Towards Efficient Distributed SPARQL Queries on Linked Data

Xuejin Li[1], Zhendong Niu[1], and Chunxia Zhang[2]

[1] School of Computer Science, Beijing Institute of Technology
xuejinli7@gmail.com, zniu@bit.edu.cn
[2] School of Software, Beijing Institute of Technology
cxzhang@bit.edu.cn

Abstract. The fast growth of the web of linked data raises new challenges for distributed query processing. Different from traditional federated databases, linked data sources cannot cooperate with each other. Hence, sophisticated optimization techniques are necessary for efficient query processing. Source selection and distributed join operations are key factors concerning performance of linked data query engines. In this paper, we propose identifier graph based source selection taking into account the logical relationship between triple patterns, and develop effective solutions for distributed join operations to avoid program errors and to minimize network traffic. In experiments, we demonstrate the practicability and efficiency of our approaches on a set of real-world queries and data sources from the Linked Open Data cloud. With the implemented prototype system, we achieve a significant improvement in the accuracy of source selection and query performance over state-of-the-art federated query engines.

Keywords: Linked Data, Semantic Web, Query Federation.

1 Introduction

In recent years, the World Wide Web has evolved from a global information space of linked documents to one where both documents and data are linked [3]. The linked data adopt the general data format (RDF), are described by predefined vocabularies which make them have restrict semantics, and then can be understood by computers. This kind of Web of Data opens up possibilities for new types of applications which can aggregate data from different data sources and integrate fragmentary information from multiple sources to achieve a more complete view. Transparently querying distributed RDF data sources is a key challenge for these possibilities.

With the ever-increasing amount of data sources accessible via SPARQL endpoints, federated query approach has attracted more and more attentions. However, federated query systems for Linked Data are still in their infancy. Improving the query performance of these systems is always in the center of their work. We outline two key factors concerning the performance of federated query systems:

Firstly, the decomposition of original queries must be accurate as far as possible; Secondly, distributed join operations should be effectively executed.

In this paper we concentrate on improving performance of federated SPARQL queries over the Web of Linked Data. To provide users with a transparent view for this Web of Data, the only available information for query decomposition is the user query strings. Hence, we argue that the more clues (presented or implied in query strings) are contributed to source selection, the better accuracy of query decomposition will be. Due to the network latency, distributed join operations may lead to poor query performance. Our goal is to provide optimizations for minimizing the number of remote requests and the amount of network traffic. Thus, sophisticated optimization strategies are needed for efficiently executing distributed join operations. Our main contributions are:

- We utilize a novel approach to make the query decomposition to be convenient and accurate.
- We propose optimization strategies for distributed join operations, mainly including join ordering and join execution.
- We implement the presented optimization techniques in our prototype system and perform experiments on a set of real-world queries and data sources.

The remainder of this paper is structured as follows. In Section 2 we review related work. Details of evaluating distributed SPARQL queries are discussed in Section 3. An evaluation of our prototype system is given in Section 4. Finally, we conclude and discuss future directions in Section 5.

2 Related Work

Related work can be divided into two main categories: (a) query decomposition (b) query optimization.

2.1 Query Decomposition

DARQ [17] extends the popular query processor Jena ARQ to an engine for federated SPARQL queries. It requires users to explicitly supply a configuration file which enables the query engine to decompose a query into sub-queries and optimize joins based on predicate selectivity. Stuckenschmidt [20] presents an index structure called source index hierarchy which is used to determine information sources that contain instances of a particular schema path. Given a predicate path in a dataset, an index hierarchy is constructed, where the source index of the indexed path is the root element. Both two approaches require predicates of triple patterns contained in the query string to be bound. SemWIQ [11] requires all subjects must be variables and for each subject variable its type must be explicitly or implicitly defined. Additional information (another triple pattern or DL constraints) is needed to tell the type for the subject of a triple pattern. It uses these additional information and extensive RDF statistics to decompose the original user query. These requirements limit the variety of user queries.

In other cases, users are required to provide additional information to determine the relevant data sources. For instance, [21] theoretically describes a solution called Distributed SPARQL for distributed SPARQL query on the top of the Sesame RDF repository. Users are required to determine which SPARQL endpoint the sub-queries should be sent to by the GRAPH graph pattern. The association between graph names and respective SPARQL endpoints at which they reside is explicitly described in a configuration file. The W3C SPARQL working group has defined a federation extension for SPARQL 1.1 [5]. However, remote SPARQL queries require the explicit notion of endpoint URIs. The requirement of additional information imposes further burden on the user. On the other hand, the proposed approach hardly imposes any restrictions on user queries.

