinator for the query above is located in North America. One possible method to execute this query is called a mediator join. Using this method, the coordinator retrieves data (triples) matching each triple pattern from the corresponding source, unions the results from all sources for each pattern, and then joins the results of the three triple patterns to obtain the final result. In the example above, all sources send all of the locations for a new pier, the location of a new shipwreck, and current port traffic, to the coordinator, which joins the results. The join results provide the ship's navigator the information needed to plan the best route to a new pier. This method has performance issues because many

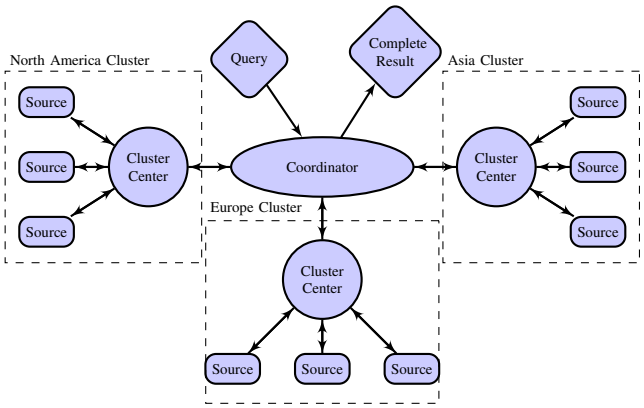


Fig. 2: Geographically Distributed Clusters

triples matching the individual triple patterns need to be sent across continents, which incurs high latency, even if the final query results are relativelly small.

Following our approach, the data sources are organized into three clusters, one for each continent, as in Fig. 2. Most shipwreck data is stored at a source close to the shipwreck location, likely in the same cluster where nearby port information is stored, so most of the join results can be computed locally within a cluster, avoiding the high network latency between clusters. Only a few join results, such as for the shipwreck information stored in the non-local source in England, need information stored across clusters.

We implement our algorithms in OpenRDF Sesame [10], an existing open source RDF processing framework and the Apache Rya [11] cloud-based RDF triple store. Experimental results show that the CBTP algorithm outperforms existing approaches for federated queries in a geographically distributed environment with high network latency, in particular for data with high locality (small data overlap between clusters) and non selective queries (such queries are more expensive).

The remainder of the paper is organized as follows. Background information is reviewed in Section II. In Section III, we introduce the CBTP algorithm and the overlap list, and we introduce maintenance algorithms in Section IV. Section V describes our experimental results. Related work is discussed in Section VI before conclusions.

II. Preliminaries

A. RDF and SPARQL

RDF's data model [1] has <subject, predicate, object> triples, where the predicate expresses a relationship between the subject and the object. For example, the statement "the ship is in the harbor" can be represented as an RDF triple with a subject ("the ship"), a predicate ("is in"), and an object ("the harbor"). RDF triples implicitly represent a labeled directed graph, with edges labeled by the predicates, directed from the subject to the object.

SPARQL is the standard query language for RDF [2]. An important concept in this language is the *triple pattern*, which is similar to a triple, but the subject, predicate, or object can be a variable. In this paper we consider BGP queries, which are conjunctions of triple patterns. Any BGP SPARQL query can be represented by a directed or, if we ignore direction, an undirected *query graph*, where subject or object variables, IRIs, or literals are vertices in the graph, and the predicates are labels for the edges connecting the subjects and predicates. The query graph for Q1 is shown in Fig. 1. We can view a query as a set of query patterns (edges in query graph). SPARQL supports federated queries, allowing information access to be distributed over multiple SPARQL endpoints across the Web.

OpenRDF Sesame [10] (now RDF4J) is an open-source Java framework for processing RDF. It provides out-of-the-box methods for parsing and executing centralized and federated SPARQL queries.

Apache Rya [12] is a cloud-based RDF triple store that supports searches for semantically related triples through SPARQL. Rya uses Apache Accumulo [13] for storage, and provides indexing and query processing capable of scaling to billions of triples across multiple nodes.

B. Federated Join Algorithms

We highlight a common bind join method used in federated systems. To aid in the discussion of the join methods, consider the simple SPARQL query $Q2$ in the following example, which involves a join by subject between two triple patterns:

```
Q2:
SELECT * WHERE
{  ?x hasNewPier ?w .
   ?x hasNewShipWreck ?y .
  }
```

In *bind join*, the most selective triple pattern in the query, say $< *, hasNewPier, * >$, is sent to each data source. Each source returns triples matching the triple pattern. The coordinator performs a union of all the received triples, projects the join column ($?x$), and sends it back to each source along with the second most selective pattern, $< *, hasNewShipWreck, * >$. Each source returns triples matching both the second pattern and the projection of the join column received. The coordinator performs the union to construct the final query result.

III. Cluster-based Two-Phase (CBTP) Algorithm

We now introduce the main idea behind improving the performance of SPARQL join queries in a federated environment. Our goal is to reduce the communication cost of executing a federated SPARQL query by taking advantage of the heterogeneity in network topology - we organize the sources (nodes) in clusters, such that the communication cost within the cluster is lower than the communication cost between clusters. For each cluster, we designate a node as the cluster center. In each cluster, nodes communicate with the cluster center (low communication cost), and the cluster centers communicate with the query coordinator (higher communication cost). In the next sections we present our CBTP join algorithm for federated SPARQL queries.

A. CBTP Algorithm

Setup: The federated system contains a coordinator node and at least two clusters. Each cluster has a cluster center node and some cluster member nodes.

