International Journal of Data Warehousing and Mining
Volume 13 • Issue 3 • July-September 2017

Extended Adaptive Join Operator with Bind-Bloom Join for Federated SPARQL Queries

Damla Oguz, Institute of Research in Computer Science of Toulouse (IRIT), Paul Sabatier University, Toulouse, France & Department of Computer Engineering, Izmir Institute of Technology, Izmir, Turkey & Department of Computer Engineering, Ege University, Izmir, Turkey

Shaoyi Yin, Institute of Research in Computer Science of Toulouse (IRIT), Paul Sabatier University, Toulouse, France

Belgin Ergenç, Department of Computer Engineering, Izmir Institute of Technology, Izmir, Turkey

Abdelkader Hameurlain, Institute of Research in Computer Science of Toulouse (IRIT), Paul Sabatier University, Toulouse, France

Oguz Dikenelli, Department of Computer Engineering, Ege University, Izmir, Turkey

ABSTRACT

The goal of query optimization in query federation over linked data is to minimize the response time and the completion time. Communication time has the highest impact on them both. Static query optimization can end up with inefficient execution plans due to unpredictable data arrival rates and missing statistics. This study is an extension of adaptive join operator which always begins with symmetric hash join to minimize the response time, and can change the join method to bind join to minimize the completion time. The authors extend adaptive join operator with bind-bloom join to further reduce the communication time and, consequently, to minimize the completion time. They compare the new operator with symmetric hash join, bind join, bind-bloom join, and adaptive join operator with respect to the response time and the completion time. Performance evaluation shows that the extended operator provides optimal response time and further reduces the completion time. Moreover, it has the adaptation ability to different data arrival rates.

KEYWORDS

Adaptive Query Optimization, Bloom Filter, Distributed Query Processing, Join Methods, Linked Data, Query Federation

1. INTRODUCTION

As the increase in the number of data sources on linked data, a distributed data space on the web is generated. This huge global data space can be automatically queried by using two approaches called link traversal (Hartig, Bizer, & Freytag, 2009) and query federation (Görlitz & Staab, 2011a). The first approach is based on discovering potentially relevant data by following the links between them. In other words, it finds the related data sources during the query execution. The second approach, query federation, divides the query into subqueries and distributes them to the SPARQL endpoints of the relevant data sources. The intermediate results from the data sources are aggregated and the final

DOI: 10.4018/IJDWM.2017070103

results are generated. Although both approaches have the advantage of providing up-to-date results, link traversal cannot guarantee finding all results because the relevant data sources change according to the starting point. For this reason, we focus on the query federation approach.

The objective of engines in query federation is to minimize both the response time and the completion time. Response time is the time to generate the first result tuple, whereas completion time is the time to provide all result tuples. Response time and completion time include communication time, I/O time and CPU time. Since the communication time dominates other costs, the main objective of the federated query engines can be stated as to minimize the communication cost. Static query optimization (Selinger, Astrahan, Chamberlin, Lorie, & Price, 1979) is not adequate for federated queries, because they are executed over the SPARQL endpoints of the selected distributed data sources on the web, and the data arrival rates are unexpected. Moreover, most of the statistics about the data sources are missing or unreliable. These constraints show that adaptive query optimization (Deshpande, Ives, & Raman, 2007) is a necessity for query federation over linked data.

Adaptive query optimization has been studied in detail in relational databases (Babu & Bizarro, 2005; Deshpande et al., 2007; Morvan & Hameurlain, 2009; Gounaris, Tsamoura, & Manolopoulos, 2013). However, it is a new research area for linked data. There are only two engines which consider adaptive query optimization for federated queries over SPARQL endpoints: ANAPSID (Acosta, Vidal, Lampo, Castillo, & Ruckhaus, 2011) and ADERIS (Lynden, Kojima, Matono, & Tanimura, 2010, 2011). The first one proposes a non-blocking join method based on symmetric hash join (Wilschut & Apers, 1991) and Xjoin (Urhan & Franklin, 2000), while the second one uses a cost model for dynamically changing the join order. Other than these, AVALANCHE (Basca & Bernstein, 2010, 2014) collects statistical information about relevant data sources and then generates its execution plan to provide the first k tuples. In addition, there are several studies which concentrate on join ordering for SPARQL queries by using different techniques such as evolutionary algorithms (Oren, Guéret, & Schlobach, 2008; Hogenboom, Milea, Frasincar, & Kaymak, 2009) and ant colony (Hogenboom, Frasincar, & Kaymak, 2013; Kalayci, Kalayci, & Birant, 2015). To the best of our knowledge, adaptive join operator (Oguz, Yin, Hameurlain, Ergenc, & Dikenelli, 2016) is the first study which aims to reduce both the response time and the completion time for query federation over SPARQL endpoints.

