Dynamic join order optimization for SPARQL endpoint federation

Hongyan Wu, Atsuko Yamaguchi, and Jin-Dong Kim

Database Center for Life Science, Research Organization of Information and Systems, Japan
{wu,atsuko,jdkim}@dbcls.rois.ac.jp

Abstract. The existing web of linked data inherently has distributed data sources. A federated SPARQL query system, which queries RDF data via multiple SPARQL endpoints, is expected to process queries on the basis of these distributed data sources. During a federated query, each data source may consist of a search space of nontrivial size. Therefore, finding the optimal join order to minimize the size of intermediate results from different sources is key to optimizing the performance of such federated queries. In this study, we present a dynamic optimization approach to determining join order, which can find more optimized join plans than static optimization approaches. Our experimental results show that our proposed approach stably improves the performance of a federated query as the query becomes increasingly complex.

Keywords: linked data, SPARQL, federated query, dynamic join order optimization

1 Introduction

Linked data technology has substantially contributed to the freeing of data confined in individual silos; however, searching over such data is still performed within a single SPARQL endpoint, making it difficult to truly affirm that data are truly freed from their respective silos even in the linked data space.

A number of federated query systems have been developed to enable search across multiple endpoints. Although it is difficult to assert that the performance of these query systems is close to production level, the research community is continuously trying to improve such performance [2,3,5,6,8,10]. In this paper, we propose a novel technique, i.e., dynamic join order optimization, to significantly improve the performance of federated search.

A federated query inherently has to explore multiple endpoints, and while traversing these endpoints, results from one endpoint must be joined with results from the next endpoint and so on. Here each endpoints may consist of a search space of nontrivial size. To efficiently perform the search across these multiple search spaces, determining the optimal join order is key to good performance.

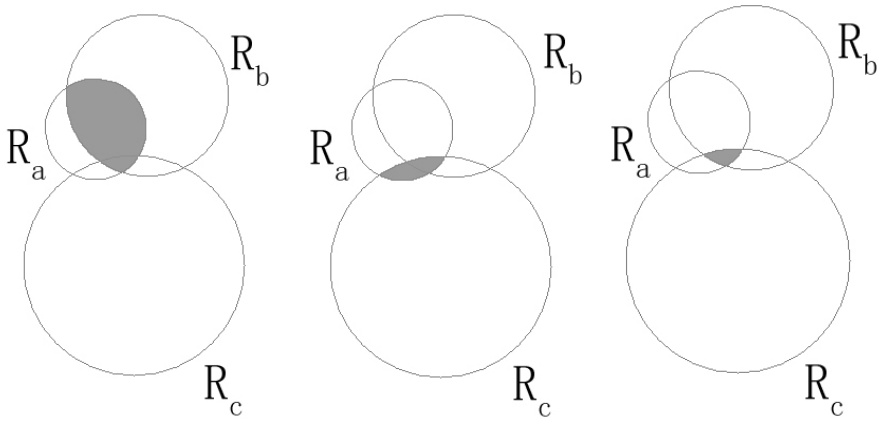
Join order optimization has been a research topic for a number of years [4,11–13]; however, in these studies, the common approach is to somehow try to find

the optimal join order before beginning actual exploration into the endpoints. We therefore call this static join optimization. Considering the importance of join order on the performance of a SPARQL query, we argue that join order cannot be sufficiently optimized at the onset of the query; further, by utilizing intermediate results obtained during search, join order can be significantly improved. We present a simple algorithm for dynamic join order optimization as well as an implementation in the form of an extension to FedX.

Our experimental results show that dynamic join order optimization is effective in controlling the search space size, thereby avoiding explosions in size. We also developed a new benchmark for evaluating the join optimization of federated query performance. This benchmark is developed to include more complex join operations than those introduced in FedBench [8]. Our experimental results here show that our dynamic join order approach stably improves the performance of federated search.

2 Related work

In relational databases, associated data entries are maintained in tables consisting of any number of columns; in RDF, data pieces are maintained in triples, the smallest unit of representation for typed binary relationships. Therefore, join operations generally occur much more frequently when processing a SPARQL query than when processing a corresponding SQL query.



(a) Static approach (b) Dynamic approach (c) Final results for two approaches

Fig. 1. Intermediate results for the static and dynamic approaches

Suppose we have a query that can be decomposed into three subqueries, Qa, Qb, and Qc, which have answers Ra, Rb, and Rc, respectively, from three different endpoints. Then, final answers are to be those that satisfy the constraints

set by the three subqueries. In Figure 1(c), the three circles Ra, Rb, and Rc represent the sets of results of the three subqueries, with the gray area representing the final results. To reach the set of final results, there are six distinct join orders, i.e., (1) A → B → C; (2) A → C → B; (3) B → A → C; (4) B → C → A; (5) C → A → B; and (6) C → B → A. Regardless of which join order is selected, the final set of results is the same; however, the number of intermediate results that must be handles varies on the basis of the different join orders. For example, if subquery Qc is executed first, Rc must be handled as the initial set of intermediate results; however, we would like to avoid that choice because $|Rc|$ produces the largest set of intermediate results among the three possible subqueries.