Assumptions: We only consider join algorithms for BGP SPARQL queries in which the predicates are specified and triple patterns share at least one common variable as subject or object. These is the typical case, with more than 90% of such queries in RDF benchmarks and realistical SPARQL queries workloads. If there are no shared variables, existing join algorithms will be applied to compute the required cross-product.

Suppose a query Q contains k triple patterns $P_1, \ldots, P_k$. Each $P_q (1 \leq q \leq k)$ has format (s_q, p_q, o_q) where s_q is the subject, p_q is the predicate, and o_q is the object. Each triple pattern shares a subject or object with some other pattern.

Suppose there are m clusters $C_1, \ldots, C_m$. Let R_{qj} be all triples in cluster C_j that satisfy triple pattern P_q. The result of query Q can be represented as

$$R_Q = \bigcup_{j=1}^{m} R_{1j} \bowtie \bigcup_{j=1}^{m} R_{2j} \ldots \bowtie \bigcup_{j=1}^{m} R_{kj} \quad (1)$$

Using the distributive property of join over union, R_Q can also be represented as

$$\begin{aligned} R_Q = & \bigcup_{j=1}^{m} R_{1j} \bowtie R_{2j} \ldots \bowtie R_{kj} \bigcup \\ & \bigcup_{c_1,\ldots,c_k \in \{1\ldots m\}, \exists i,j: c_i \neq c_j} R_{1c_1} \bowtie R_{2c_2} \ldots \bowtie R_{kc_k} \quad (2) \\ = & R_Q\_intra \bigcup R_Q\_inter \end{aligned}$$

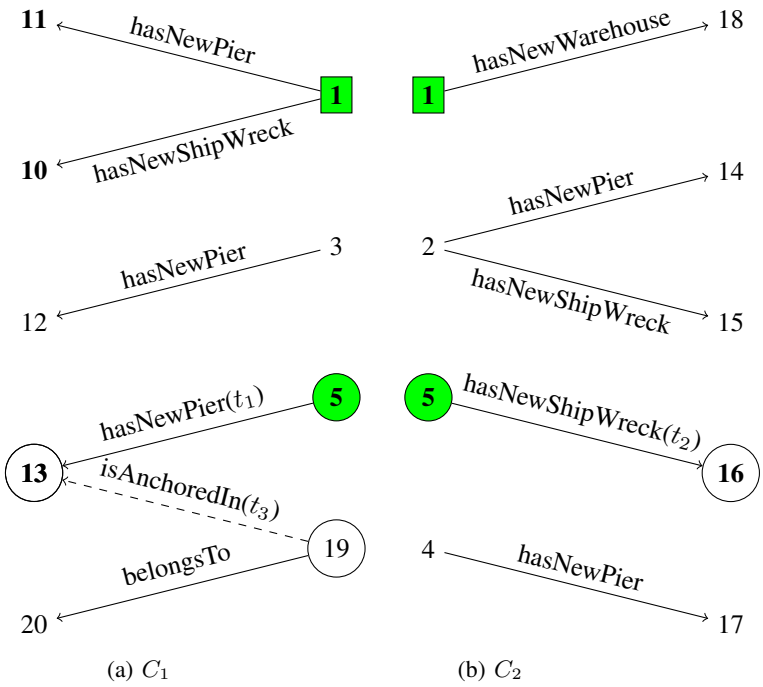
In our cluster-based federation, we compute the intra-cluster join (triples that can be computed locally within a cluster) and the inter-cluster join (triples from different clusters that join to form an answer) and then union the results to obtain the final query answer.

Algorithm: The CBTP algorithm for processing SPARQL queries is shown in Algorithm 1. In Phase I, the intra-cluster join results are computed. The coordinator sends the query to each cluster center (line 1). Each cluster center processes the query to compute the query result based on the data available at the nodes in the cluster (line 2), using any existing federated join algorithm, and returns the results to the coordinator. The coordinator computes R_1, or R_Q_intra, the union of all results received in this phase (line 4). In Phase II, the inter-cluster join results are computed (line 5). Section III-B describes the details.

Running Example: To illustrate how our algorithms work, consider the RDF triples stored in two clusters, C_1 and C_2, as shown in Fig. 3. The values 1, 2, ..., 20 represent IRIs (Internationalized Resource Identifiers). Let's assume we try to find all ships anchored in a port that has both a new pier and a new ship wreck in the area ($Q1$ shown in Fig. 1). As can be seen in Fig. 3, based on the data in each individual cluster only, there is no such ship, but if the data from both

Algorithm 1 : `CBTPAlgorithm`(Q)

1: {Start Phase I: Intra-Cluster Join}
Send query Q to center of each cluster C_j, $j = 1, \ldots, m$
2: Cluster center of C_j computes results for Q as RQ_j based on data stored on nodes in C_j (it can use any join methods such as mediator join, bind join, or semi join).
3: Cluster center of C_j sends results RQ_j to coordinator.
4: Coordinator unions results $R_1 = \bigcup RQ_j$ for all j
{Start Phase II: Inter-Cluster Join}
5: Coordinator computes R_2, a super-set of the query results obtained from joining triples from different clusters using Algorithm 2
6: Coordinator returns union of R_1 and R_2 as results for Q



(a) C_1 (b) C_2

Fig. 3: Graph Data at Two Clusters

clusters is considered, there is one ship, 19, anchored at pier 13, that satisfies the query requirements as port 5 has both a new pier, 13, and a new ship wreck, 16.