As mentioned above, the communication cost is the dominant cost in distributed environments. Bloom filter (Bloom, 1970), which is a space efficient data structure, is widely used in relational databases (Mackert & Lohman, 1986; Mullin, 1990; Michael, Nejdl, Papapetrou, & Siberski, 2007; Ives & Taylor, 2008). It is utilized in different linked data tasks such as identity reasoning (Williams, 2008) and data source selection (Hose & Schenkel, 2012). Bloom filter is also employed to reduce the communication cost in two studies of linked data (Basca & Bernstein, 2014; Groppe, Heinrich, & Werner, 2015).

In this paper, we present an extended version of our previous work (Oguz et al., 2016) in which adaptive join operator is proposed. The new contributions of this paper are as follows: i) We improve our previous proposal with bind-bloom join (Basca & Bernstein, 2014; Groppe et al., 2015) for both single join queries and multi-join queries by including bind-bloom join to the candidate join methods. ii) We present a detailed performance evaluation study which shows the advantage of our new proposal. iii) We extend our related work with new studies and comparison of adaptive query optimization methods in query federation. Our operator uses symmetric hash join in the beginning to minimize the response time, and can change the join method to bind join or bind-bloom join. Bind-bloom join, shortly can be defined as a kind of bind join enhanced with bloom filter in order to minimize the communication time. It is explained in detail in the following section. Performance evaluation shows that the extended operator has both the advantage of optimal response time and the adaptation ability to different data arrival rates in order to minimize the completion time. Moreover, it provides faster completion time than our previous operator in all test cases.

The rest of the paper is organized as follows: Section 2 introduces our approach for both single join queries and multi-join queries. Section 3 presents the results and discussions on performance evaluation. Section 4 covers the related work and Section 5 concludes the paper.

2. PROPOSED EXTENDED ADAPTIVE JOIN OPERATOR

In our previous work (Oguz et al., 2016), we have proposed an adaptive join operator for federated queries over linked data endpoints, called AJO. It always begins with symmetric hash join in order to minimize the response time, and when all the tuples of a relation arrive, it estimates the remaining times for symmetric hash join and bind join in order to minimize the completion time. It changes the join method to bind join if it estimates that it is more efficient than symmetric hash join. In this paper, we propose an extended version of AJO with bind-bloom join in order to further reduce the communication time. In this section, we first explain the principles of symmetric hash join, bind join, bloom filter, and bind-bloom join. Second, we present our proposal for single join queries and multi-join queries.

2.1. Background

Symmetric hash join (Wilschut & Apers, 1991) maintains a hash table for each relation. Thus, it is a non-blocking join method which produces the first result tuple as early as possible. In other words, it is good at response time. Bind join (Haas, Kossmann, Wimmers, & Yang, 1997), which is the most popular join method among the federated query engines (Oguz, Ergenc, Yin, Dikenelli, & Hameurlain, 2015), passes the bindings of the intermediate results of the outer relation to the inner relation to filter the result set. It provides good completion time when the cardinalities of the first relation and the intermediate results are low. Equation 1 and Equation 2 show the cost functions of these join methods that are the variations of the formulas in (Quilitz & Leser, 2008). *R1* and *R2* are relations, *card(R)* is the number of tuples in *R*, c_{t_R} is the transfer cost of *R* for one result tuple, and *R2′* is the relation with the bindings of *R1*:

$$cost\left(R1 \bowtie_{SHJ} R2\right) = card\left(R1\right) \cdot c_{t_{R1}} + card\left(R2\right) \cdot c_{t_{R2}} \tag{1}$$

$$cost\left(R1 \bowtie_{BJ} R2\right) = card\left(R1\right) \cdot c_{t_{R1}} + card\left(R1\right) \cdot c_{t_{R2}} + card\left(R2'\right) \cdot c_{t_{R2}} \tag{2}$$