Recently, several attempts have been made to do source selection without local statistics. FedX [19] asks all known data sources by SPARQL ASK query form whether they contain matched data for each triple pattern presented in a user query. FedSearch[14] is based on FedX and extends it with sophisticated static optimization strategies. If the amount of known data sources is very large(it is common in an open setting), the query performance may leave much to be desired. SPLENDID [6] relies on the VOID descriptions existed in remote data sources. However, a VOID description is not an integral part of Linked Data principles. [1].

2.2 Query Optimization

Research on query optimization has a long history in the area of database systems. Concepts in these research areas have been adopted to optimize queries on local RDF stores. OptARQ [2] reorders triple patterns in SPARQL queries based on their selectivity. Hartig [9] adapted the query graph model (QGM) for SQL queries to represent SPARQL queries. Based on SQGMs, SPARQL queries are rewritten for optimization purpose. Due to the triple nature of RDF data, optimization for queries on local repositories has also focused on the use of specialized indices to accelerate the join operations, e.g. [7].

In [17] Quilitz et.al have adopted some of existing techniques from relational systems to federated SPARQL queries. They present a cost based optimization for join ordering. However, their estimation on the result size of joins is inaccurate by simply setting the selectivity factor for the join attributes to a constant. Because unbound queries generally returning a large result set, other join implementations are proposed as an alternative to local nested-loop implementation of joins, such as pipeline join [8] and semijoin [21]. Due to the variety of the Web, none of these approaches can effectively process all user queries. The reasons are discussed in Section 3.3. In this paper, we propose a novel way, called groupjoin, to execute join operations. The size of group can be modified flexibly for enhancing performance of the system in different situations.

3 Federated SPARQL Query

A federated query system has the similar architecture shown in Figure 1. A mediator(also called query federator) analyzes and decomposes the user query into several sub-queries and distributes them to autonomous data sources which execute these sub-queries and return the results, and then integrates intermediate results into query answers. This section describes in detail how to evaluate distributed SPARQL queries.

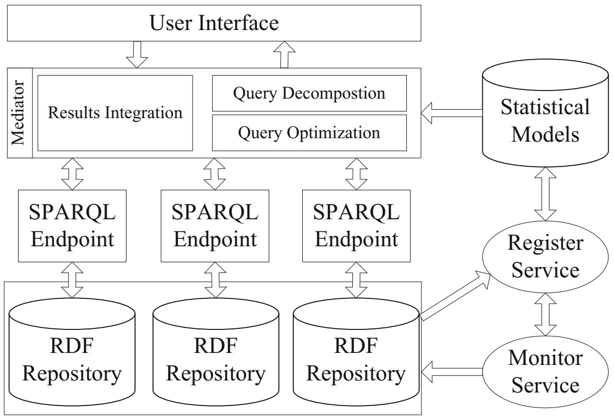


Fig. 1. A common architecture of federated query systems

3.1 Query Decomposition

RDF data is a kind of graph-structured data, and the Web of Linked Data can be seen as a huge distributed RDF graph. SPARQL is a query language for RDF, based on graph patterns and subgraph matching. The simplest graph pattern defined for SPARQL is the triple pattern which is like the RDF triple except that each of the subject, predicate and object may be a variable. The basic graph pattern(BGP) consists of a set of triple patterns which are conjunctive relationship, and also has a graph structure. Other complex graph patterns can be constructed by BGP using SPARQL logical operators(UNION, OPTIONAL). Solutions of a SPARQL query are decided by non-variable parts of triple patterns and the logical relationship between graph patterns. Hence, the decision of query decomposition should be made not only by non-variable parts of triple patterns but also by the logical relationship between graph patterns.

Formal Definitions. Before discussing our approaches, we give formal definitions of concepts used in this section.

