Adaptive Join Operator for Federated Queries over Linked Data Endpoints

Damla Oguz[1,2,3](✉), Shaoyi Yin[2], Abdelkader Hameurlain[2], Belgin Ergenc[1], and Oguz Dikenelli[3]

[1] Department of Computer Engineering, Izmir Institute of Technology, Izmir, Turkey
{damlaoguz,belginergenc}@iyte.edu.tr

[2] IRIT Laboratory, Paul Sabatier University, Toulouse, France
{yin,hameurlain}@irit.fr

[3] Department of Computer Engineering, Ege University, Izmir, Turkey
oguz.dikenelli@ege.edu.tr

Abstract. Traditional static query optimization is not adequate for query federation over linked data endpoints due to unpredictable data arrival rates and missing statistics. In this paper, we propose an adaptive join operator for federated query processing which can change the join method during the execution. Our approach always begins with symmetric hash join in order to produce the first result tuple as soon as possible and changes the join method as bind join when it estimates that bind join is more efficient than symmetric hash join for the rest of the process. We compare our approach with symmetric hash join and bind join. Performance evaluation shows that our approach provides optimal response time and has the adaptation ability to the different data arrival rates.

Keywords: Distributed query processing · Linked data · Query federation · Join methods · Adaptive query optimization

1 Introduction

Linked data, which is a way of publishing and connecting structured data on the web, provides connected distributed data across the web. In other words, linked data creates a global data space on the web. Link traversal and query federation are the two approaches for querying this space on the distributed data sources. Link traversal [1] finds the related data sources during the query execution whereas query federation [2] selects the related data sources before the execution. In short, link traversal has the disadvantage of not guaranteeing complete results. For this reason, we turn our attention to query federation.

Query federation divides the query into subqueries and distributes them to the SPARQL endpoints of the selected data sources. The intermediate results from the data sources are aggregated and the final results are generated. These processes are employed via a federated query engine whose objective is to minimize the response time and the completion time. Response time refers to the

J. Pokorný et al. (Eds.): ADBIS 2016, LNCS 9809, pp. 275–290, 2016.
DOI: 10.1007/978-3-319-44039-2_19

time to receive the first result tuple whereas completion time refers to the time to receive all the result tuples. Response time and completion time include communication time, I/O time and CPU time. Since the communication time dominates the other costs, the main objective can be stated as to minimize the communication cost. Schwarte et al. [3] use heuristics in query optimization whereas Quilitz and Leser [4], Gortlitz and Staab [5] and Wang et al. [6] concentrate on static optimization which produces an execution plan at query compilation time and uses statistics to estimate the cardinality of the intermediate results. However, federated query processing is done on the distributed data sources on the web which causes unpredictable data arrival rates. In addition, most of the statistics are missing or unreliable. For these reasons, we think that adaptive query optimization [7] is a need in this unpredictable environment. There are only two engines ANAPSID [8] and ADERIS [9,10] which consider adaptive query optimization for query federation. Acosta et al. [8] propose a non-blocking join method based on symmetric hash join [11] and Xjoin [12] whereas Lynden et al. [10] propose a cost model for dynamically changing the join order. To the best of our knowledge, there is not any study that exploits an adaptive join operator that aims to reduce both response time and completion time.

As mentioned earlier, communication time has the highest effect on overall cost and therefore join method has an important role in query optimization. However, there is not any study which changes the join method during the execution according to the data arrival rates. In this study, we propose an adaptive join operator for federated query processing on linked data which can change the join method during the execution by using adaptive query optimization. Performance evaluation shows that our proposal has both the advantage of optimal response time and the adaptation ability to the different data arrival rates. By this adaptation ability, completion time is minimized as well.

The rest of the paper is organized as follows: Sect. 2 introduces our approach for both single join queries and multi-join queries. Section 3 points out the results of our performance evaluation. Section 4 presents a brief survey of query optimization methods in relational databases and query federation over linked data. Finally, we conclude the paper and give remarks for the future work in Sect. 5.

2 Proposed Adaptive Join Operator

Bind join [13] passes the bindings of the intermediate results of the outer relation to the inner relation in order to filter the result set and is substantially efficient when the intermediate results are small. Symmetric hash join [11] maintains a hash table for each relation. Thus, symmetric hash join is a non-blocking join method which produces the first result tuple as early as possible. Equations 1 and 2 [4] are the cost functions of bind join and symmetric hash join respectively, where $R1$ and $R2$ are relations, $card(R)$ is the number of tuples in R, c_t is the transfer cost for receiving one result tuple, and c_r is the transfer cost for sending a SPARQL query. $R2'$ is the relation with the bindings of $R1$. $card(R2')$ is equal to $card(R1 \bowtie R2)$ when we assume that the common attribute values are unique.