If the size of the intermediate results is known or can be estimated in advance, the join order may be optimized. For example, the result size of the individual subqueries may be estimated in advance as $|R_a| < |R_b| < |R_c|$. Based on this information, the join order may be optimized as A → B → C. Below are the necessary operations that must occur in the given order:

1. Receive result set R_a.
2. Bind variables in query Q_b using result set R_a and then submit intermediate results to E_b.
3. Receive result set $R_a \cap R_b$.
4. Bind variables in query Q_c using result set $R_a \cap R_b$ and then submit intermediate results to E_c.
5. Receive final result $R_a \cap R_b \cap R_c$.

With the given join order, the size of the intermediate result sets that must be handled is $|R_a| + |R_a \cap R_b|$. This is more or less the scenario in which most federated search systems have been developed in terms of join order optimization, i.e., to better optimize the join order, attempt to estimate the result set sizes of individual subqueries with heuristics or statistical information.

In this paper, we argue that even if the initial estimation is performed perfectly, there is still large room for further optimization. Note that after Qa is first executed, there are two choices for the next execution, i.e., Qb and Qc. Although $|Rb|$ is estimated to be smaller than $|Rc|$, choosing Qc for the next execution is in fact a more optimal choice because $|Ra \cap Rc|$ (i.e., Figure 1(b)) is smaller than $|Ra \cap Rb|$ (i.e., Figure 1(a)). To select the optimal choice in this case, we propose a dynamic join order optimization approach that evaluates queries as follows: (1) evaluate the size of all subqueries, obtaining $|R_a| < |R_b| < |R_c|$; (2) evaluate Qa, then apply Ra to Qb and Qc, noting that $|R_a \cap R_c|$ is less than $|R_a \cap R_b|$; (3) evaluate $|R_a \cap R_c|$; and (4) join Qc. Therefore, the join order is A → C → B. Here the dynamic approach obviously performs better than the static approach because the intermediate result space $|R_a| + |R_a \cap R_c|$ is smaller than the static approach space (i.e., $|R_a| + |R_a \cap R_b|$).

To date, research regarding join order optimization, both in relational database and RDF data management systems, has been centered on static optimization in which optimization is performed only once before queries are actually executed. As an example, FedX builds a subquery for a group of triple patterns

in which each triple exclusively shares a single relevant source. FedX assumes this type of exclusive subquery, and the subquery with fewer free variables has a high selectivity ranking. The assumed selectivity ranking and variable counting technologies are not suitable for all situations as queries become complex. DARQ [6], SPLENDID [3], ADERIS [5], Avalanche [2], and other similar systems use pre-computed information, such as service description or VoID, to estimate selectivity and optimize join order; however, none of these can overcome the fragility of static optimization techniques. More specifically, the search space changes as the query is processed. Based on this and the frequency of join operations in a SPARQL query, we argue that join order should be optimized by utilizing intermediate results with a dynamic approach.

3 Dynamic join order optimization model

3.1 Static join order optimization

To best introduce our dynamic join order optimization algorithm, we first show a simple algorithm that uses the static join order strategy. Here we assume the existence of a *sortSubQueries* operation to sort subqueries by some measure and an *evaluateQuery* operation to output a set *preResults* of results for variables appearing in a given SPARQL query.

Algorithm 1 Query execution with static join order optimization

```
1: function StaticJoin(setSubQueries: a set of subqueries)
2:     listSubQueries ← sortSubQueries(setSubQueries)
3:     preResult ← ∅
4:     while listSubQueries is not empty do
5:         curSubQuery ← pop(listSubQueries)
6:         preResult ← evaluateQuery(preResult, curSubQuery)
7:     end while
8:     return preResult
9: end function
```

Algorithm 1 shows the flow of query execution when a static join order optimization scheme is applied. Given the *setSubQueries* set of subqueries, the algorithm first sorts the subqueries on the basis of estimations of their result sizes and then executes the subqueries in the given order. In other words, the optimal join order is determined before the execution of any subqueries, and the join order does not change during execution, which is why we call it a "static" optimization strategy.

3.2 Dynamic join order optimization

Algorithm 2 shows the flow of query execution with dynamic join order optimization. Unlike the static optimization strategy described above, the optimal

subquery to be executed next is determined at each step of query execution by considering the intermediate results obtained thus far. We therefore call this approach a "dynamic" optimization strategy.

Algorithm 2 Query execution with static join order optimization

```
1: function DynamicJoin(setSubQueries: a set of subqueries)
2:     preResult ← ∅
3:     while setSubQueries is not empty do
4:         curSubQuery ← findOptimalSubQuery(setSubQueries, preResult)
5:         preResult ← evaluateQuery(preResult, curSubQuery)
6:         setSubQueries ← setSubQueries − {curSubQuery}
7:     end while
8:     return preResult
9: end function
```

In the algorithm, **findOptimalSubQuery** finds the subquery with the highest selectivity among all subqueries (line 4). On line 6, the executed subquery is removed from the subquery set, and then this process repeats until all subqueries finish.