Example: To illustrate in more details how Algorithm 1 works, consider the RDF triples in Fig. 3 and we apply Algorithm 1 to evaluate query $Q1$ in Section I. In the first phase, the intra-cluster join is computed: query $Q1$ is sent to the cluster centers for C_1 and C_2 and the cluster centers evaluate the query based solely on the data available at that cluster. In this case, each cluster center returns an empty result to the coordinator. The coordinator unions the results received, and R_1, the result of the first phase, is empty.

In the next section we describe Phase II of the algorithm for efficiently computing the inter-cluster join results.

B. Inter-Cluster Join Using Overlap List

Overlap list: We define the *overlap list* O_j at each cluster center C_j as the set of subject or object values (IRIs or literals)

in the current cluster that also appear as subjects or objects in other cluster(s). In Fig. 3, values 1 and 5 appear as subjects or objects in both clusters, so the overlap lists at each cluster center is O_1=O_2=$\{1, 5\}$, shown in green in Fig. 3.

Query boundary values: We call any subject or object value a *boundary value* if it is part of the overlap list (i.e., it belongs to multiple clusters) and appears in the inter-cluster join result of a query. E.g., 5 in Fig. 3 is a boundary value for $Q1$ whose inter-cluster join result is $(x = 5, y = 13, z = 19, w = 16)$ (circled in Fig. 3). Clearly, non empty inter-cluster join results must contain at least one boundary value, otherwise all results are in the same cluster. The result may contain non boundary values as well. $Q1$'s inter-cluster join result on the example data contains both boundary values (5) and non boundary values (13, 16, 19).

Algorithm 2 : `phaseTwo`(Q with patterns P_1, ..., P_k)

1: Convert query to an undirected graph $G = (V, E)$ where V are variables or literals used as objects or subjects and E are predicates.
2: **for** each vertex $v \in V$ with degree $d \geq 2$ **do**
3: {Compute R_v, the result of a modified bind join for the patterns incidental to v, where intermediate results are intersected with the overlap list}
4: Let $P_{v_1}, \ldots, P_{v_d}$ be the triple patterns incident to v
5: R_v = `bindJoinWithIntersect`(Q_v with patterns $P_{v_1}, \ldots, P_{v_d}$)
6: Use regular bind join (no overlap list used) to compute the join of R_v with remaining query patters. Result is R'_v
7: **end for**
8: Coordinator computes final join by doing the union of all received R'_v

Intuition of our algorithm: The inter-cluster join algorithm, shown in Algorithm 2, is based on two key observations.

Observation 1: If inter-cluster join result of a query Q is not empty, there must be at least one query vertex v in Q's query graph whose degree is at least two and query patterns incident on v match triples from different clusters and match some boundary values, i.e., their subjects or objects appear in overlap list. Therefore, the overlap list can be used to select only such patterns to speed up the query.

For example, 5 matches query vertex $?x$ in $Q1$'s query graph in Fig. 1. $Q1$ contains three query patterns: P_1 = $(?x, hasNewPier, ?y)$, P_2 = $(?x, hasNewShipWreck, ?w)$, and P_3 = $(?z, isAnchoredIn, ?y)$. The patterns incident to $?x$ are P_1 and P_2. They match triples t_1 and t_2 in Fig. 3 and both contain boundary value 5.

Observation 2: Some query patterns may match triples containing only non boundary values, so overlap lists cannot be used to select these patterns. However, such values must be connected to boundary values directly or indirectly in order to be part of the inter-cluster join query result. E.g., P_3 in $Q1$ matches t_3 in Fig. 3 which does not contain boundary values, but value 13 is connected to a boundary value, 5.

Inter-cluster join algorithm: Based on these observations, the inter-cluster join algorithm shown in Algorithm 2 considers all vertices v that could produce a boundary value (line 2) and computes result R_v of joining patterns incident on v (in line 5) using a modified version of the bind join (shown in Algorithm 3). Since this step uses the overlap list to filter the results of each query pattern, the join is efficient if the data has high locality. The remaining patterns are joined without using the overlap list (line 6) to determine any non-boundary values that are part of the query result (line 6).

Algorithm 3 : `bindJoinWithIntersect`(Q with patterns P_1, ..., P_d)

1: $B = \emptyset$ {B stores bounded values of the join columns (the variables shared by multiple triple patterns)}
2: **for** $q = 1, \ldots, d$ **do**
3: Coordinator sends triple pattern P_q and a set B of bounded values to each cluster center
4: Each cluster center of C_j sends P_q and B to each node in the cluster
5: Each node retrieves its triples that satisfy P_q and match the join column values in B and sends these triples to the cluster center
6: Cluster center of C_j unions the retrieved triples
7: Cluster center of C_j sends to the coordinator only the triples that have either the subject or the object in the overlap list O_j.
8: Coordinator unions received triples and adds projected join column values of these triples to B
9: **end for**
10: Coordinator computes R the join using received triples

Algorithm 3: In the modified bind join, the coordinator only communicates with the cluster centers (line 3) and the cluster centers communicate with the nodes in the respective clusters (line 4). The cluster center unions results from the nodes in the cluster and returns results to the coordinator, and only returns triples that have a subject or object in the overlap list (line 7), as these are the only triples that could contribute to an inter-cluster join. The coordinator unions triples received from all the cluster centers, and adds the projected join variable value to B, the set of bindings (line 8). The coordinator computes the join results in line 10.