Equation 2 is used for nested loop join in [4]. However, the cost functions of nested loop join and symmetric hash join are the same when only communication time is considered.

$$cost(R1 \bowtie_{BJ} R2) = card(R1) \cdot c_t + card(R1) \cdot c_r + card(R2') \cdot c_t \tag{1}$$

$$cost(R1 \bowtie_{SHJ} R2) = card(R1) \cdot c_t + card(R2) \cdot c_t + 2 \cdot c_r \tag{2}$$

Deciding the join method by using a cost model before the query execution has some problems. The join cardinality, $card(R1 \bowtie R2)$, and the data arrival rates of relations are unknown before the query execution. Using bind join can cause response time problem if the data arrival rate of the first relation is slow. On the other hand, symmetric hash join can produce the first result tuple as soon as there is a match between $R1$ and $R2$, without waiting for all tuples of $R1$ to arrive. However, if $R2$ is very large while join cardinality is low, the query completion time of symmetric hash join can be longer than the completion time of bind join. We notice that, the data arrival rates of relations are known after a short time of execution. So, the remaining completion time can be estimated. For these reasons, we propose to set the join method as symmetric hash join in the beginning and to use cost functions after having information about the data arrival rates of endpoints. We decide whether to change the join method as bind join according to the cost estimations. In order to learn the cardinalities, we send count queries in the beginning of the execution. As mentioned before, the communication time dominates the I/O time and CPU time. So the cost of count queries is negligible. In brief, our approach is based on the idea of changing the join method during the query execution according to the data arrival rates.

2.1 Adaptive Join Operator for Single Join Queries

Adaptive join operator for single join queries is depicted in Algorithm 1. Firstly, we send count queries to the endpoints of datasets $R1$ and $R2$ in order to learn their cardinalities. We always begin with symmetric hash join. During the execution, if all the tuples from one dataset arrive and the tuples from the other dataset continue to arrive, we estimate the remaining time of continuing with symmetric hash join and switching to bind join. We decide the join method according to these cost estimations. If we switch to bind join, we emit the duplicate results of symmetric hash join and bind join. The join cardinality estimation formula and the remaining time estimation formulas will be presented in the following subsections.

Join Cardinality and Remaining Time Estimations. In this subsection, we introduce our join cardinality estimation formula and remaining time estimation formulas for symmetric hash join and bind join. We use the estimated join cardinality in order to estimate the remaining times. Equation 3 is our join cardinality estimation formula where $|R_i \bowtie R_{j\_arrived}|$ is the cardinality of $R_i \bowtie R_{j\_arrived}$, $|R_j|$ is the cardinality of R_j, and $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j.

Algorithm 1. Adaptive Join Operator for Single Join Queries

```
1  $|R1| \longleftarrow$ cardinality of $R1$ received from the COUNT query
2  $|R2| \longleftarrow$ cardinality of $R2$ received from the COUNT query
3  $|R1_{arrived}| \longleftarrow$ cardinality of arrived $R1$ tuples
4  $|R2_{arrived}| \longleftarrow$ cardinality of arrived $R2$ tuples
5  Set JOIN method as Symmetric Hash Join (SHJ)
6  while $(|R1_{arrived}| < |R1|$ or $|R2_{arrived}| < |R2|)$ do
7      if $(|R1_{arrived}| == |R1|$ and $|R2_{arrived}| < |R2|$ or
       $|R2_{arrived}| == |R2|$ and $|R1_{arrived}| < |R1|)$ then
8          $ERT_{SHJ} \longleftarrow$ estimated remaining time if continued using SHJ
9          $ERT_{BJ} \longleftarrow$ estimated remaining time if switched to Bind Join (BJ)
10         if $(ERT_{SHJ} > ERT_{BJ})$ then
11             Set JOIN method as BJ
12             Emit the duplicate results of SHJ and BJ
13         end
14     end
15 end
```

We use this formula in order to calculate the estimated cardinality of $R_i \bowtie R_j$ when all the tuples of R_i arrive. We expect that there is a directional proportion between the join cardinality and number of tuples of R_j.

$$JoinCardinality_{estimation} = \frac{|R_i \bowtie R_{j\_arrived}| \cdot |R_j|}{|R_{j\_arrived}|} \tag{3}$$

As stated earlier, when all the tuples of R_i arrive, the algorithm estimates the remaining time if adaptive join operator continues with symmetric hash join and the remaining time if it changes the join method as bind join. We have an idea about the data arrival rate of R_j during the execution, so the estimation is possible. Equation 4 shows the estimated remaining time if adaptive join operator continues with symmetric hash join where ERT_{SHJ} is the estimated remaining time if it continues with symmetric hash join, $|R_j|$ is the cardinality of R_j, $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j, and $t_{Rj\_arrived}$ is the time for $|R_{j\_arrived}|$ tuples to arrive.

$$ERT_{SHJ} = \frac{(|R_j| - |R_{j\_arrived}|) \cdot t_{Rj\_arrived}}{|R_{j\_arrived}|} \tag{4}$$

Equation 5 shows the estimated remaining time, ERT_{BJ}, if the algorithm switches to bind join where $|R_i|$ is the cardinality of R_i, t_{SQ} is the time for sending one query to a SPARQL endpoint ($\approx \frac{t_{Rj\_arrived}}{|R_{j\_arrived}|}$), $|JoinCardinality_{estimation}|$ is the estimated cardinality of $R_i \bowtie R_j$, $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j, and $t_{Rj\_arrived}$ is the time for $|R_{j\_arrived}|$ tuples to arrive. The estimated remaining time for bind join includes sending all tuples of R_i to the endpoint of R_j and the retrieving time of $R_i \bowtie R_j$ from the endpoint of R_j.

$$ERT_{BJ} = (|R_i| \cdot t_{SQ}) + \frac{|JoinCardinality_{estimation}| \cdot t_{Rj\_arrived}}{|R_{j\_arrived}|} \quad (5)$$

2.2 Adaptive Join Operator for Multi-join Queries

Different from the single join queries, we use multi-way symmetric hash join [14] in the beginning. The algorithm for multi-join queries is depicted in Algorithm 2. When all tuples from a relation arrive, called R_i, the algorithm estimates the remaining time if adaptive join operator switches to bind join for each relation which has a common attribute with R_i. The algorithm chooses the relation with minimum estimated bind join cost, called R_j, and compares the estimated remaining time if it changes the join method as bind join for $R_i \bowtie R_j$ with the estimated remaining time if the operator continues with multi-way symmetric hash join for all relations. The above procedure is repeated every time a relation is completely received.