Finding the optimal subquery In Algorithm 3, we apply a greedy strategy at each step to find the subquery that has the smallest result size.

Algorithm 3 Finding the optimal subquery

```
1: function findOptimalSubQuery(setSubQueries, preResult)
2:     optimalSubQuery ← setSubQueries[0]
3:     minSize ← MAX_VALUE
4:     for each subQuery in setSubQueries do
5:         if |setSubQueries| equals 1 then
6:             break
7:         end if
8:         size ← estimateResultSize(subQuery, preResult)
9:         if size < minSize then
10:            optimalSubQuery ← subQuery
11:            minSize ← size
12:        end if
13:    end for
14:    return optimalSubQuery
15: end function
```

Estimating result size There are many approaches for estimating the result size of a subquery, for example, using pre-computed statistical information. In

this paper, our implementation uses COUNT queries that do not need any precomputed information. More specifically, we bind previous subquery results to each remaining subquery, construct a COUNT query, and send it on the fly to the relevant sources to determine under the current conditions how many intermediate results they will produce.

Note that a COUNT query is a SPARQL query with the form "select count(*)..." that evaluates the result size of a subquery. We construct COUNT queries for all subqueries as follows: (1) search the triple pattern with a bound value from previous results $preResult$; (2) bind the variables in the remaining subqueries, i.e., $setSubQueries$, with their corresponding values; and (3) use UNION keywords to combine multiple small queries for a subquery into a large query to decrease the number of COUNT queries. An example of our approach here is shown in Figure 2; note that this example comes from our benchmark Q7 and that the bold font portion represents the bound variable and its value.

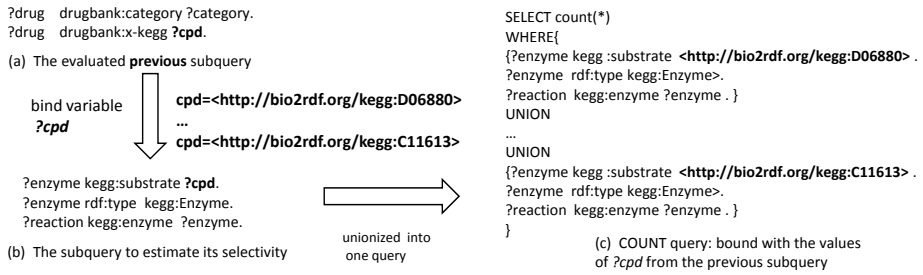


Fig. 2. An example using a COUNT query to evaluate selectivity for a subquery

For dynamic join order optimization, the system must apply all previous query results to the candidate subqueries; however, when there are a large number of values in the intermediate results, it is costly to bind all values to the remaining subqueries and execute the large query. Note that for join order optimization, we need only a rough estimate of the size of the query results on which the subqueries may be ordered. This estimation does not need to be very precise because a small difference in the size of results will not significantly impact the overall performance.

Thus, rather than exhaustively consider the entire set of intermediate results, we take a small sample of size n and order the subqueries by the size of the results after binding relevant variables with the sample values. In this work, we simply set the size of n to be 3. While it may be necessary to estimate the optimal sample size, at this point, we assume that it is not a critical factor for the reason noted above.

Instead of estimating the cost of expressions with VoID as SPLENDID, the **estimateResultsSize** function actually sends the COUNT query to its relevant data sources. Here, we note two important observations: (1) the performance cost

of a COUNT query at its local endpoint is not very large and (2) a COUNT query returns only one number, which is far less information than that if a full result set was returned.

4 Evaluation

4.1 Evaluation of join optimization

We investigated how dynamic join optimization influences the query performance in comparison with the static join. As we noted above, FedX is the fastest engine among the current federated SPARQL endpoint query systems according to recent benchmarks. We therefore implemented all the functions, including source selection, on the basis of the FedX system, and compared the differences before and after using dynamic join optimization in conjunction with the FedX system. Further, we evaluated SPLENDID, which is expected to produce a good join order plan using statistical information and optimizing plans on the basis of dynamic programming techniques.