Definition 1 (Triple). *Assume that I(IRIs), B(Blank nodes) and L(RDF literals) are pairwise disjoint infinite sets. An RDF statement can be represented* *as a tuple:* $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$*. In this tuple, s is the subject, p is the predicate and o is the object. The tuple representing an RDF statement is called a RDF triple, simply called triple.*

A group of resources with similar characteristics is called a class. The members of a class are instances of the class. In RDF, the predicate rdf:type[1] generally is used to express a source being an instance of a class. For example, P rdf:type C, denotes that P is an instance of C. In the context of this paper, we divide RDF triples into instance triples and class triples. Formally, they are defined as:

Definition 2 (Class Triple). *A RDF class triple (C,p,D) is a RDF triple, both C and D are instances of the rdfs:Class. If c is an instance of C and d is an instance of D, then the triple (c,p,d) is called an instance triple of (C,p,D) and (C,p,D) is called a class triple of (c,p,d).*

Definition 3 (Triple Pattern). *Assume that I(IRIs),B(Blank nodes),L(RDF literals) and V(variables) are pairwise disjoint infinite sets. A triple pattern tp satisfies:* $tp \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$*.*

Similarly, we divide RDF triple patterns into RDF class triple patterns and RDF instance triple patterns. Following the definition above, we give the formalized definitions of RDF class triple patterns and RDF instance triple patterns:

Definition 4 (Class Triple Pattern). *Assume that V is a infinite set of variables, for a given triple pattern* (v_1, v_2, v_3)*, if* $v_1 \notin V$ *and* v_1 *is a instance of the class C, then* $s = C$*, else* $s = v_1$*; if* $v_3 \notin V$ *and* v_3 *is a instance of the class D, then* $o = D$*, else* $o = v_3$*. The triple pattern* (s, v_2, o) *is called a class triple pattern of the triple pattern* (v_1, v_2, v_3) *and* (v_1, v_2, v_3) *is called an instance triple pattern of* (s, v_2, o)*.*

Source Selection. Before source selection, we previously extract class triples from all known data sources. The RDF graph consisting of all class triples from one data source is named by the URI of the SPARQL endpoint of this data source and stored into a local RDF dataset. A RDF dataset represents a collection of RDF graphs. It comprises one default graph and none or more named graphs, where each named graph is identified by an IRI [16]. Besides, we also compute the total number of instance triples and the number of distinct subjects associated with each class triple. The object values domain for predicates is represented by histograms[15]. Consequently, a Web of Linked Class is built on top of the Web of Linked Data. The former is much smaller than the latter, and can be loaded into memory during the system running.

If all triple patterns contained in a SPARQL query are class triple patterns, then this query is called a class query. If all class triple patterns in a class query

[1] In this paper we use the following prefixes: rdf:
`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs: `http://www.w3.org/2000/01/rdf-schema#`

are respectively replaced with an instance triple pattern of them, then the new query is called an instance query of the class query. Before a SPARQL query being evaluated, it is transformed into class queries. Graph patterns containing in class queries are used as identifier graph for source selection.

SPARQL provides the mechanism accessing names of named graphs in a dataset. By using the keyword of GRAPH, a query engine can access the name of a named graph from which the matched data of one or a group of triple patterns come. Our main idea for query decomposition is that firstly translating the original query into class queries; then adding the GRAPH keyword for each class triple pattern; finally evaluating each class query on the statistical model dataset. Classes of an IRI resource can be obtained by dereferencing this IRI. Non-IRI resources are assigned with a common class of rdfs:Literal. SPARQL GRAPH keywords do not change the logical relationships between SPARQL graph patterns. Hence, the result of query decomposition is a comprehensive action of the information of classes, predicates and logical relationships between graph patterns in a query. To the best of our knowledge, there are not any existing approaches considering all these three factors. Hence, we can expect that the presented approach is more accurate than others.

3.2 Cardinality Estimation

Single Triple Pattern. Estimating the cardinality of one single triple pattern $tp = (s, p, o)$ on a data source d includes two steps: Firstly transforming tp into its class triple pattern ctp and evaluating ctp on the class graph dm of d, then a subset $d^{'}$ of d can be decided; the cardinality of tp is estimated on $d^{'}$.