Algorithm 2. Adaptive Join Operator for Multi-join Queries

```
1  $S \longleftarrow \{R_1, R_2, R_3, ..., R_n\}$
2  $MIN\_ERT_{BJ} \longleftarrow \infty$
3  $BJ\_Candidate \longleftarrow \Phi$
4  Start $MSHJ(S)$
5  while ($S$ is not empty) do
6      if (all the tuples of $R_i$ arrive) then
7          $ERT_{MSHJ} \longleftarrow ERT$ if continued with MSHJ
8          foreach $R_j$ having a common attribute with $R_i$ do
9              $ERT_{BJ\_R_{ij}} \longleftarrow ERT$ if switched to BJ for $R_i$ and $R_j$
10             if ($ERT_{BJ\_R_{ij}} < MIN\_ERT_{BJ}$) then
11                 $MIN\_ERT_{BJ} \longleftarrow ERT_{BJ\_R_{ij}}$
12                 $BJ\_Candidate \longleftarrow \{R_i, R_j\}$
13             end
14         end
15         if ($MIN\_ERT_{BJ} < ERT_{MSHJ}$) then
16             $\acute{R_i} \longleftarrow BJ(R_i, R_j)$
17             $S \longleftarrow S - BJ\_Candidate + \{\acute{R_i}\}$
18             Run $MSHJ(S)$ and eliminate duplicate results
19         end
20     end
21 end
```

Join Cardinality Estimation and Remaining Time Estimations. We use the same formula to calculate the join cardinality estimation for single join queries and multi-join queries. Thus, we use Eq. 3 for join cardinality estimation for multi-join queries as well. We need this estimation in order to calculate the estimated remaining time if adaptive join operator switches to bind join or if the algorithm continues with multi-way symmetric hash join.

Equation 6 shows the estimated remaining time if adaptive join operator uses bind join for R_i and R_j, and uses multi-way symmetric hash join for the other relations which are involved in the query. $|R_i|$ is the cardinality of R_i, t_{SQ} is the time for sending one query to the SPARQL endpoint containing $R_j(\approx \frac{t_{Rj\_arrived}}{|R_{j\_arrived}|})$, $|R_i \bowtie R_j|$ is the estimated cardinality of $R_i \bowtie R_j$, $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j, $t_{Rj\_arrived}$ is the time for $|R_j\_arrived|$ tuples to arrive, ERT_{rest} is the estimated remaining time for the rest of other relations to arrive and it is calculated by using Eq. 7, where $k \in (1, ..., n)$ and $k \neq i$ and $k \neq j$. Lastly, Eq. 7 shows the estimated remaining time if adaptive join operator continues with multi-way symmetric hash join. Completion time is equal to the maximum completion time of the relations which compose the query.

$$ERT_{BJ\_R_{ij}} = max\big((|R_i| \cdot t_{SQ} + \frac{|R_i \bowtie R_j| \cdot t_{Rj\_arrived}}{|R_{j\_arrived}|}); ERT_{rest}\big) \tag{6}$$

$$ERT_{MSHJ} = max\big(\frac{(|R_k| - |R_{k\_arrived}|) \cdot t_{Rk\_arrived}}{|R_{k\_arrived}|}\big) \text{ where } k \in (1, ..., n) \tag{7}$$

3 Performance Evaluation

In this section, we present the evaluation results on the performances of symmetric hash join/multi-way symmetric hash join, bind join and adaptive join operator for single join queries and for multi-join queries. The reason of comparing our proposal with symmetric hash join and bind join is as follows. Bind join is the most popular join method in query federation engines and symmetric hash join provides efficient response time by being a non-blocking join method [15]. As stated in the previous sections, the goal of the query optimization in query federation is to minimize the response time and the completion time. For this reason, we use response time and completion time as the evaluation metrics. Query cost in distributed environment is mainly defined by communication cost. In order to simulate the real network conditions and consider only the communication cost, we conducted our experiments in the network simulator *ns-3*[1].

We assume that the size of all queries is the same and each result tuple is considered to have the same size, as well. Each query size is accepted as 500 bytes whereas each result tuple size is employed as 250 bytes. Each count query size is assumed as 750 bytes and the message size is set to 100 tuples. Each selectivity factor is $0.5/\big(max(\text{cardinality of } R1, \text{cardinality of } R2)\big)$ [16]. We set the low, medium and high cardinality as 1000 tuples, 5000 tuples and 10000 tuples respectively.

[1] https://www.nsnam.org/.

3.1 Performance Evaluation for Single Join Queries

In this subsection, we compare adaptive join operator (AJO) with symmetric hash join (SHJ) and bind join (BJ) in two cases. We aim to show the impact of data sizes in the first case whereas we focus on the effect of different data arrival rates in the second case.