Bloom filter (Bloom, 1970) is a data structure which represents a set of elements in a bit vector with a low rate of false positives. The idea is to represent a set $S = \{e_1, e_2, e_3, \ldots, e_n\}$ of n elements in a vector v of m bits. Initially all the bits are set to 0. Then, k independent hash functions, $h_1, h_2, \ldots, h_k$, with range $\{1, \ldots, m\}$ are used. For each element $e_i \in S$, the bits at positions $h_1(e_1), h_2(e_1), \ldots, h_k(e_1)$ in v are set to 1. Given a query for e_j, the bits at positions $h_1(e_j), h_2(e_j), \ldots, h_k(e_j)$ are checked. If any of them is 0, certainly e_j is not in the set S. Otherwise, e_j is accepted as a member of set S, although there is a probability that it is not a member (Fan, Cao, Almeida, & Broder, 2000). Independent of the size of the elements, less than 10 bits per element are required for a 1% false positive probability (Bonomi, Mitzenmacher, Panigrahy, Singh, & Varghese, 2006).

We propose to use b bits per each element and k hash functions to minimize the false positive rate (Fan et al., 2000). We propose a custom SPARQL function CheckBloom*(?commonAttribute*, *?bitVector*) which returns true if the positions corresponding to h_1(*?commonAttribute*), h_2(*?commonAttribute*), . . ., h_k(*?commonAttribute*) are set to 1 in bloom filter *?bitVector*. We explain the advantage of using a bloom filter in bind join by using the federated query example in Listing 1. Initially, the first subquery is executed on *:service1* and then the second subquery is executed on *:service2* with the bindings of the first subquery as shown in Listing 2. The intermediate results from *:service1* can be seen from

Table 1. Query size is proportional to the number of intermediate results, and the communication cost increases as the number of intermediate results increases. In order to decrease this cost, bind join can be employed by using a bloom filter as shown in Listing 3. *BloomFilter* is a bit array whose length in bits is equal to multiplication of the number of distinct common attribute values and *b* bits. Since our proposal uses *b* bits per each intermediate result, the size of the bloom filter in bits is equal to multiplication of the number of distinct common attribute values and *b* bits. On the other hand, bind join can be more efficient than bind-bloom join in some cases according to the number of false positives and the size of the result set. For this reason, our proposal estimates the remaining times of bind join and bind-bloom join when the tuples of a relation all arrive.

Listing 1: Federated query example

```
SELECT * WHERE
 {
   SERVICE <:service1> {?student :name :studentName.}
   SERVICE <:service2> {?student :enroll ?course.}
 }
```

Listing 2: Bind query

```
SELECT * WHERE
{
  ?student :enroll ?course.
  FILTER (?student=:student_1 ||
                    ...       ||
          ?student=:student_n)
}
```

Listing 3: Bind query with bloom filter

```
PREFIX ex:<http://irit.fr/bloom/>
SELECT * WHERE
{
   ?student:enroll ?course .
   FILTER (ex:CheckBloom(?student,
               "BloomFilter") )
}
```

Table 1. Intermediate results

Line	Student
1	*student_1*
2	*student_2*
. . .	. . .
n	*student_n*

2.2. Extended Adaptive Join Operator for Single Join Queries

Extended 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 in order to minimize the response time. During the execution, when all the tuples from one dataset arrive and the tuples from the other dataset continue to arrive, we estimate the remaining times of continuing with symmetric hash join, switching to bind join, and switching to bind-bloom join. We decide the join method according to these cost estimations. If we switch to bind join or bind-bloom join, we emit the duplicate results of symmetric hash join with bind join or bind-bloom join. The cardinality estimation formula and the remaining time estimation formulas are presented in the following subsections.