FedBench is a comprehensive benchmark suite for federated semantic data that considers the evaluation of UNION, FILTER, and OPTIONAL clauses; however, we note that almost all queries in this benchmark have a common characteristic, i.e., they include a single triple pattern with two bound variables and only one free variable, as shown in the query below from Cross Domain evaluation CD6.

```
SELECT ?name ?location ?news
WHERE {
?artist <http://xmlns.com/foaf/0.1/name> ?name .                              (1)
?artist <http://xmlns.com/foaf/0.1/based_near> ?location .                    (2)
?location <http://www.geonames.org/ontology#parentFeature> ?germany .         (3)
?germany <http://www.geonames.org/ontology#name> 'Federal Republic of Germany' (4)
}
```

Triple pattern (4) with two bound variables usually has a higher selectivity. A good join optimization plan should execute this type of triple pattern at an earlier stage in a sequence of joins; however, this type of triple pattern can be simply identified even with very simple optimization technologies, such as the variable counting technique used in the FedX system to count the number of bound variables. To better evaluate the influence of dynamic and static joins, we designed a benchmark to evaluate join optimization for federated SPARQL endpoint queries.

Benchmark setup For our benchmarks, we used five real biological SPARQL endpoints from the Bio2RDF project [1], which is a different setup than FedBench [8], SP^{2}Bench [9], and the fine-grained evaluation of SPARQL endpoint federation systems [7], all of which use a simulated federated environment and synthetic data or a subset of real data. For the life science field, FedBench uses three biological datasets, namely KEGG, ChEBI, and Drugbank. Because the

SPARQL endpoint for CHEBI in the Bio2RDF project [1] is still under construction, we selected KEGG, Drugbank, SIDER, OMIM, and PharmGKB.

These datasets connect to one another closely by relationships between gene, drug, disease, reaction, side effect, and others. Table 1 presents the details of each dataset. The data are far more complicated than the FedBench life science data. The largest biological dataset in FedBench is a subset of ChEBI that includes 7.33 million triples, 28 predicates, and a single type. In the Bio2RDF project, the server of each endpoint is set to return a maximum of 10,000 results at a time, regardless of the real result size. This restriction is commonplace to lessen the burden on the server. Note that all settings in the Bio2RDF servers are beyond our control.

Table 1. Bio2RDF dataset

Dataset	Endpoint	#Triples(M)	#Pred	#Types
Drugbank	http://cu.kegg.bio2rdf.org/sparql	3.48	105	91
OMIM	http://cu.drugbank.bio2rdf.org/sparql	8.35	101	34
SIDER	http://cu.pharmgbk.bio2rdf.org/sparql	16.81	39	16
KEGG	http://cu.sider.bio2rdf.org/sparql	47.87	141	63
PharmGKB	http://cu.omim.bio2rdf.org/sparql	265.17	88	50

This benchmark focuses on testing the join operation in the SPARQL endpoint federation. We consider the following points in designing the queries: (1) the number of triple patterns (#Tp) varies from two to nine; (2) the number of queried endpoints (#Src) has a size ranging from two to five; and (3) the number of returned results (#Res) ranges from 1 k to 109 k. Queries returning large result set sizes are very useful when integrating data from multiple data sources. Table 2 shows the query characteristics in detail.

Table 2. Bio2RDF query characteristics

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
#Tp	2	4	4	4	8	9	6	8
#Src	2	2	2	2	5	5	5	5
#Res	1	5	5	5	9492	132	32003	111962

In addition, the query set considers the numbers of variables in a triple pattern. RDF data could connect to each other via different paths, which brings about more free variables. The fewer bound variables, the more difficult it is to estimate selectivity. Here Q2 and Q3 include one triple pattern in which all variables are free, Q3 changes the position of the triple pattern, and Q4 increases another such triple pattern. The rest of the queries consider the influence of the triple pattern with two bound variables. In this case, Q5, Q7, and Q8 have only one triple pattern with two bound variables, whereas Q6 has two such

triple patterns. Next, Q7 is a variation of LS4 from FedBench, with Q7 obtained by slightly modifying the bound variables, thereby increasing the result size. Further, Q5 is a variation of Q7 obtained by changing the connected dataset and constructing a more complicated star Q8 subquery. Here Q6 tests a query with complicated star subqueries, evaluating the query connecting three datasets. Finally, Q1 is designed to evaluate the extreme case with only two join triple patterns.

We sequentially executed each query five times, removing the largest and smallest values, calculating the mean value of the three remaining values.

Query performance Figure 3 summarizes query performance, and Table 3 shows how intermediate results changed. Dynamic join optimization outperformed the original static FedX system during all queries, except for Q5. As for the time cost, for Q1, Q2, Q6, and Q8, the dynamic approach was faster than the original FedX system. FedX failed on Q4, which has two triple patterns in which all variables are free [1]. With regard to the result completeness, the dynamic approach returned all results for all queries, whereas Fedx returned incomplete results for Q2, Q3, Q6, and Q7. Finally, SPLENDID returned all results for Q1, Q2, and Q4, in which Q2 and Q4 were slower than the dynamic join and Q1 was slightly faster; note that SPLENDID failed all other queries by reaching the one-hour timeout limitation.

Intermediate results shown in Table 3 detail the query performance of both FedX and our dynamic join approach. Intermediate results for the first step, namely the results of the first subquery, show the selectivity of the first subquery. In the table, the number outside the bracket shows the real intermediate result size that the subquery should return, whereas the number inside the bracket shows the actual intermediate size returned within the 10,000-result limitation of the server.