Impact of Data Sizes. The behaviors of the SHJ, BJ and AJO are analyzed when the data arrival rates of both endpoints are fixed to 0.5 Mbps and the delays to 10 ms while the data sizes of $R1$ and $R2$ are changed. In order to analyze all conditions, we evaluated the response time and the completion time when the data sizes of $R1$ and $R2$ are low-low (LL); low-medium (LM); low-high (LH); medium-low (ML); medium-medium (MM); medium-high (MH); high-low (HL); high-medium (HM) and high-high (HH) respectively.

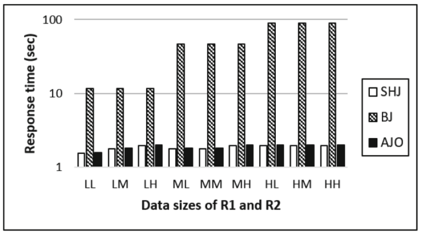
As Fig. 1a shows, BJ has the worst response time for all conditions whereas SHJ and AJO behave similar to each other. As the data sizes of $R1$ increases, the response time of BJ increases as well due to waiting for the arrival of all results of $R1$ and sending them to the endpoint of $R2$. On the other hand, SHJ and AJO can generate the first result tuple as soon as there is a match between $R1$ and $R2$, without waiting for all tuples of $R1$ to arrive. As shown in Fig. 1b, the completion time of BJ is shorter than others when the cardinality of $R1$ is low and the cardinality of $R2$ is medium or high. On the other hand, SHJ and AJO perform better than BJ in seven of nine conditions. AJO's completion time is the best when the cardinality of $R1$ is medium or high and the cardinality of $R2$ is low. Also, AJO's completion time is faster than SHJ's when the cardinality of $R1$ is low and the cardinality of $R2$ is medium or high.

The speedup[2] values between AJO and SHJ can be seen in Fig. 1c. Although they have almost the same response time for all cases, the completion time of AJO is 3 times as fast compared to SHJ when one of the relation's cardinality is high and the other one's is low. As shown in Fig. 1d, AJO can provide speedup in response time from 5.9 times to 45.5 times compared to BJ. AJO also provides speedup in completion time up to 6 times except two cases.

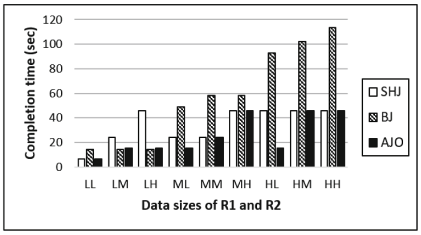
Impact of Data Arrival Rates. In this case, we fixed the data arrival rate of $R1$ to 2 Mbps and changed the data arrival rate of $R2$. We conducted the simulations for two different cardinality options: (i) low cardinality of $R1$ and high cardinality of $R2$; (ii) high cardinality of $R1$ and low cardinality of $R2$.

Low Cardinality of R1 and High Cardinality of R2. As Fig. 2a shows, the response time of BJ is always longer than SHJ and AJO. The gap between the response times of BJ and the others increases when the data arrival rate of $R2$ gets slower. As shown in Fig. 2b, the completion time of BJ is better than

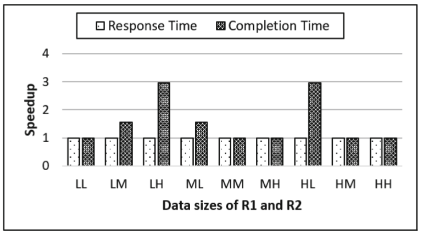
[2] Speedup of x compared to y (response time) = response time of y / response time of x
Speedup of x compared to y (completion time) = completion time of y / completion time of x.



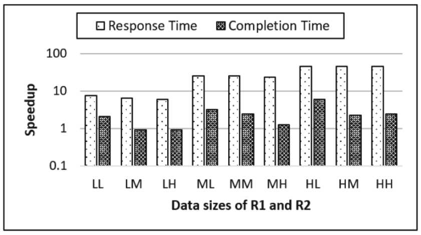
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to SHJ



(d) Speedup of AJO compared to BJ

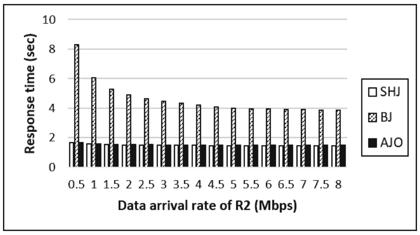
Fig. 1. Data arrival rates of $R1$ and $R2$ are fixed

others for all conditions because the first relation's cardinality is low. As the data arrival rate of the second relation gets faster, the difference between BJ and others decreases. The completion time of AJO is always faster than SHJ.

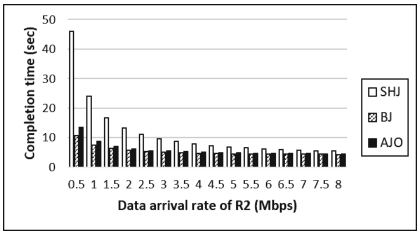
As shown in Fig. 2c, compared to SHJ, AJO has almost the same response time, however it can provide speedup in completion time up to 3.4 times. Although the speedup decreases while the second relation's data arrival rate increases, we expect it to be nearly 1 in the worst case. Compared to BJ, AJO degrades completion time up to 0.8 times, however it can improve the response time up to 4.9 times, as shown in Fig. 2d.