Example: We apply Algorithm 2 to query $Q1$ in Fig. 1. Nodes $?x$ and $?y$ have a degree greater than one and could identify boundary values. The patterns incident to $?x$ are P_1 and P_2 and the patterns incident to $?y$ are P_1 and P_3. We first use Algorithm 3 to compute the potential inter-cluster results for the join of P_1 and P_2.

The coordinator first sends the triple pattern P_1 to the cluster center of C_1. The center forwards them to the nodes in C_1 and then retrieves triples matching P_1. The results are $< 1, hasNewPier, 11 >, < 3, hasNewPier, 12 >, < 5, hasNewPier, 13 >$. These results are intersected with the overlap list O_1 (which contains $\{1, 5\}$) at the center, and only triples $< 1, hasNewPier, 11 >, < 5, hasNewPier, 13 >$ are

sent to the coordinator.

Following similar processing, the cluster center of C_2 sends an empty result, as the intersection with its overlap list is empty. The coordinator unions the triples received, only from C_1 in this case.

P_2 and the candidate values for $?x$ $\{(x = 1), (x = 5)\}$ are sent by the coordinator to the cluster centers and from there to the nodes. Each cluster matches P_2 and received values and intersects with its overlap list. The cluster center of C_1 returns the triple $< 1, hasNewShipWreck, 10 >$ and the cluster center of cluster C_2 returns triple $< 5, hasNewShipWreck, 16 >$.

The coordinator computes R_x, the join of received triples, and the results are $\{(x = 1, y = 11, w = 10), (x = 5, y = 13, w = 16)\}$ (shown in bold font in Fig. 3).

Regular bind join is used to join the result of the previous step with the remaining pattern P_3. The cluster center of C_1 returns $< 19, isAnchoredIn, 13 >$ to the coordinator. The cluster center of C_2 returns no triples (none matches P_3 and received values for $?y$). The coordinator computes the result of this join as $R'_x = \{(x = 5, y = 13, z = 19, w = 16)\}$.

The same processing that was done for vertex $?x$ is done to join P_1 and P_3, incident on $?y$. The result R_y is empty because P_3's intersection with overlap list is empty so a regular join with remaining pattern P_1 is not needed. R'_y is empty.

The coordinator unions results of all query nodes with degree greater than one (R'_x and R'_y here) as the final Phase II query result (line 8). This result is added to results from Phase I (empty in this case) in line 6 of Algorithm 1. The final result is $(x = 5, y = 13, w = 16, z = 19)$.

Optimization: The inter-cluster join algorithm shown in Algorithm 2 considers all query vertices v with degree at least two. However, if information about the location of data is available, for example based on the `SERVICE` keyword used in SPARQL 1.1 to specify the SPARQL endpoint for given triple patterns, then only vertices v incident on edges from different clusters need to be considered, as only those vertices can produce boundary values. This decreases the processing cost of the algorithm.

The CBTP algorithm gives correct and complete results. The proof is omitted due to space constraints. The CBTP algorithm bind join is more efficient than the regular bind join if the following two conditions are satisfied: 1) the communication cost across clusters is significantly higher than the cost within a cluster so the cost of Phase I of our algorithm is low; 2) data locality is high, i.e., very few values are in multiple clusters so the cost of Phase II (intra-cluster join) is low.

IV. Overlap List Maintenance

In this section we describe the algorithms used to construct and maintain the overlap list at each cluster center. As defined in Section III-A, the overlap list contains the IRIs or literals in the current cluster that also appear as subject or object in another cluster(s).

Each center of cluster C_j, $j = 1, ...m$ maintains the following data structures:

- IRI index, $I_j = \{x | x \in C_j\}$: an index of all the IRI and literals existing in that cluster as subject or object (or a Bloom filter if we trade off space for communication cost)
- Overlap list, $O_j = \{x | x \in C_j \wedge \exists i \neq j : x \in C_i\}$: all IRIs and literals from this cluster that also exist at some other cluster

A. Bulk construction

In order to construct the IRI index I_j based on the data already stored in a federated system, each cluster center issues a "select all" query "SELECT ?s ?o WHERE ?s ?p ?o" in its cluster, and then stores the received subjects and objects in the IRI index. Any data structure that supports efficient key lookups is appropriate for the IRI index.

To construct the overlap list, each cluster center of C_j computes the intersection between its own IRI index and all the other IRI indexes at the other cluster centers. To minimize the data transferred, each cluster center C_j constructs a Bloom filter for the items in its IRI index I_j. This Bloom filter is sent to all the other cluster centers, and the cluster centers send back only the data items matching the Bloom filter. If the item is in the local IRI index, the item is added to the overlap list.

B. Incremental maintenance

To incrementally maintain the overlap list as triples are inserted into the different triple stores, each cluster node notifies the cluster center when a new triple is inserted.

When the cluster center receives an insert message from a node in its cluster, for the triple $< S, P, O >$, the cluster center:

- Updates its IRI index if needed and notifies other clusters if the IRI index was updated: if S or O is not in the IRI index, the cluster center inserts it in the IRI index and sends an "insert triple $< S, P, O >$" message to all other cluster centers and waits for replies
- Updates its overlap list if needed: If the cluster center receives a positive reply from the "insert triple $< S, P, O >$" message, it inserts the value received, S or O, into the overlap list

When the cluster center receives an insert message $< S, P, O >$ from other clusters, the cluster center checks whether S or O are in its IRI index. If S (or O) is in the IRI index, a positive reply with the value in the IRI index is sent to the cluster center that sent the original message. In addition, if S (or O) is in the IRI index but not in the overlap list, S (or O) is added to the overlap list.