The cardinality of tp can be estimated by the following function:

$$card_d(tp) = card_{d'}(tp) = |T^{'}| \times sel_{d'}(tp) = |T^{'}| \times sel_{d'}(s) \times sel_{d'}(p) \times sel_{d'}(o) \quad (1)$$

Where $|T^{'}|$ is the total number of triples in $d^{'}$. $sel_{d'}(s)$, $sel_{d'}(p)$ and $sel_{d'}(o)$ respectively are the selectivity of s, p and o on $d^{'}$. For s, p and o, if it is a variable, then its selectivity is set to 1. Otherwise, their selectivity are respectively computed by the following functions:

$$sel_{d'}(s) = \frac{1}{|I'|} \quad (2) \qquad\qquad sel_{d'}(p) = \frac{|T_p^{'}|}{|T'|} \quad (3)$$

$$sel_{d'}(o) = \begin{cases} \sum_{s_i \in S} \sum_{p_j \in P_{s_i}} c(s_i, p_j, o_c) & if\ both\ s\ and\ p\ are\ not\ bound \\ \sum_{s_i \in S} c(s_i, p, o_c) & if\ s\ is\ not\ bound\ and\ p\ is\ bound \\ \sum_{p_j \in P_{s_i}} c(s, p_j, o_c) & if\ s\ is\ bound\ and\ p\ is\ not\ bound \\ c(s, p, o_c) & if\ both\ s\ and\ p\ are\ bound \end{cases} \quad (4)$$

where $|I^{'}|$ is the total number of URIs in $d^{'}$, $c(s, p, o_c) = \frac{h_c(s,p,o_c)}{|T^{'}_{(s,p)}|}$, i.e., the frequency of o_c normalized by the number of triples matching s and p, and o_c is the histogram class in which the object o falls into, $|T_p^{'}|$ corresponds to the number of triples matching predicate p in $d^{'}$. If p is bound, then $|T_p^{'}| = |T^{'}|$. Hence, $sel_{d'}(p) \equiv 1$.

Pattern Groups. The function estimating the cardinality of a group of triple patterns $TP = (tp_1, tp_2, ..., tp_n)$ is:

$$card(TP) = min(m_1, m_2, ..., m_n) \prod_{i=1}^{n} \frac{card(tp_i)}{m_i} \tag{5}$$

Where m_i is the number of different values of tp_i in the joint position. If the joint variable is in subject position, then $m_i = |R^{'}|$. If the joint variable is in predicate position, then m_i is the number of different predicates. If the joint variable is in object position, then m_i is the number of different values of all predicates.

Join Cardinality. We compute the join cardinality as

$$card(q_1 \bowtie q_2) = |R_1||R_2|sel_{\bowtie}(q_1, q_2) \tag{6}$$

Where $|R_1|$ and $|R_2|$ are respective the cardinality of q_1 and q_2; $sel_{\bowtie}(q_1, q_2)$ is the join selectivity of q_1 and q_2. It is a reduction factor which depends on the selectivity of the join variable in both datasets. We use the maximum selectivity of the join variable as the join selectivity.

3.3 Join Reordering

The join order determines the number of intermediate results and is thus one of key factor for query performance. For the federated setup, we propose a rule-based join optimizer, which orders a list of subqueries according to a heuristics-based cost estimation. Our algorithm uses a variation of technique proposed in [19] and is depicted in Algorithm 1. Firstly, It selects the subquery with minimum cardinality(line 3) and append it to the result list(line 4). Then, it selects the subquery from remaining subqueries which has minimum join cardinality with the last subquery in the result list (line 7-8) and append it to the end of the result list(line 9).

Algorithm 1. Join Order Optimization

```
1:  order($sqs$ : $list\ of\ n\ joint\ subqueries$)
2:  $result \leftarrow \varnothing$
3:  $mincard \leftarrow min(card(sqs[1-n]))$
4:  $result \leftarrow result + \{sqs[j]\}$//j is the index of subquery with minimum cardinality
5:  $sqs \leftarrow sqs \backslash sqs[j]$
6:  while $sq \neq \varnothing$ do
7:      $q \leftarrow result[result.len - 1]$
8:      $mincost \leftarrow card(q \bowtie sqs[i])$//i is the index of subquery which has the minimum join cardinality with q
9:      $result \leftarrow result + \{sqs[i]\}$
10:     $sqs \leftarrow sqs \backslash sqs[i]$
11: end while
12: return $result$
```

3.4 Join Execution

While pipeline join(PJ) directly passes each solution produced by one operation to the operation that uses it, semijoin(SJ) buffers the obtained variable binding sets and sends them in a batch as conditions in a SPARQL FILTER expression to remote SPARQL endpoints. The former may produce too many concurrent access to remote data sources, and the latter may lead to program errors due to long query strings.