The real intermediate result sizes of Q1, Q2, Q3, Q4, Q6, Q7, and Q8 of FedX were far larger than those of the dynamic join optimization; therefore, FedX was much slower for queries Q1, Q2, Q6, and Q8. For Q7, because of the restrictions on the returned size, the returned intermediate results size was 10,000 (though it should be 80,460), which was less than that of the dynamic join; therefore, the query seemed faster; however, Fedx returned incomplete results, while our dynamic approach returned all results. For Q3, Fedx failed in the second step because this step returned zero results, while our dynamic join approach successfully finished the query in the third step. For Q5, FedX and our dynamic approach produced the same number of intermediate results and the same join plan. In this case, the dynamic approach needed an additional join order optimization cost and was therefore a little slower. We did not measure the details of the intermediate results for SPLENDID.

[1] A SPARQL compiler error occurs when FedX joins a certain intermediate result with another subquery. The dynamic join avoids this problem because the number of intermediate results is much less than that of FedX.

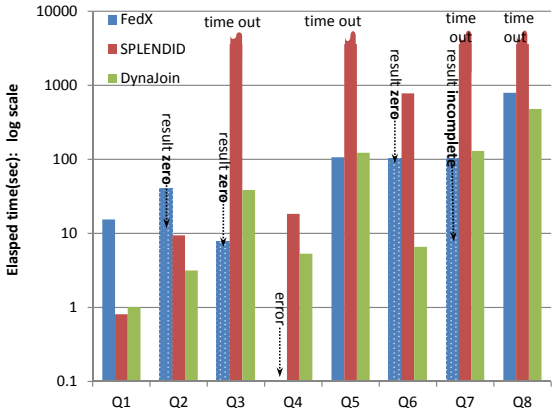


Fig. 3. Bio2RDF results

Table 3. Intermediate results of the first two steps

	1st step		2nd step	
	FedX	Dyna	Fedx	Dyna
Q1	14609 (10000)	2	1	1
Q2	95443 (10000)	1	95443 (10000)	3
Q3	91656866 (10000)	1	0	10000
Q4	14609 (10000)	1	14609 (10000)	3
Q5	19	19	9492	9492
Q6	70115 (10000)	2	0	132
Q7	80406 (10000)	4323	>10000[a]	32077
Q8	36	36	60362	27

[a] The exact size could not be measured because its previous step was not completely executed.

We investigated why Fedx returned no results for Q2. Figure 4(a) and 4(b) illustrate the produced join plan of Q2. In the figures, the number inside the bracket shows the intermediate result after executing the operation. The first evaluation produced by FedX was the *exclusive group*. With the limitation of the OMIM server, the evaluation returned 10,000 intermediate results that contributed no final results; here the actually produced intermediate result size was 95,443. The dynamic join approach sends COUNT queries; thus, determining the fourth triple pattern has the highest selectivity, thereby returning only one result. It then binds the results of variable *?o2* to the other triple patterns, constructs the COUNT queries, and sends them to the relevant endpoints. In this case, the dynamic approach judged the third triple pattern to have fewer results and finally joined the exclusive group. The reason why Q3 and Q6 returned no results is similar to that of Q2.

For Q7 and Q8, it was more difficult to make a join order plan. There is a single triple pattern with two bound variables, which seemingly has higher selectivity. For Q7, FedX first produced a larger initial search space of 80,406, which partially contributed to the final results and therefore returned only part of the results. The dynamic join first evaluated the group (i.e., 4323 results from the fourth and sixth triple patterns), with the search space size being far less than that of Fedx. Consequently, FedX returned only part of the results, whereas the dynamic join returned all results.

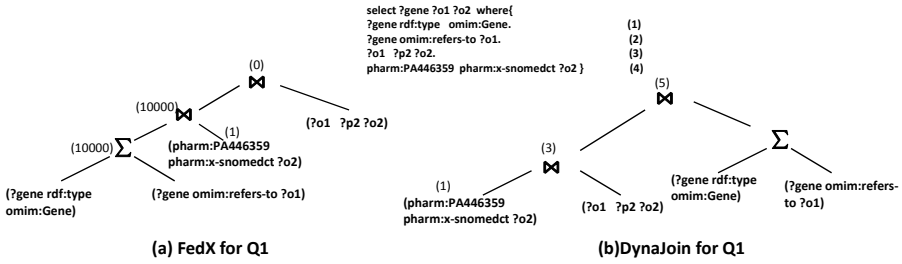
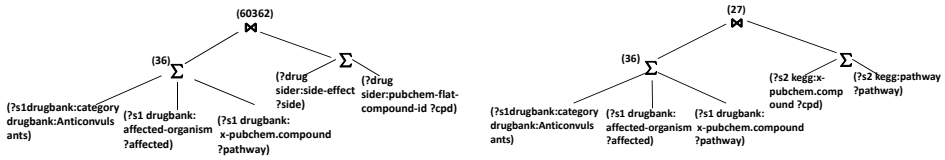


Fig. 4. Join order and intermediate result sizes (inside the brackets) for Q2.