V. Experimental Evaluation

In this section we describe the experimental setup and the performance evaluation results of our CBTP algorithm, and comparisons of our approach with regular federated bind join.

A. Setup

Machines: We created six Virtual Machines (VMs), each representing a node in a real-world network. Out of the six nodes, we formed three clusters, each cluster with two nodes, one of the nodes being designated as the cluster center. One of the nodes was randomly selected as the coordinator. Each VM run Ubuntu 14.0.4 (64bit), and had a 6-Core Processor, 8GB RAM, and 500GB hard disk.

Implementation: OpenRDF Sesame [10] supports bind join for federated queries out of the box. We modified Sesame's federated query processing to implement our CBTP algorithm. Because statistics-based query optimization was not enabled in the underlying system, for each query tested, we manually reordered the triple patterns such that the most selective triple pattern was evaluated first. This allowed us to compare the different algorithms on exactly the same query. The overlap list for our algorithm was stored in an Accumulo [13] table at each cluster center, and we used a Java HashList to iterate through that table. We used Sesame Workbench to upload and query data. The development environment is Eclipse Java EE IDE 4.5.2, Java EE 1.8, Accumulo 1.7.4, Hadoop 2.7.5, Rya 3.2.10, Zookeeper 3.4.5, and OpenRDF Sesame 2.7.6. Rya was used as a triple store and the federated query processing component was in Sesame.

Network latency: We simulated network latency variations based on three scenarios: 50ms (typical for regional round trips within North America), 100ms (typical for transatlantic round trips between Europe and the US), and 200ms (typical for transpacific round trips between Asia and the US). For each test, we set up a network delay value between clusters and compare the performance of our CBTP algorithm and the original bind join under varying network delay settings.

Data and Queries: We used data and queries from two standard benchmarks for our experiments. The Lehigh University Benchmark (LUBM) [14] was developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. LUBM synthetically generates datasets using a fixed OWL ontology of university organizations, lecturers, teachers, students, and courses.

The original LUBM data is not distributed. For our experiments, we generated data for six universities and loaded on each node data for one university. We modified the original generated RDF files to have some data that overlaps between the different clusters, to simulate for example a professor that teaches at two different universities. We created three versions of the datasets, with various levels of data locality: high, medium and low. High locality means data is mostly co-located in the same cluster so only a small fraction of IRI are in overlap list (i.e., appearing in multiple clusters). Each dataset had approximately 750,000 triples.

LUBM offers 14 test queries, but not all of them would produce results from multiple universities in a federated scenario. We chose five queries: Query 1, 2, 4, 7, and 9 as test queries. Due to space constraints, results for Query 1, 4 and 9 are not shown, but we note that the results of Query 1 and 4 are similar to that of Query 7, and the results of Query 9 are similar to that of Query 2.

The second benchmark we used is LargeRDFBench (LRDFB) [15], a comprehensive benchmark encompassing real data and typical queries of varying complexities, suitable for analyzing the efficiency of federated query processing over multiple SPARQL endpoints. The different datasets in LRDFB are connected, so data did not need to be modified to have overlapping IRIs. Due to hardware limitations of our experimental environment, instead of using the full data provided in LRDFG for some of the datasets, we randomly sampled 1% of the very large datasets such as $DBPedia$, and 10% of the medium datasets such as $NewYorkTimes$. The resulting datasets contained more than 1,500,000 triples in total. Each dataset was loaded into one cluster.

LRDFB has 40 queries, divided into four different types: simple, complex, large data, and complex+high data sources queries. The complex queries are most meaningful for our algorithms, and we selected queries $C2$, $C3$, $C6$ and $C9$ for experiments. $C6$, shown below, asks for news about actors starring in any movie, and requires data from two datasets, NYTimes and LinkedMDB.

```
SELECT ?actor ?filmTitle ?news ?variants
?articleCount ?first_use ?latest_use
WHERE {
?film purl:title  ?filmTitle .
?film linkedmdb:actor  ?actor .
?actor owl:sameAs  ?dbpediaURI.
?nytURI owl:sameAs  ?dbpediaURI .
?nytURI nytimes:topicPage  ?news ;
nytimes:number_of_variants  ?variants;
nytimes:assoc_article_count ?articleCount;
nytimes:first_use  ?first_use;
nytimes:latest_use  ?latest_use }
ORDER BY (?actor)
```

Due to space constraints, results for $C3$ are not shown, but the performance is similar with that of $C6$.

B. Experimental Results

Figures 4, 5, 6, 7, and 8 show the execution time for LUBM Query 2, 4, and LRDFB $C2$, $C6$, and $C9$ respectively, when we vary the network latency for all queries and the level of data locality for LUBM data. For each query, we compare the regular federated bind join algorithm with the CBTP algorithm.

The results show that as the level of data locality decreases (overlap list size increases) our method's execution time increases because it takes a longer time to compute inter-cluster join in the second phase.

When network latency increases, both join methods take longer to finish. However, the increase for CBTP in most cases is much smaller than that of regular bind join because our method organizes sources in clusters based on the network topology and reduces data transferred between clusters.

When queries are not selective (LUBM Query 2, LRDFB $C6$), our CBTP algorithm is orders of magnitude more efficient than bind join when network latency is over 50 ms (note that the unit of y-axis is 10^4 seconds in Fig 4 and 7) because for such queries many triples need to be transferred to evaluate the join results, and the regular bind join does not use the overlap list filter and more data gets transferred.