We propose groupjoin(GJ) which restrains the number of the cached solutions in the mediator. In contrast to caching all solutions of the prior sub-query in semijoin, these solutions are divided into some groups, each group contains n solutions. Assume that, q_1 and q_2 are two join query and are respectively evaluated on dataset D_1 and D_2; the cardinality of q_1 is N_1 and the times which D_2 allows one client to access in a period of time is N_C. Then, the size of each group should be $n \geq \frac{N_1}{N_C}$. Again, the maximum length of query string evaluated on D_2 is N_F. Hence, $\frac{N_1}{N_C} \leq n \leq \frac{N_F}{N_T}$, where N_T is the average length of RDF terms in D_2. In practice, n is firstly set to an experimental value between $\frac{N_1}{N_c}$ and $\frac{N_F}{N_T}$. When errors occurred due to too many remote connections, the query engine increases the group size, and thus decreases the number of concurrent threads. When errors occurred due to too many value constraints, the query engine decreases the group size.

The difference between PJ, SJ and GJ lies in the different number of concurrent threads during executing join operations. However, in case of distributed query processing the amount of transferred data has the highest influence on query execution time. Essentially, PJ, SJ and GJ need equal network traffic. For simplicity, we consider the transfer cost of SJ. The cost of a semijoin is estimated as

$$cost_{sj}(q_1 \bowtie q_2) = |R_1||V_1|c_t + |\Pi_V(R_1)||V|c_t + |R_2^{'}||V_2|c_t + 2c_q \tag{7}$$

Where c_t and c_q are the respective transfer costs for one result tuple[2] and one query; R_1 is the result set of q_1; $R_2^{'}$ is the result set of $q_2^{'}$ which is the query with variables bound with values of a result tuple from q_1; V_1 and V_2 are the respective variable set of q_1 and q_2; V is the intersection of V_1 and V_2, $\Pi_V(R_1)$ is the projection of R_1 on V.

While semijoin projects R_1 on V in the mediator, double semijoin(DSJ)[12] executes this operation in D_1. The cost of a double semijoin is estimated as

$$cost_{dsj}(q_1 \bowtie q_2) = (2|\Pi_V(R_1)|+|\Pi_V(R_2^{'})|)|V|c_t+|R_2^{'}||V_2|c_t+|R_1^{'}||V_1 \setminus V|c_t+3c_q \tag{8}$$

Where $|R_1^{'}|$ is the result set of $q_1^{'}$ which is the query with variables bound with values of $R_2^{'}$.

Distributed join operations are parallel executed in GJ. In each thread, we select the optimal way according to function (7-8).

[2] For simplicity, we currently disregard the specific tuple size.

4 Evaluations

We have developed a prototype system(LDMS[3]) implementing the proposed approaches and conducted an experimental study to empirically analyze the effectiveness of it compared with several existing federated SPARQL query systems.

Our evaluation is based on FedBench[4][18]. In contrast to other SPARQL benchmarks[4,13], FedBench focus on testing and analyzing the performance of **federated** query processing strategies on semantic data. It includes two subsets of data sources in the Linked Data cloud: Cross Domain(DBpedia, NYTimes, LinkedMDB, Jamendo, GeoNames) and Life Sciences(KEGG, Drugbank, ChEBI, DBpedia). For each data set, it defines seven queries. In this paper, we discuss the evaluation of graph pattern containing BGP and UNION, omitting other kinds of graph patterns. Hence, thirteen out of fourteen queries are adopted in our experiments. The overview of the data sets is shown in Table 1(a) in terms of number of triples(#Triples), size of statistical models and time taken to create them in hh:mm:ss. Queries are shown in Table 1(b) in terms of number of BGPs and patterns in the WHERE clause and size of results.

Table 1. FedBench datasets and queries used for the evaluation

(a)

Dataset	#Triples	SM Size	SM Time
DBpedia	43.6M	12.8MB	03:55:18
NYTimes	335k	103KB	00:01:27
LinkedMDB	6.15M	368KB	00:27:36
Jamendo	1.05M	33KB	00:5:12
Geo Names	108M	68KB	08:43:47
SW DogFood	104k	646KB	00:00:30
KEGG	1.09M	42KB	00:05:30
Drugbank	767k	195KB	00:02:12
ChEBI	7.33M	23KB	00:25:12

(b)