For Q8, 111,962 results were returned-the largest size in this group of queries. The query was evaluated across three endpoints, as shown in Figure 5. Both FedX and our dynamic join first evaluated the exclusive group (i.e., the first three triple patterns) at the Drugbank endpoint. Next, FedX evaluated the second exclusive group (i.e., the fourth and fifth triple patterns); however, the dynamic join approach judged the second exclusive group to have more results than the third exclusive group (i.e., the seventh and eighth triple patterns). Evaluating the third exclusive group earlier substantially reduced the size of the intermediate results, thereby accelerating the query.



(a) FedX and SPLENDID for Q8. (b) DynaJoin for Q8.

Fig. 5. Join order and intermediate result sizes (inside the brackets) for Q8.

For Q5, FedX and our dynamic approach produced the same join plan. The dynamic approach needed an additional join order optimization cost; therefore, FedX was slightly faster. Table 4 shows the additional overhead and their corresponding percentages accounting for the total query time in detail. The largest overhead here was 3.74 seconds for Q5. Consequently, the size increased and the query became heavier, thereby causing the optimization cost to no longer seem insignificant.

In addition, our evaluation shows that SPLENDID cannot produce a better join plan than our dynamic approach despite using pre-computed statistical information. More specifically, we checked the join plan produced by SPLENDID. For Q7 and Q8, SPLENDID produced the same join order as FedX, which generated far larger intermediate results than our dynamic approach. For other queries, SPLENDID produced the same join order plan as our dynamic join

Table 4. Additional overhead of our dynamic join approach

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
time(sec)	0.4	2.33	2.96	3.49	3.74	1.29	1.09	2.24
%	40.5	73.9	7.7	65.8	3.0	19.5	0.8	0.4

approach. The additional cost of the dynamic approach for Q1 resulted from the two COUNT queries, while SPLENDID used pre-computed information to evaluate the selectivity of the two triple patterns.

We also checked the difference when using an index cache; however, we do not provide details here because the cache was not used in the dynamic join order procedure. Here source selection was implemented in the same way as that in case of FedX, which does not impact performance; therefore, the aforementioned conclusions still hold.

4.2 Fedbench benchmark

As mentioned in the above section, the FedBench benchmark cannot measure the performance of join optimization in the federated query well because of its simplicity in producing a join plan; however, in this section, we still provide evaluation results with the Fedbench benchmark as a reference.

Our experiments were conducted on the AWS platform, with five m3.2xlarge instances for the Cross Domain dataset and four instances for life science data. These instances were configured with Intel(R) Xeon(R) CPU E5-2670 v2 2.50 GHz 4 Core CPU with 30 GB RAM and high network performance property (AWS standards) with a 64-bit GNU/Linux operating system and the 64-bit Java VM 1.7.0_75. All datasets were stored with an 8 GiB general purpose SSD EBS, except for the Geonames dataset, which used a 100 GiB one. Endpoints used open-source Virtuoso 07.00.3203.

Table 5 summarizes the FedBench dataset, while Table 6 presents query characteristics. #Tp., #Src, and #Res represent the number of triple patterns, data sources, and results, respectively. Figure 6 presents our experimental results.

Except for query LS6, FedX was slightly faster than our approach, with a maximum difference of less than 0.5 seconds. Our proposed dynamic join eventually generated the same join plan as FedX. Therefore, the cost difference mainly came from the additional optimization cost of our proposed dynamic optimization algorithm. Overall, the additional cost is not substantial. Further, as the queries in the life science field become heavier than queries in the cross domain, the additional cost will decrease.

CD1 shows an extreme case in which only two triple patterns were joined. In this query, Fedx simply identified the triple pattern with higher selectivity. Our dynamic join approach seemed to experience a large cost (0.5 seconds) for optimization; however, the evaluation of Q1 in our designed benchmark, which also joined two triple patterns, showed our dynamic join approach to be much faster than FedX. In such cases, they applied different join order plans. LS6

illustrated a special case in which our dynamic join outperformed FedX. The results of this query are different from what was described in the FedX paper; the FedX team has confirmed these results with our current dataset and settings. We are jointly investigating the reasons why these inconsistencies exist.