When queries are more selective (LUBM Query 7, LRDFB $C2$), the savings provided by our method are less dramatic (Fig 5, 6). This is expected because selective queries require fewer triples to be transferred so regular bind join will be efficient. In fact, when the network latency is very low, our method is about two times as expensive as regular bind join because we have to repeat the join operation(s) in the second phase. However, as network latency increases, our method quickly outperforms bind join and the saving is significant when network latency is large (over 100 ms) and when data has higher locality.

One exception is LRDFB $C9$. Comparing to $C2$, there are two reasons why our CBTP algorithm can outperform bind join in $C2$ but not in $C9$: for $C2$, the data required for the join has high locality so the overlap list filters most intermediate results, but the opposite is true for $C9$. In addition, $C9$ contains highly selective query patterns so the regular bind-join is very efficient.

Overall, the results show that the CBTP algorithm is more efficient than the regular bind join when there is significant network delay between clusters and the data demonstrate locality (the data from different clusters has small overlap). The improvement is more significant for the more expensive non-selective queries. For inexpensive selective queries, existing join methods such as a bind join are sufficient.

C. Maintenance Cost Results

We record the execution time of constructing and maintaining the overlap list for low-locality data. Each IRI index table contains approximately 500,000 items.

Fig. 9 shows the overlap list construction cost in various network latency settings for low-locality LUBM data. The final overlap list contains about 2500 IRIs. Fig. 9 also shows the break down of the time to construct the IRI index, the time to create and use the bloom filter and the time to create the final overlap list.

The time to construct the overlap list is no more than one minute even when network delay is 200 ms and this is comparable to running a single query. Most of the time is spent on creating the IRI index and constructing the overlap list after using bloom filters. The results for LRDFB data are similar and omitted due to space constraints.

Fig. 10 shows the time to maintain the IRI index and the overlap list as 100 new IRIs are inserted into the LUBM data. The IRIs are in multiple clusters, which means they need to be inserted into the overlap list. When an incremental update is done in batches, such as 100 new IRIs per batch, maintaining the overlap list takes around 0.02 seconds per IRI without network latency and about 0.7 second per IRI when network latency is 200ms. If an incremental update is done one IRI at a time, it takes around 1 second per IRI due to the time to establish connections with different sources.

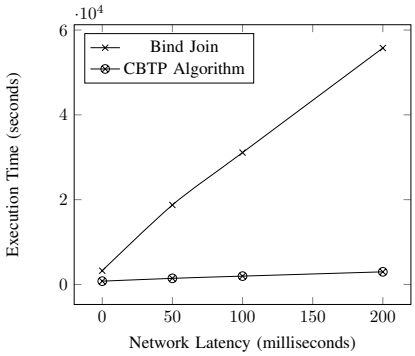
In practice, batch update will be preferred as it is cheaper. Overall, the cost of constructing and maintaining the overlap list is low to moderate, while the improvements in the cost of query evaluation can be significant.

VI. Related Work

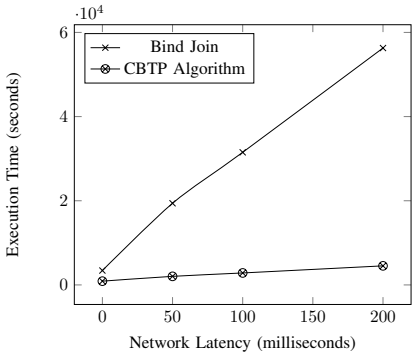
Many federated SPARQL systems exist, such as FedX [16], HiBISCus [17], and SPLENDID [18]. A comprehensive survey and comparison of these systems is found in [19]. However, these systems do not account for data location when matching triple patterns. which results in retrieval of unnecessary data and potentially poor scalability in a geographically distributed system. The CBTP algorithm proposed in this paper can be combined with the join algorithms used in these systems to improve their performance.

Pregelix [20] and Apache Giraph [21] are large-scale graph analytics systems that support the Pregel API [22]. We also apply large-scale graph analytics, but focus on supporting SPARQL queries in federated systems.

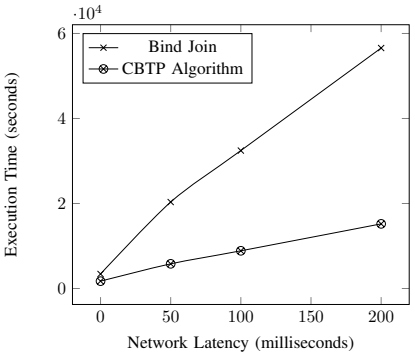
Our work is related to geo-distributed optimization for applications such as graph processing, streaming, and queries. Iridium's [23] distributed job execution aggregates data in centers based on bandwidth constraints, and combines results at job completion time. This is similar to our grouping of federated members into clusters based on geographically influenced network transfer costs. However, Iridium replicates additional copies of data that are tracked by a global manager whereas our overlap list approach to resolving inter-cluster joins does not replicate data. The Hierarchical Synchronous Parallel (HSP) model [24] supports synchronization across data centers with a focus on improving efficiency in inter-data center communication by distinguishing between local and global data centers. This similar to our approach in that HSP's global updates require all-to-all communication among all worker nodes in all data centers. However, our approach sends a Bloom filter only to cluster centers (not all nodes), with which to update our overlap list. Vertex-cut algorithms such as G-Cut [25] seek to minimize the data transfer time of graph processing jobs in geo-distributed data centers. G-Cut uses a two phase approach with the first phase focused on reducing inter-data center data transfer size while the second phase identifies network bottlenecks and refines graph partitioning accordingly, and assumes that computation resources in geo-distributed DCs are less scarce than the bandwidth. We also use a two phase approach, but focus on efficient data transfer when communication costs between clusters is significantly higher than those within the cluster. The JetStream system [26] provides tunable wide-area data analysis by sampling data locally and sending only a small fraction of the data as a degradation technique when bandwidth becomes scarce. Our approach likewise seeks to reduce data transfer costs but focuses on federated joins for data with high locality and relatively expensive inter-cluster communication costs.