High Cardinality of R1 and Low Cardinality of R2. The results observed from Fig. 3a are similar to the results in Fig. 2a. Since the cardinality of the first relation is high in this case, response time of BJ is dramatically longer than SHJ and AJO. The response times of SHJ and AJO are nearly the same.

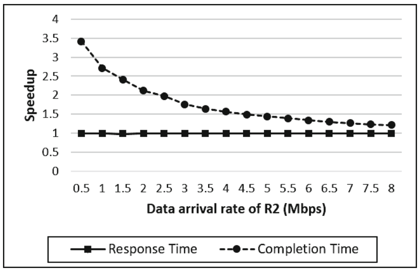
As shown in Fig. 3b, the completion times of SHJ and AJO are shorter than the completion time of BJ in all of the conditions because the first relation's cardinality is high. AJO performs better than SHJ in all the cases. Compared to SHJ, AJO has almost the same response time, however the speedup in completion time varies from 1.4 times to 2.2 times as shown in Fig. 3c. Compared to BJ, AJO improves both the response time and the completion time as illustrated in Fig. 3d. The speedup in response time increases from 11 times to 34.3 times while the speedup in completion time varies from 2.8 to 6.2 times.



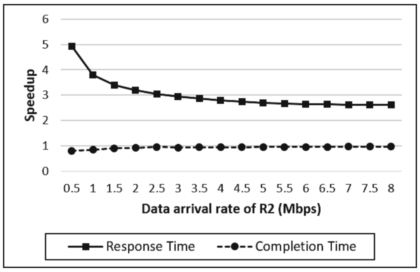
(a) Response time



(b) Completion time

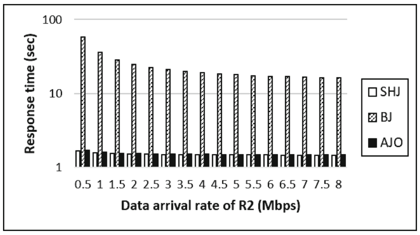


(c) Speedup of AJO compared to SHJ

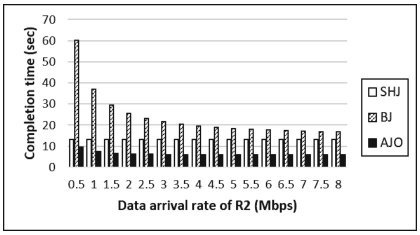


(d) Speedup of AJO compared to BJ

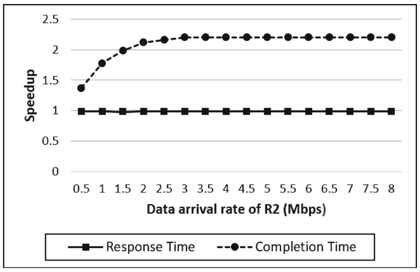
Fig. 2. Data sizes of $R1$ and $R2$ are fixed with $card(R1) \ll card(R2)$



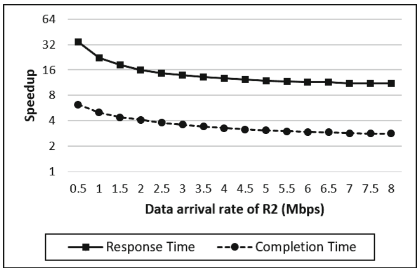
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to SHJ



(d) Speedup of AJO compared to BJ

Fig. 3. Data sizes of $R1$ and $R2$ are fixed with $card(R1) \gg card(R2)$

3.2 Performance Evaluation for Multi-Join Queries

In this subsection, we compare AJO with multi-way symmetric hash join (MSHJ) and BJ when there are three relations in the query. A query example that we use in our experiments is shown below. $R1$ ($service1$) and $R2$ ($service2$) have a common attribute, ?$student$, $R2$ and $R3$ ($service3$) have a common attribute, ?$course$.

```
SELECT ?student ?level ?course ?instructorName WHERE {
    SERVICE <:service1> { ?student :name :studentName .
                          ?student :level ?level . }
    SERVICE <:service2> { ?student :enroll ?course . }
    SERVICE <:service3> { ?course :instructor ?instructorName . }
}
```

Impact of Data Sizes. We fixed the data arrival rates of all relations to 0.5 Mbps and the delays to 10 ms. We conducted our experiments when the data sizes of $R1$, $R2$, $R3$ are low-low-high (LLH); low-medium-high (LMH); and low-high-high (LHH).

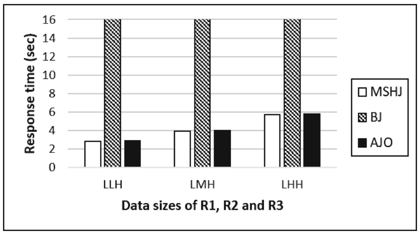
As Fig. 4a shows, the response times of MSHJ and AJO are almost the same whereas BJ's response time is substantially slower. BJ's completion time is the fastest as illustrated in Fig. 4b, because the first relation's cardinality is low. However, AJO's completion time is much better than MSHJ and close to BJ's. BJ's both response time and completion time would increase, if the first relation's cardinality were medium or high.

As shown in Fig. 4c, compared to MSHJ, AJO has almost the same response time, however it can provide speedup in completion time up to 2.3 times. Speedup values between AJO and BJ can be seen in Fig. 4d Compared to BJ, AJO degrades completion time up to 0.85 times, however it can improve the response time up to 5.75 times.