2.2.1. Cardinality and Remaining Time Estimations

Equation 3 shows the cost function of bind join where R_i and R_j are relations, $|R|$ is the number of tuples in R, c_{t_R} is the transfer cost of R for each result tuple, and R_j' is the relation with the bindings of R_i. In order to estimate the remaining times of bind join and bind-bloom join, we need the estimated cardinality of the second relation which is reduced by the bindings of the first relation, namely R_j':

$$cost\left(R_i \bowtie_{BJ} R_j\right) = \left|R_i\right| \cdot c_{t_{Ri}} + \left|R_i\right| \cdot c_{t_{Rj}} + \left|R_j^{'}\right| \cdot c_{t_{Rj}} \tag{3}$$

Algorithm 1. Extended adaptive join operator for single join queries

```
1  |R1| ← cardinality of R1 received from the COUNT query
2  |R2| ← cardinality of R2 received from the COUNT query
3  |R1_arrived| ← cardinality of arrived R1 tuples
4  |R2_arrived| ← 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 ← estimated remaining time (ERT) if continued with SHJ
9          ERT_BJ ← ERT if switched to Bind Join (BJ)
10         ERT_BBJ ← ERT if switched to Bind-Bloom Join (BBJ)
11         Set MIN_ERT to the minimum among ERT_SHJ, ERT_BJ and ERT_BBJ
12         if (MIN_ERT == ERT_BJ) then
13             Set JOIN method as BJ
14             Emit the duplicate results of SHJ and BJ
15         end
16         if (MIN_ERT == ERT_BBJ) then
17             Set JOIN method as BBJ
18             Emit the duplicate results of SHJ and BBJ
19         end
20      end
21 end
```

Equation 4 depicts the 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, $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j, and ADF (R_i, R_j) is the average duplication factor of R_i on each common attribute value of R_i and R_j. The formula for ADF (R_i, R_j) is depicted in Equation 5 where $|R_i|$ is the cardinality of R_i and $|R_{i\_uca}|$ is the

is the cardinality of unique common attribute values in R_i. We use Equation 4 in order to calculate the estimated cardinality of R_j' when all the tuples of R_i arrive. We expect that there is a directional proportion between the join cardinality and the number of tuples of R_j:

$$\left|R_{j\_estimation}'\right| = \frac{\left|R_i \triangleright\triangleleft R_{j\_arrived}\right| . \left|R_j\right|}{\left|R_{j\_arrived}\right|} \Big/ ADF\left(R_i, R_j\right) \tag{4}$$

$$ADF\left(R_i, R_j\right) = \frac{\left|R_i\right|}{\left|R_{i\_uca}\right|} \tag{5}$$

As stated earlier, when all the tuples of R_i arrive, the algorithm estimates the remaining time if extended adaptive join operator continues with symmetric hash join, the remaining time if it changes the join method to bind join, and the remaining time if it changes the join method to bind-bloom join. We have an idea about the data arrival rate of R_j during the execution, so the estimation is possible. Equation 6 shows the estimated remaining time, ERT_{SHJ}, if extended adaptive join operator continues with symmetric hash join where $|R_j|$ 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:

$$ERT_{SHJ} = \frac{\left(\left|R_j\right| - \left|R_{j\_arrived}\right|\right) \cdot t_{R_{j\_arrived}}}{\left|R_{j\_arrived}\right|} \tag{6}$$

Equation 7 shows the estimated remaining time, ERT_{BJ}, if the algorithm switches to bind join. $|R_{i\_uca}|$ is the is the cardinality of unique common attribute values in R_i, t_{ST} is the time for sending one result tuple to the SPARQL endpoint of R_j ($\approx t_{Rj\_arrived} / |R_{j\_arrived}|$), and $|R_{j\_estimation}'|$ is the estimated cardinality of R_j which is reduced by the bindings of R_i. $|R_{j\_arrived}|$ 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_uca} to the endpoint of R_j, and the retrieving time of R_j' from the endpoint of R_j:

$$ERT_{BJ} = \left(\left|R_{i\_uca}\right| \cdot t_{ST}\right) + \frac{\left|R_{j\_estimation}'\right| \cdot t_{R_{j\_arrived}}}{\left|R_{j\_arrived}\right|} \tag{7}$$