Query	#BGPs	#Patterns	#Results
CD1	2	3	90
CD2	1	3	1
CD3	1	5	2
CD4	1	5	1
CD5	1	4	2
CD6	1	4	11
CD7	1	4	1
LS1	2	2	1159
LS2	2	3	333
LS3	1	5	9054
LS4	1	7	3
LS5	1	7	393
LS6	1	5	28

The data server was set up using OpenRDF Sesame framework which provides a query service (SPRAQL endpoint) for each data source. Benchmark datasets simulated on the same physical host and were respectively loaded as a single repository with the type of Sesame Native Store. The prototype system(i.e. test client) was on a Windows XP with two Dual-Core Intel Xeon processors (2.8 GHz) and 3GB memory. The server was running a 64 Bit Debian Linux Operation System with two Intel Xeon CPU E7530 processors (each with twelve cores at 2 GHz), 32 GB main memory. The statistical models for data sources were loaded into memory when starting the system.

[3] LDMS is available as Java source code(eclipse project) from the SVN repository: `https://svn.code.sf.net/p/semwldms/code/LDMS/trunk`

[4] FedBench can be downloaded at `http://code.google.com/p/fbench/`

4.1 Evaluation of Join Execution

Based on LDMS, benchmark queries were respectively evaluated by four ways of execution of join operations: pipeline join(LDMS-PJ), nested loops join(LDMS-NLJ), semijoin(LDMS-SJ) and groupjoin(LDMS-GJ). We measured the query evaluation time to see how different ways of join execution affects the overall performance of the query system. For group-join, the size of group was set to 100. All queries were evaluated five times with the five minutes timeout. Figure 2 shows the average time of returning completed answers.

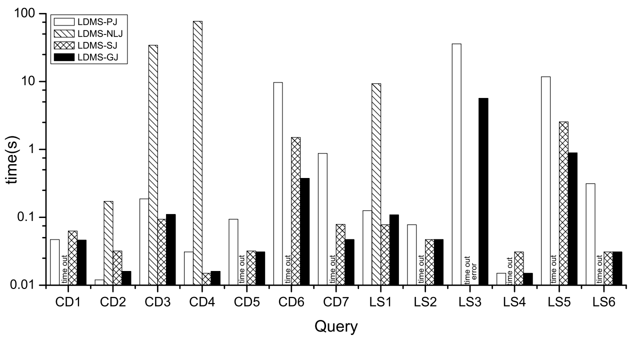


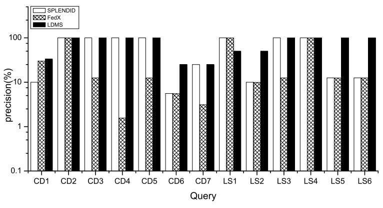
Fig. 2. The Comparison of Time Performance for Different Ways to Execute Join Operations(not including the time for query decomposition)

Due to all intermediate results being transferred over network, the time performance of nested loops join is in the worst situation. However, if all sub-queries have small result sets, it still can be comparable to other ways, i.e. CD2. While pipeline join needing too many remote requests, semijoin suffers from too many intermediate results. When the amount of intermediate results being attached to a sub-query is very large, the internal performance of the remote data sources may become very low, i.e. CD6 and LS5. For LS3 LDMS-SJ sends too long query strings to KEGG data source and encounters program errors. No distributed join operations are concerned in CD1, LS1 and LS2. Hence, LDMS-PJ, LDMS-SJ and LDMS-GJ evaluate these three queries in the same way, and are similar in time performance. For queries that the group size is larger than the size of intermediate result sets, LDMS-SJ is equal to LDMS-GJ, i.e. CD3, CD4, CD5 ,CD7 and LS6. For CD6 and LS4-5, LDMS-GJ is faster than LDMS-SJ.

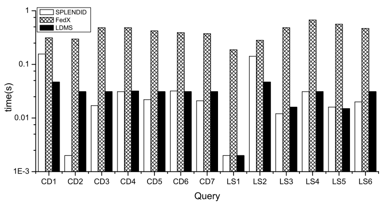
4.2 Comparison with Other Federated SPARQL Query Systems

Some other state-of-the-art federated SPARQL query systems were deployed in our experimental environments, namely SPLENDID and FedX to which LDMS was compared. Every system evaluates all benchmark queries and returns completed answers. We test the accuracy of query decomposition and time performance for these three systems.