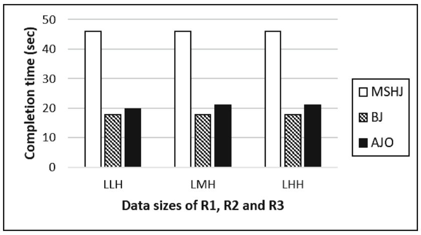
Impact of Data Arrival Rates. In order to show the impact of data arrival rates on MSHJ, BJ and AJO, we fixed the data arrival rates of $R1$ and $R3$ to 2 Mbps and changed the data arrival rate of $R2$. We conducted the simulations for two different cardinality options: (i) low cardinality of $R1$, high cardinality of $R2$, and low cardinality of $R3$ (LHL); (ii) low cardinality of $R1$, high cardinality of $R2$ and $R3$ (LHH).

Low Cardinality of R1, High Cardinality of R2, Low Cardinality of R3. Figure 5a. shows that BJ performs worser response time than MSHJ and AJO in this case as well. As can be seen from Fig. 5b, BJ's completion time is faster than MSHJ because the first relation's cardinality is low. On the other hand, AJO performs the best in seven of nine cases due to having the adaptation ability.

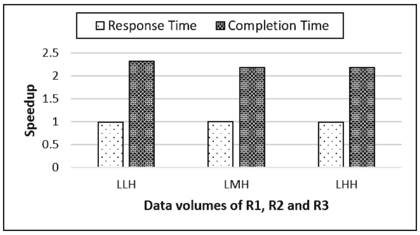
Compared to MSHJ, AJO has almost the same response time but it can provide speedup in completion time up to 3.4 times as shown in Fig. 5c. Compared



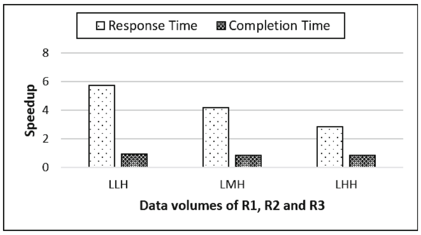
(a) Response time



(b) Completion time

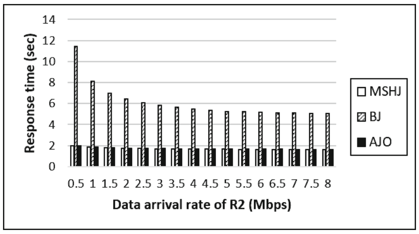


(c) Speedup of AJO compared to MSHJ

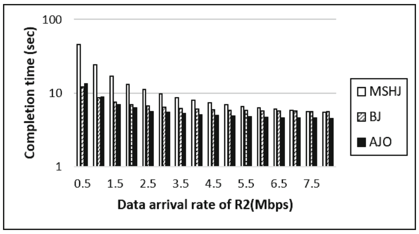


(d) Speedup of AJO compared to BJ

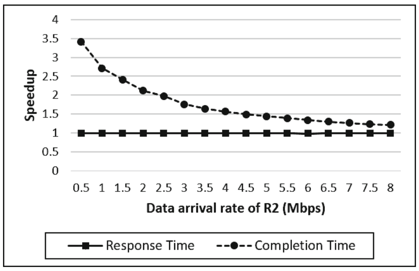
Fig. 4. Data arrival rates of $R1$, $R2$ and $R3$ are fixed



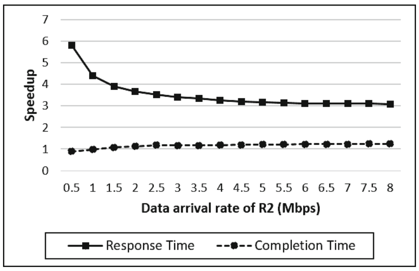
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to MSHJ



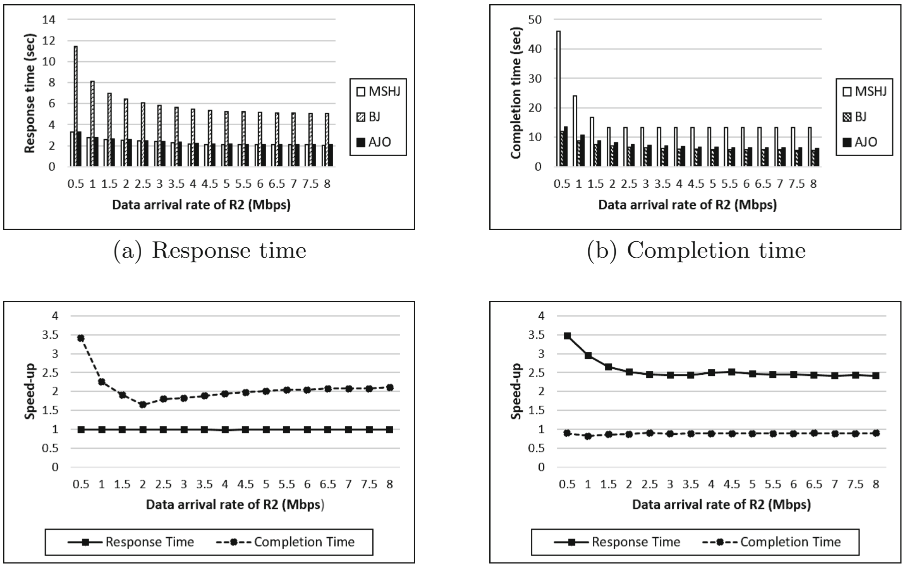
(d) Speedup of AJO compared to BJ

Fig. 5. Data sizes of $R1$, $R2$ and $R3$ are fixed with $card(R1) = card(R3) \ll card(R2)$

to BJ, AJO can improve the response time and the completion time up to 5.8 times and 1.2 times respectively as illustrated in Fig. 5d.