Equation 8 shows the estimated remaining time, ERT_{BBJ}, if the algorithm switches to bind-bloom join where b is the number of bits per each element, $|R_{i\_uca}|$ is the cardinality of unique common attribute values in R_i, dr_j is the data arrival rate (in bits/seconds) of the SPARQL endpoint ($\approx s(|R_{j\_arrived}|) / |R_{j\_arrived}|$, where $s(|R_{j\_arrived}|$ is the size of $R_{j\_arrived}$ tuples in bits), $|R_{j\_estimation}'|$ is the estimated cardinality of R_j reduced by the bindings of R_i, $|fp|$ is the estimated cardinality of false positives, $|R_{j\_arrived}|$ 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-bloom join includes sending unique common tuples of R_i in a bloom filter to the endpoint of R_j, and the retrieving time of R_j' from the endpoint of R_j:

$$ERT_{BBJ} = \frac{b \cdot \left|R_{i\_uca}\right|}{dr_j} + \frac{\left(\left|R_{j\_estimation}{}'\right| + \left|fp\right|\right) \cdot t_{R_{j\_arrived}}}{\left|R_{j\_arrived}\right|} \tag{8}$$

2.3. Extended Adaptive Join Operator for Multi-Join Queries

In multi-join queries, we begin with multi-way symmetric hash join (Viglas, Naughton, & Burger, 2003) in order to minimize the response time as in single join queries. The algorithm for multi-join queries is depicted in Algorithm 2. When the tuples from a relation all arrive, called R_i, the algorithm estimates the remaining times if the extended join operator switches to bind join or bind-bloom join for each relation which has a common attribute with R_i. The algorithm chooses the relation with the minimum estimated bind join cost and the minimum estimated bind-bloom cost, called R_j. It compares the estimated remaining times if it changes the join method to bind join or bind-bloom 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.

2.3.1. Cardinality and Remaining Time Estimations

We use the same formula, Equation 4, for single join queries and multi-join queries to estimate the cardinality of the second relation reduced by the bindings of the first relation. We need this estimation in order to calculate the estimated remaining time if extended adaptive join operator switches to bind join or bind-bloom join.

Equation 9 shows the estimated remaining time if the operator continues with multi-way symmetric hash join. Completion time is equal to the maximum completion time of the relations which compose the query:

$$ERT_{MSHJ} = \max\left(\frac{\left(\left|R_k\right| - \left|R_{k\_arrived}\right|\right) \cdot t_{R_{k\_arrived}}}{\left|R_{k\_arrived}\right|}\right) where\ k \in \left(1,2,..,n\right) \tag{9}$$

Equation 10 shows the estimated remaining time if extended 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. The estimated time if the operator uses bind join for R_i and R_j is depicted in Equation 11. $|R_{i\_uca}|$ is the cardinality of unique common attribute values in R_i, t_{ST} is the time for sending one result tuple to the SPARQL endpoint of R_j ($\approx t_{Rj\_arrived} / |R_{j\_arrived}|$), $|R_{j\_estimation}{}'|$ is the estimated cardinality of R_j which is reduced by the bindings of R_i, $|R_{j\_arrived}|$ 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 Equation 12:

$$ERT_{BJ\_R_{ij}} = \max\left(ET_{BJ\_R_{ij}}; ERT_{rest}\right) \tag{10}$$

$$ET_{BJ\_R_{ij}} = \left(\left|R_{i\_uca}\right| \cdot t_{ST}\right) + \frac{\left|R_{j\_estimation}{}'\right| \cdot t_{R_{j\_arrived}}}{\left|R_{j\_arrived}\right|} \tag{11}$$

$$ERT_{rest} = \max\left(\frac{\left(\left|R_k\right| - \left|R_{k\_arrived}\right|\right) \cdot t_{R_{k\_arrived}}}{\left|R_{k\_arrived}\right|}\right) where\ k \in (1,2,..,n) and\ k \neq i, k \neq j \quad (12)$$