Table 5. FedBench datasets

Dataset	#Triples(M)	#Pred	#Types
DBpedia subset	43.6M	1063	248
GeoNames	108M	26	1
LinkedMDB	6.15M	222	53
Jamendo	1.05M	26	11
New York Times	335k	36	2
KEGG	1.09M	21	4
ChEBI	7.33M	28	1
Drugbank	767k	119	8

Table 6. Query characteristics

Cross Domain(CD)				Life Science (LS)		
Query	#Tp.	#Src	#Res	#Tp	#Src	#Res
1	3	2	90	2	2	1159
2	3	2	1	3	4	333
3	5	5	2	5	3	9054
4	5	5	1	7	2	3
5	4	5	2	6	3	393
6	4	4	11	5	3	28
7	4	5	1	5	3	144

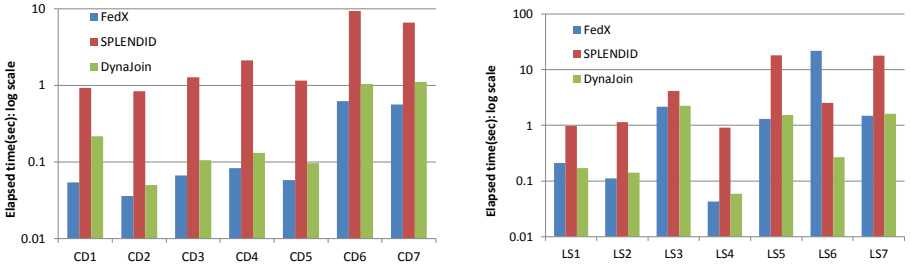


Fig. 6. FedBench results

5 Conclusions

In this paper, we proposed a novel dynamic join order optimization technique. Because the search space of SPARQL queries is always changing, we believe that the join order should be dynamically optimized during query execution, considering the frequency and importance of the join operation in such SPARQL queries. We perform a SPARQL query by executing a group of subqueries in which we optimize the join order by binding the variable values from previous subqueries to the remaining subqueries and then evaluating the next intermediate result size and selecting the plan with the minimum intermediate result size. Both the Fedbench benchmark and our heavier federated biological benchmark proved that in comparison with the static optimization approach, our proposed

dynamic approach engine can stably present an optimal join plan and therefore improve the performance of a federated query, with the degree of improvement becoming clearer as the query becomes more complex. Our dynamic approach does introduce additional overhead with its multiple updates of the join plan, with the overhead being significant in queries that return a small number of results and therefore have join orders that are not complex; however, as queries become more complex and result sizes increase, the optimization cost becomes increasingly insignificant.

Note that the overhead of the COUNT queries could be further controlled by parallelizing the COUNT queries and setting timeout limitations. For the first returned COUNT query, we could assume that it has less of a join cost because the amount of data, the scale of server computational ability, or the degree of network cost is better than others. We plan to implement this in the future to gain a better understanding here. In addition, although we implemented selectivity estimation via COUNT queries in this paper, other approaches are available. With fine-grained metadata, selectivity estimation could be estimated with less cost, although previous results provide concrete instances.

Acknowledgements

This work was supported by the National Bioscience Database Center (NBDC) of the Japan Science and Technology Agency (JST). We also thank the continued support from the FedX team for evaluating FedBench.

References

1. Bio2rdf, http://bio2rdf.org/
2. Basca, C., Bernstein, A.: Avalanche: Putting the spirit of the web back into semantic web querying. In: 9th International Semantic Web Conference (ISWC2010) (November 2010), http://data.semanticweb.org/conference/iswc/2010/paper/527
3. Grlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: In Proceedings of the 2nd International Workshop on Consuming Linked Data (2011)
4. Haas, P.J., Naughton, J.F., Seshadri, S., Swami, A.N.: Selectivity and cost estimation for joins based on random sampling. Journal of Computer and System Sciences 52(3), 550 – 569 (1996), http://www.sciencedirect.com/science/article/pii/S0022000096900410
5. Lynden, S.J., Kojima, I., Matono, A., Tanimura, Y.: Aderis: Adaptively integrating rdf data from sparql endpoints. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA (2). Lecture Notes in Computer Science, vol. 5982, pp. 400–403. Springer (2010), http://dblp.uni-trier.de/db/conf/dasfaa/dasfaa2010-2.html#LyndenKMT10
6. Quilitz, B., Leser, U.: Querying distributed rdf data sources with sparql. In: Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications. pp. 524–538. ESWC'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1789394.1789443

7. Saleem, M., Khan, Y., Hasnain, A., Ermilov, I., Ngonga Ngomo, A.C.: A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web Journal (2014), http://svn.aksw.org/papers/2014/fedeval-swj/public.pdf
8. Schmidt, M., Grlitz, O., Haase, P., Ladwig, G., Schwarte, A., Tran, T.: Fedbench: A benchmark suite for federated semantic data query processing. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N.F., Blomqvist, E. (eds.) International Semantic Web Conference (1). Lecture Notes in Computer Science, vol. 7031, pp. 585–600. Springer (2011), http://dblp.uni-trier.de/db/conf/semweb/iswc2011-1.html#SchmidtGHLST11
9. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: Sp2bench: A sparql performance benchmark. CoRR abs/0806.4627 (2008)
10. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: A federation layer for distributed query processing on linked open data. In: The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II. pp. 481–486 (2011), http://dx.doi.org/10.1007/978-3-642-21064-8_39
11. Steinbrunn, M., Moerkotte, G., Kemper, A.: Optimizing join orders. Citeseer (1993)
12. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web. pp. 595–604. WWW '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1367497.1367578
13. Swami, A., Schiefer, K.: On the estimation of join result sizes. In: Jarke, M., Bubenko, J., Jeffery, K. (eds.) Advances in Database Technology EDBT '94, Lecture Notes in Computer Science, vol. 779, pp. 287–300. Springer Berlin Heidelberg (1994), http://dx.doi.org/10.1007/3-540-57818-8_58