(a) High-Locality

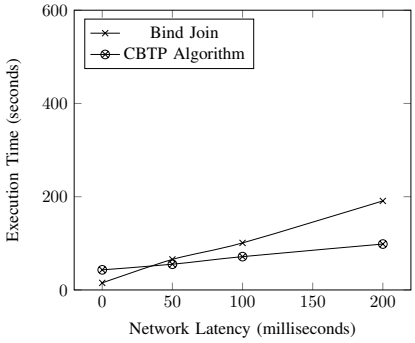


(b) Medium-Locality

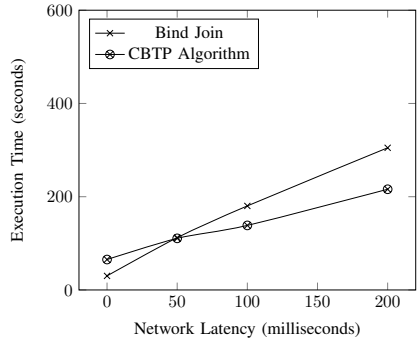


(c) Low-Locality

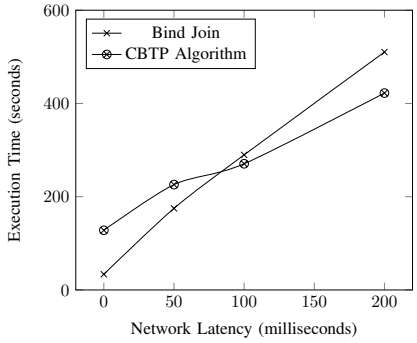
Fig. 4: LUBM Query 2



(a) High-Locality



(b) Medium-Locality



(c) Low-Locality

Fig. 5: LUBM Query 7

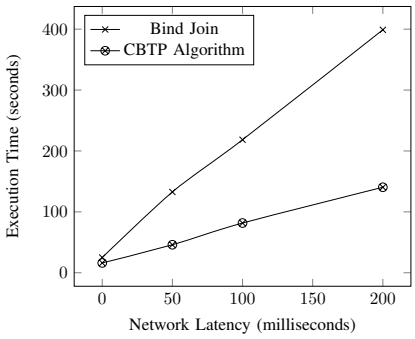


Fig. 6: LRDFB Query C2

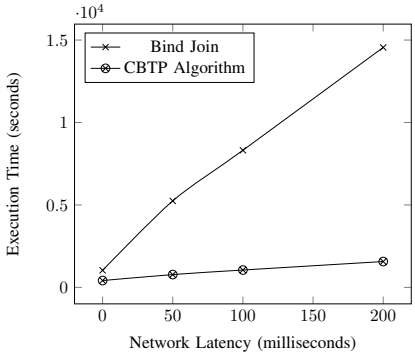


Fig. 7: LRDFB Query C6

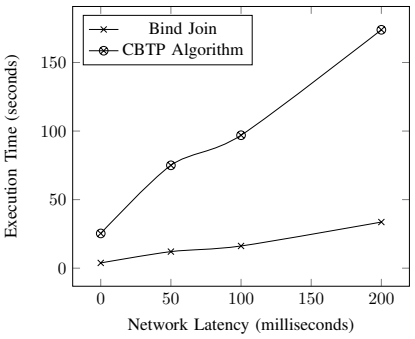


Fig. 8: LRDFB Query C9

DREAM's distributed RDF engine's adaptive query planner [27] replicates the data to all nodes in order to avoid shuffling large amounts of intermediate data. Harbi et al. proposes a hash-based incremental redistribution approach to reduce communication cost of distributed RDF queries [28]. In our approach, each node stores its own data (no redistribution). Nodes are organized in clusters to reduce the communication cost. We avoid transferring large amounts of intermediate data by maintaining a, usually small, overlap list that identifies the data that could participate in an inter-cluster join.

ClusterJoin [29] provides a framework for scalable similarity joins, based on metric distance functions, as an extension of QuickJoin [30] for MapReduce. ClusterJoin partitions records into regions known as home partitions, with similar records clustered and verified by different machines. Records close to the boundary of a home partition, referred to as the outer partition, can still participate in a join. PRoST (Partitioned RDF on Spark Tables) is an Apache Spark based distributed system for RDF storage and SPARQL querying that partitions data through Vertical Partitioning and a Property Table [31]. The ClusterJoin and PRoST approaches are similar with our idea of maintaining an overlap list for records that could participate in an inter-cluster join. However, these algorithms do not work for federated environments and assume a homogeneous communication cost between the distributed nodes.