Evaluation of Query Decomposition. We define $R = \frac{N_e}{N_E}$ and $P = \frac{N_e}{N}$ to measure the quality of query decomposition, where N_e is the number of effective query plans generated by query systems and N_E is the number of all effective query plans that a original query should have, N is the number of all query plans generated by query systems. An effective query plan means that it can produce query answers. A poor recall will produces incomplete query answers and a poor precision means unnecessary access to the remote data resources which leads to poor time performance. Therefore, we investigated how different strategies affect the accuracy of the source selection. For each query, we look at the recall and the precision of query plans. We test approaches used in LDMS, SPLENDID and FedX respectively. While the recall of these three systems in term of query decomposition is 100%, the precision is different.



(a)



(b)

Fig. 3. The Precision(a) and Time Performance(b) of Query Decomposition

As shown in Figure 3(a), both LDMS and SPLENDID have 100% precision for CD2-5, LS3-4. The query decomposition strategies of SPLENDID can be approximately seen as the integration of approaches used by DARQ and SemWIQ(reviewed in Section 2.1). For queries with unbound predicates, SPLENDID have to use additional SPARQL ASK queries to refine selected data sources. Nevertheless, for six queries LDMS is better than SPLENDID, i.e. CD1, CD6-7, LS2 and LS5-6. SPLENDID misses consideration of the path information which is common in conjunctive queries. The similar shortcoming is happened to FedX. For example, FedX decides that (*?x* <owl:sameAs> *?present*) and (*?present* <rdf:type> <dbpedia-owl:President>) in CD3 are relevant to DBpedia which can not give any answers for the conjunctive query comprising these two triple pattern. FedX has 100% precision for only three out of thirteen queries, i.e. CD2, LS1 and LS4.

FedX directly asks all known data sources whether they contain matched data for each triple pattern in a query. On the other hand, LDMS accesses remote data sources when getting types of IRIs in the position of subject and object. As shown in Figure 3(b), for all queries LDMS is better than FedX in terms of query decomposition time. SPLENDID hardly needs remote requests, hence takes just a little time for query decomposition. For CD1 and LS2, FedX is comparable to SPLENDID. The reason is that these two queries contains triple patterns comprising three variables and SPLENDID needs accessing to all remote data sources for their source selection. For CD4, CD6 and CD7, all predicates are bound and no IRIs presented in the position of subject or object, hence, LDMS needs no remote requests, and then is comparable to SPLENDID in term of decomposition time.

Time Performance. We measure the overall time performance for LDMS, SPLENDID and FedX. Again, all queries were evaluated five times and the average time is used for comparisons. Besides of query decomposition, these three systems are different in join optimization strategies. FedX uses heuristics to reordering join operations whereas SPLENDID and LDMS use statistical information to optimize query plans based on dynamic programming. While FedX uses bound join to optimize traditional implementation of semi-join, SPLENDID adopt nested loops join and pipeline join. LDMS reorders joins based on the result of cardinality estimation of sub-queries and executes join operations in the way of groupjoin.

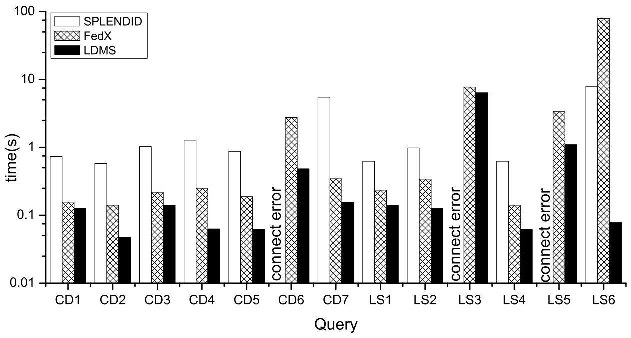


Fig. 4. The Comparison of Time Performance with other state-of-the-art Federated SPARQL Query Systems

The result of the experiment is encouraging, shown in Figure 4. For all queries LDMS is faster than other two systems. However, FedX is comparable to LDMS for queries with a large amount of results, i.e. LS3. It is because that the cost of query decomposition is insignificant for the overall time performance. SPLENDID fails to return results for CD6, LS3 and LS5. The reason is that SPLENDID opens too many connections to data sources and encounters connection errors. For six queries FedX is faster than SPLENDID, i.e. CD1-5, CD7, LS1-2, LS4. For LS6, FedX generates many ineffective query plans and the first sub-query evaluated in some of query plans has non-empty result set. It means that many intermediate results need to be transferred to local federator, but produce no results when join with the next sub-query.