Low Cardinality of R1, High Cardinality of R2, High Cardinality of R3. The results observed from Fig. 6a are similar to the results in Fig. 5a. BJ has the worst response time again, whereas MSHJ and AJO have almost the same response time. However, as shown in Fig. 5b, BJ's completion time is better than MSHJ's completion time which has the disadvantage of waiting all the tuples of $R2$ and $R3$. On the other hand, AJO performs much better than MSHJ. Its completion time is close to BJ's completion time because it can change the join method when it decides that is more efficient.

The speedup values between AJO and MSHJ can be seen from Fig. 6c. Compared to MSHJ, AJO has almost the same response time but it can provide speedup in completion time up to 3.4 times. Compared to BJ, AJO degrades the completion time up to 0.8 times, however it can improve the response time up to 3.5 times as shown in Fig. 6d.



(a) Response time

(b) Completion time

(c) Speedup of AJO compared to MSHJ

(d) Speedup of AJO compared to BJ

Fig. 6. Data sizes of $R1$, $R2$ and $R3$ are fixed with $card(R1) \ll card(R2) = card(R3)$

4 Related Work

Adaptive query optimization [7] responds to the unforeseen variations of run-time environment to provide a better response time or more efficient CPU utilization. In our concept, the run-time environment is on the web and the main objective is to minimize the response time and the completion time. Thus, adaptive query optimization is a need to manage the changing conditions of the web. Although, adaptive query optimization is a new research area for linked data, it has been studied in detail in relational databases. Evolutionary methods which provide inter-operator adaptation, focus on generating plans that can be switched during execution according to delays or estimation errors. Query scrambling [17], mid-query re-optimization [18], Tukwilla/ECA rules [19], progressive query optimization [20–22] and proactive re-optimization [23] are some known examples of evolutionary methods. On the other hand, revolutionary methods provide intra-operator adaptation. First group of intra-operator methods are adaptive operators like double hash join [19], XJoin [12] and mobile join [24,25], where the operator itself is able to adapt its way of execution according to variations encountered during its execution. Second group of intra-operator methods optimize the query processing in tuple level [26–31].

Another work for distributed database environment is also quite relevant to our work. Khan et al. [32] propose an adaptive probing mechanism to have statistics about the data and choose the optimal execution plan during query execution. Compared to our work, the probe phase of their method delays the response time since the first result tuple is generated before the end of probing and decision for adaptability.

When we look at the adaptive methods of query federation engines on linked data, we see only two adaptive methods, intra-operator adaptivity of ANAPSID [8] and inter-operator adaptivity of ADERIS [10]. ANAPSID focuses on the problem of bursty data traffic and endpoint unavailability. In order to overcome these problems, ANAPSID implements a non-blocking join method which is based on symmetric hash join [11] and XJoin [12]. The proposed method continues to produce new results when one of the endpoints becomes blocked. ADERIS generates predicate tables for each predicate which cover the related subjects and objects of that predicate. The first version of ADERIS [9] joins two predicate tables as they become complete while the other predicate tables are being generated. In the second version, Lynden et al. [10] propose an adaptive cost model to determine the join order. In other words, ADERIS uses adaptive query optimization by changing the join order during the execution. In addition to these studies, Basca and Bernstein [33] propose a technique which gathers statistics on the fly before query execution. It produces only the first k results. In addition, Verborgh et al. [34] and Acosta et al. [35] focus on adaptive query optimization for triple pattern fragments. However, triple pattern fragments are beyond the scope of this paper.

Intra-operator adaptivity of ANAPSID and inter-operator adaptivity of ADERIS have showed that adaptive query optimization is well suited to unpredictable characteristics of linked data environment. Although they provide adaptive solutions for query federation, none of them use adaptive query optimization in order to change the join method during the execution according to the data arrival rates to minimize both response time and completion time at the same time.

5 Conclusion

In this paper, we presented an adaptive join operator for single join queries and multi-join queries which aims to minimize both response time and completion time. It begins with symmetric hash join in order to provide optimal response time and changes the join method to bind join when it decides that bind join is more efficient than symmetric hash join for the rest of the query.

The results of the performance evaluation have shown the efficiency of the proposed adaptive join operator. It has almost the same response time with symmetric hash join and multi-way symmetric hash join, but it provides faster completion time. Compared to bind join, adaptive join operator performs substantially better with respect to the response time and can also improve the completion time. Bind join can provide slightly better completion time than adaptive join operator when the first relation's cardinality is low.

In conclusion, adaptive join operator provides both optimal response time and completion time for single join queries and multi-join queries. It performs quite well both in fixed and different data arrival rates. We plan to make experiments with more joins. We are also motivated to consider the case where a relation is distributed over multiple sources.

Acknowledgment. This work is partially supported by The Scientific and Technological Research Council of Turkey (TUBITAK).