VII. Conclusions

In this paper we propose CBTP, a novel cluster-based join method that can improve the performance of federated

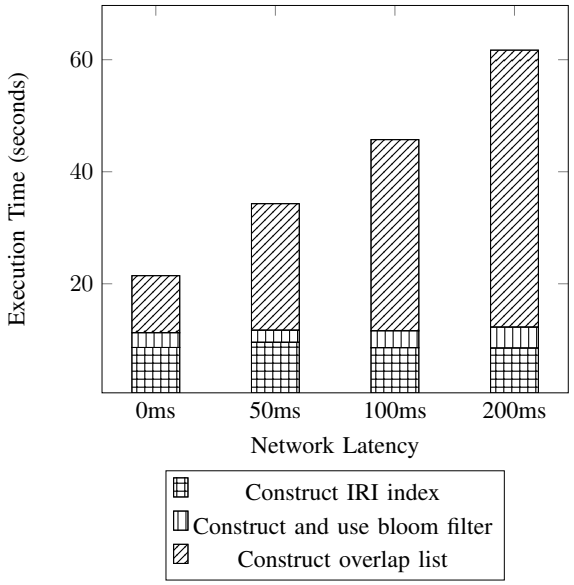


Fig. 9: Bulk Construction Cost

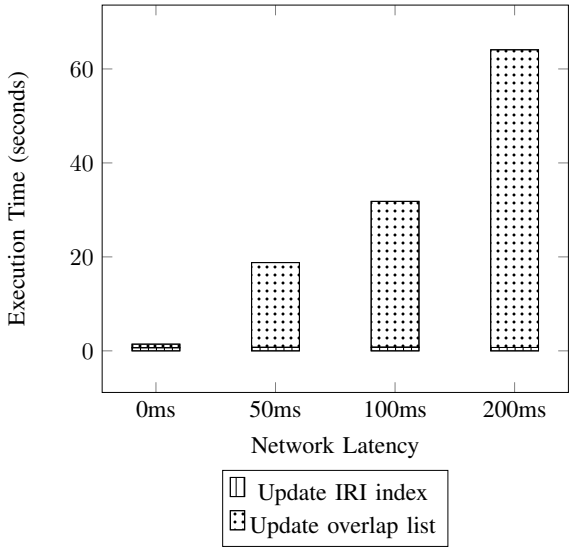


Fig. 10: Incremental Maintenance Cost for 100 IRIs

SPARQL queries over geographically distributed large RDF triple stores when there is significant network delay between clusters of nodes. The key idea is to organize federated nodes into clusters based on network latency, push most computation to occur within local clusters, and use overlap lists to speed up inter-cluster joins. Experimental results demonstrate that the proposed algorithm is up to an order of magnitude faster than existing federated join methods for more expensive queries over data with high locality.

REFERENCES

[1] RDF, "http://www.w3.org/rdf/."

[2] SPARQL 1.1 Overview, "https://www.w3.org/tr/sparql11-overview/."

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007.

[4] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient rdf store over a relational database," in *SIGMOD*, 2013.

[5] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A generic architecture for storing and querying RDF and RDF Schema," in *ISWC*, 2002.

[6] T. Neumann and G. Weikum, "Rdf-3x: a risc-style engine for rdf," in *VLDB*, 2008.

[7] ——, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, Feb. 2010.

[8] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *VLDB*, 2008.

[9] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triplebit: A fast and compact system for large scale rdf data," *VLDB*, 2013.

[10] OpenRDF Sesame, "http://rdf4j.org/."

[11] Apache Rya, "https://rya.apache.org/."

[12] R. Punnoose, A. Crainiceanu, and D. Rapp, "SPARQL in the cloud using Rya," *Information Systems*, vol. 48, no. C, pp. 181–195, Mar. 2015.

[13] Apache Accumulo, "https://accumulo.apache.org/."

[14] LUBM, "http://swat.cse.lehigh.edu/projects/lubm/."

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

[16] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "Fedx: Optimization techniques for federated query processing on linked data," in *ISWC*, 2011.

[17] M. Saleem and A.-C. N. Ngomo, "Hibiscus: Hypergraph-based source selection for sparql endpoint federation," in *ESWC*, 2014.

[18] O. Görlitz and S. Staab, "Splendid: Sparql endpoint federation exploiting void descriptions," in *Second International Conference on Consuming Linked Data*, 2011.

[19] M. Saleem, Y. Khan, A. Hasnain, I. Ermilov, and A.-C. Ngonga Ngomo, "A fine-grained evaluation of sparql endpoint federation systems," *Semantic Web*, vol. 7, no. 5, pp. 493–518, 2016.

[20] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big(ger) graph analytics on a dataflow engine," *VLDB*, 2014.

[21] Apache Giraph, "https://giraph.apache.org/."

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *SIGMOD*, 2010.

[23] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.

[24] S. Liu, L. Chen, B. Li, and A. Carnegie, "A hierarchical synchronous parallel model for wide-area graph analytics," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 531–539.

[25] A. C. Zhou, S. Ibrahim, and B. He, "On achieving efficient data transfer for graph processing in geo-distributed datacenters," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1397–1407.

[26] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 275–288.

[27] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr, "Dream: Distributed rdf engine with adaptive query planner and minimal communication," *VLDB*, 2015.

[28] 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*, vol. 25, no. 3, pp. 355–380, 2016.

[29] A. Das Sarma, Y. He, and S. Chaudhuri, "Clusterjoin: A similarity joins framework using map-reduce," *VLDB*, 2014.

[30] E. H. Jacox and H. Samet, "Metric space similarity joins," *ACM Trans. Database Syst.*, vol. 33, no. 2, pp. 7:1–7:38, Jun. 2008.

[31] M. Cossu, M. Färber, and G. Lausen, "PRoST: Distributed Execution of SPARQL Queries Using Mixed Partitioning Strategies," *EDBT*, 2018.