Appendix: Query Set

```
Q1: Find out the gene resource related to "ADRAR".
select ?gene ?p
where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p "ADRAR"^^<http://www.w3.org/2001/XMLSchema#string>
}
Q2: Find out the genes related to diabetes.
select ?gene ?o1 ?o2  where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene <http://bio2rdf.org/omim_vocabulary:refers-to> ?o1.
?o1   ?p2 ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2 }
Q3: Find out the genes related to diabetes.
select ?gene ?o1 ?o2 where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p1 ?o1.
?o1   <http://bio2rdf.org/omim_vocabulary:x-snomed>  ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2 }
Q4: Find the genes related to diabetes.
select ?gene ?o1 ?o2 where{
?gene <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>   <http://bio2rdf.org/omim_vocabulary:Gene>.
?gene ?p1 ?o1.
?o1   ?p2 ?o2.
<http://bio2rdf.org/pharmgkb:PA446359> <http://bio2rdf.org/pharmgkb_vocabulary:x-snomedct> ?o2}
Q5: Find out the generic name ,title, side effect for all the anti-allergic agents.
select * where{
?drug <http://bio2rdf.org/sider_vocabulary:generic-name> ?generic.
?drug <http://purl.org/dc/terms/:title> ?drug_name .
?drug <http://bio2rdf.org/sider_vocabulary:side-effect> ?side.
?drug <http://bio2rdf.org/sider_vocabulary:pubchem-flat-compound-id> ?cpd.
?generic <http://purl.org/dc/terms/title> ?generic_name.
?side <http://purl.org/dc/terms/title> ?side_effect.
?drug_drugbank <http://bio2rdf.org/drugbank_vocabulary:category>
<http://bio2rdf.org/drugbank_vocabulary:Anti-Allergic-Agents>.
?drug_drugbank <http://bio2rdf.org/drugbank_vocabulary:x-pubchemcompound> ?cpd        }
Q6: Find out clinical phenotype features, general and specific functions, and omim articles about F8 gene.
select * where {
?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/omim_vocabulary:Phenotype> .
?s <http://www.w3.org/2000/01/rdf-schema#label> ?o.
?s <http://bio2rdf.org/omim_vocabulary:clinical-features> ?clinicFeature.
?s <http://bio2rdf.org/omim_vocabulary:article> ?article.
?s <http://bio2rdf.org/omim_vocabulary:x-uniprot> ?protein.
?drug  <http://bio2rdf.org/drugbank_vocabulary:gene-name> "F8"^^<http://www.w3.org/2001/XMLSchema#string>.
?drug <http://bio2rdf.org/drugbank_vocabulary:x-uniprot> ?protein.
?drug  <http://bio2rdf.org/drugbank_vocabulary:general-function> ?genFunction.
?drug  <http://bio2rdf.org/drugbank_vocabulary:specific-function> ?speFunction }
Q7: Find out all the drugs, which are substrate of some enzyme, their category and reaction.
select * where {
?enzyme <http://bio2rdf.org/kegg_vocabulary:substrate> ?cpd.
?enzyme <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://bio2rdf.org/kegg_vocabulary:Enzyme>.
?reaction <http://bio2rdf.org/kegg_vocabulary:enzyme>  ?enzyme.
?drug    <http://bio2rdf.org/drugbank_vocabulary:category> ?category.
?drug <http://purl.org/dc/terms:description> ?desc.
?drug <http://bio2rdf.org/drugbank_vocabulary:x-kegg> ?cpd }
Q8: Find out side effects and pathways of all the anticonvulsants medicine.
select * where{
?s1 <http://bio2rdf.org/drugbank_vocabulary:category>
<http://bio2rdf.org/drugbank_vocabulary:Anticonvulsants>.
?s1 <http://bio2rdf.org/drugbank_vocabulary:affected-organism> ?affected.
?s1 <http://bio2rdf.org/drugbank_vocabulary:x-pubchemcompound> ?cpd.
?drug <http://bio2rdf.org/sider_vocabulary:side-effect> ?side.
?drug <http://bio2rdf.org/sider_vocabulary:pubchem-flat-compound-id> ?cpd.
?side <http://purl.org/dc/terms/title> ?side_effect.
?s2 <http://bio2rdf.org/kegg_vocabulary:x-pubchem.compound> ?cpd.
?s2 <http://bio2rdf.org/kegg_vocabulary:pathway>  ?pathway }
```