References

1. 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)
2. Görlitz, O., Staab, S.: Federated data management and query optimization for linked open data. In: Vakali, A., Jain, L.C. (eds.) New Directions in Web Data Management 1. SCI, vol. 331, pp. 109–137. Springer, Heidelberg (2011)
3. 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)
4. 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)
5. Görlitz, O., Staab, S.: SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In: Proceedings of the Second International Workshop on Consuming Linked Data (COLD 2011), CEUR Workshop Proceedings, Bonn, Germany, 23 October 2011, vol. 782 (2011). http://CEUR-WS.org
6. Wang, X., Tiropanis, T., Davis, H.C.: LHD: optimising linked data query processing using parallelisation. In: Proceedings of the WWW 2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May 2013 (2013)
7. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007)

8. Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: an adaptive query processing engine for SPARQL endpoints. 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. 18–34. Springer, Heidelberg (2011)
9. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: Adaptive integration of distributed semantic web data. In: Kikuchi, S., Sachdeva, S., Bhalla, S. (eds.) DNIS 2010. LNCS, vol. 5999, pp. 174–193. Springer, Heidelberg (2010)
10. Lynden, S., Kojima, I., Matono, A., Tanimura, Y.: ADERIS: an adaptive query processor for joining federated SPARQL endpoints. In: Meersman, R., et al. (eds.) OTM 2011, Part II. LNCS, vol. 7045, pp. 808–817. Springer, Heidelberg (2011)
11. Wilschut, A.N., Apers, P.M.G.: Dataflow query execution in a parallel main-memory environment. In: Proceedings of the First International Conference on Parallel and Distributed Information Systems. PDIS 1991, pp. 68–77. IEEE Computer Society Press (1991)
12. Urhan, T., Franklin, M.J.: XJoin: a reactively-scheduled pipelined join operator. IEEE Data Eng. Bull. **23**(2), 27–33 (2000)
13. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing queries across diverse data sources. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997, pp. 276–285. Morgan Kaufmann Publishers Inc. (1997)
14. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 285–296. VLDB Endowment (2003)
15. Oguz, D., Ergenc, B., Yin, S., Dikenelli, O., Hameurlain, A.: Federated query processing on linked data: a qualitative survey and open challenges. Knowl. Eng. Rev. **30**(5), 545–563 (2015)
16. Shekita, E.J., Young, H.C., Tan, K.L.: Multi-join optimization for symmetric multiprocessors. In: Proceedings of the 19th International Conference on Very Large Data Bases, VLDB 1993, pp. 479–492. Morgan Kaufmann Publishers Inc. (1993)
17. Amsaleg, L., Franklin, M.J., Tomasic, A.: Dynamic query operator scheduling for wide-area remote access. Distrib. Parallel Databases **6**(3), 217–246 (1998)
18. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. SIGMOD Rec. **27**(2), 106–117 (1998)
19. Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration. SIGMOD Rec. **28**(2), 299–310 (1999)
20. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD 2004, pp. 659–670. ACM (2004)
21. Kache, H., Han, W.S., Markl, V., Raman, V., Ewen, S.: POP/FED: progressive query optimization for federated queries in DB2. In: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB 2006, pp. 1175–1178. VLDB Endowment (2006)
22. Han, W.S., Ng, J., Markl, V., Kache, H., Kandil, M.: Progressive optimization in a shared-nothing parallel database. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, pp. 809–820. ACM (2007)
23. Babu, S., Bizarro, P., DeWitt, D.: Proactive re-optimization. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD 2005, pp. 107–118. ACM (2005)

24. Arcangeli, J., Hameurlain, A., Migeon, F., Morvan, F.: Mobile agent based self-adaptive join for wide-area distributed query processing. J. Database Manag. (JDM) **15**(4), 25–44 (2004)
25. Ozakar, B., Morvan, F., Hameurlain, A.: Mobile join operators for restricted sources. Mob. Inf. Syst. **1**(3), 167–184 (2005)
26. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. SIGMOD Rec. **29**(2), 261–272 (2000)
27. Raman, V., Deshpande, A., Hellerstein, J.M.: Using state modules for adaptive query processing. In: Proceedings of the 19th International Conference on Data Engineering, 5–8 March 2003, Bangalore, India, pp. 353–364 (2003)
28. Deshpande, A.: An initial study of overheads of eddies. SIGMOD Rec. **33**(1), 44–49 (2004)
29. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, vol. 30, pp. 948–959. VLDB Endowment (2004)
30. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based routing: different plans for different data. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 757–768. VLDB Endowment (2005)
31. Zhou, Y., Ooi, B.C., Tan, K., Tok, W.H.: An adaptable distributed query processing architecture. Data Knowl. Eng. **53**(3), 283–309 (2005)
32. Khan, L., McLeod, D., Shahabi, C.: An adaptive probe-based technique to optimize join queries in distributed internet databases. J. Database Manag. **12**(4), 3–14 (2001)
33. Basca, C., Bernstein, A.: Avalanche: putting the spirit of the web back into semantic web querying. In: Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts, Shanghai, China, 9 November 2010 (2010)
34. Verborgh, R., et al.: Querying datasets on the web with high availability. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 180–196. Springer, Heidelberg (2014)
35. Acosta, M., Vidal, M.E.: Networks of linked data eddies: an adaptive web query processing engine for RDF data. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 111–127. Springer International Publishing, Heidelberg (2015)