5 Conclusions

We have presented an approach for evaluating SPARQL queries over the Web of Linked Data, based on general statistical models which form a local web of linked classes. We have shown how the statistical model can be used to select relevant sources, and how to optimize distributed join. As revealed by our benchmarks, source selection approaches are effective in terms of accuracy and time performance. We use almost all clues presented or implied in the original user queries to make query decompositions. By decreasing the number of classes of entities, the precision of query decompositions is satisfactory. Compare with the traditional ways of executing join operations, groupjoin makes a compromise between pipeline join and semijoin. By setting an appropriate group size, LDMS is better than or at least comparable to the state-of-art federated SPARQL query systems.

The approach presented in this paper can be seen as a very first step towards a solution for the problems of federated query processing on Linked Data. A number of limitations exist in the current proposal with respect to the generality of the approach and assumptions made. In federation query, query service is necessary for relevant data sources. However, providing a SPARQL endpoint is not required in the Linked Data principles. Both traditional federation query and our approach just omit those datasets not providing query services. For a more general query interface, additional technologies should be considered. The link traversal based query execution [10] is a possible solution.

Though the network communication is the main factor influencing the time performance of systems, the internal efficiency of remote data sources is also important. We aim at providing an infrastructure for developing semantic applications. In a future release, we propose to combine these technologies into a hybrid one.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (No 61272361) and the National Basic Research Program of China (No 2012CB7207002).

References

1. Berners-Lee, T.: Design issues: Linked data (2006), `http://www.w3.org/DesignIssues/LinkedData.html` (2011)
2. Bernstein, A., Kiefer, C., Stocker, M.: OptARQ: A SPARQL optimization approach based on triple pattern selectivity estimation. Citeseer (2007)
3. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems (IJSWIS) 5(3), 1–22 (2009)
4. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) 5(2), 1–24 (2009)
5. Garlik, S.H., Seaborne, A., Prudhommeaux, E.: Sparql 1.1 query language. In: World Wide Web Consortium (2013)

6. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: COLD (2011)
7. Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. In: Third Latin American Web Congress, LA-WEB 2005, p. 10. IEEE (2005)
8. Hartig, O., Bizer, C., Freytag, J.-C.: Executing sparql queries over the web of linked data. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 293–309. Springer, Heidelberg (2009)
9. Hartig, O., Heese, R.: The sparql query graph model for query optimization. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 564–578. Springer, Heidelberg (2007)
10. Ladwig, G., Tran, T.: Linked data query processing strategies. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 453–469. Springer, Heidelberg (2010)
11. Langegger, A., Wöß, W., Blöchl, M.: A semantic web middleware for virtual data integration on the web. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 493–507. Springer, Heidelberg (2008)
12. Mokadem, R., Hameurlain, A., Morvan, F.: Performance improving of semi-join based join operation through algebraic signatures. In: International Symposium on Parallel and Distributed Processing with Applications, ISPA 2008, pp. 431–438. IEEE (2008)
13. Morsey, M., Lehmann, J., Auer, S., Ngonga Ngomo, A.-C.: Dbpedia sparql benchmark–performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
14. Nikolov, A., et al.: Fedsearch: Efficiently combining structured queries and full-text search in a sparql federation. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 427–443. Springer, Heidelberg (2013)
15. Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. In: ACM SIGMOD Record, vol. 14, pp. 256–276. ACM (1984)
16. Prud'hommeaux, E., Seaborne, A., Laboratories, H.P.: Sparql query language for rdf. W3C Recommendation 15 (January 2008)
17. Quilitz, B., Leser, U.: Querying distributed rdf data sources with sparql. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008)
18. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: FedX: Optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 585–600. Springer, Heidelberg (2011)
19. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 601–616. Springer, Heidelberg (2011)
20. Stuckenschmidt, H., Vdovjak, R., Houben, G.J., Broekstra, J.: Index structures and algorithms for querying distributed rdf repositories. In: Proceedings of the 13th International Conference on World Wide Web, pp. 631–639. ACM (2004)
21. Zemánek, J., Schenk, S., Svatek, V.: Optimizing sparql queries over disparate rdf data sources through distributed semi-joins. In: International Semantic Web Conference, Posters & Demos (2008)