Algorithm 2. Extended adaptive join operator for multi-join queries

```
1  S ← {R1, R2, …, Rn}
2  Send COUNT queries to the endpoints of R1, R2, …, Rn
3  MIN_ERT_BJ = MIN_ERT_BBJ ← ∞
4  MIN_ET_BJ = MIN_ET_BBJ ← ∞
5  BJ_Candidate = BBJ_Candidate ← φ
6  Start MSHJ(S)
7  while (S is not empty) do
8       if (all the tuples of Ri arrive) then
9          ERT_MSHJ ← estimated remaining time (ERT) if continued with MSHJ
10         foreach Rj having a common attribute with Ri do
11            ERT_BJ_Rij ← ERT if switched to BJ for Ri and Rj
               ERT_BBJ_Rij ← ERT if switched to BBJ for Ri and Rj
               ET_BJ_Rij ← estimated time for BJ between Ri and Rj
               ET_BBJ_Rij ← estimated time for BBJ between Ri and Rj
12            if (ERT_BJ_Rij < MIN_ERT_BJ) then
13               MIN_ERT_BJ ← ERT_BJ_Rij
14               MIN_ET_BJ ← ET_BJ_Rij
15               BJ_Candidate ← {Ri, Rj}
16            end
17            if (ERT_BBJ_Rij < MIN_ERT_BBJ) then
18               MIN_ERT_BBJ ← ERT_BBJ_Rij
19               MIN_ET_BBJ ← ET_BBJ_Rij
20               BBJ_Candidate ← {Ri, Rj}
21            end
22         end
23         if (MIN_ERT_BJ <= ERT_MSHJ) then
24            if (ET_BBJ_Rij < ET_BJ_Rij) then
25               Ri+ ← BBJ (Ri, Rj)
26               S ← S - BBJ_Candidate + {Ri+}
27               Run MSHJ(S) and eliminate duplicate results
28            end
29            Ri+ ← BJ (Ri, Rj)
30            S ← S - BJ_Candidate + {Ri+}
31            Run MSHJ(S) and eliminate duplicate results
32         end
33      end
34 end
```

Equation 13 shows the estimated remaining time if the extended adaptive join operator switches to bind-bloom join for R_i and R_j, and uses multi-way symmetric hash join for the other relations which are involved in the query. The estimated time if the operator uses bind-bloom join for R_i and R_j is depicted in Equation 14. b is the number of bits per each element, $|R_{i\_uca}|$ is the cardinality of R_i, and dr_j is the data arrival rate (in bits/seconds) of the SPARQL endpoint ($\approx s(|R_{j\_arrived}|) / |R_{j\_arrived}|$, where

$s(|R_{j\_arrived}|$ is the size of $R_{j\_arrived}$ tuples in bits). $|R_{j\_estimation}'|$ is the estimated cardinality of R_j reduced by the bindings of R_i, $|fp|$ is the estimated cardinality of false positives, $|R_{j\_arrived}|$ is the cardinality of arrived tuples of R_j, and $t_{Rj\_arrived}$ is the time for $R_{j\_arrived}$ tuples to arrive. We use Equation 4 and Equation 12 in order to calculate $|R_{j\_estimation}'|$ and ERT_{rest}, respectively:

$$ERT_{BBJ\_R_{ij}} = \max\left(ET_{BBJ\_R_{ij}}; ERT_{rest}\right) \tag{13}$$

$$ET_{BBJ\_R_{ij}} = \frac{b \cdot \left|R_{i\_uca}\right|}{dr_j} + \frac{\left(\left|R_{j\_estimation}'\right| + \left|fp\right|\right) \cdot t_{R_{j\_arrived}}}{\left|R_{j\_arrived}\right|} \tag{14}$$

3. PERFORMANCE EVALUATION

In this section, first we explain the experimental environment, and then present evaluation on the performances of symmetric hash join/multi-way symmetric hash join, bind join, bind-bloom join, adaptive join operator and extended adaptive join operator for single join queries and multi-join queries. The focus of the evaluation is on their performances with respect to the response time and the completion time. Speedup[1] comparison between our previous proposal, adaptive join operator (Oguz et al., 2016), and extended adaptive join operator is also presented to be self-contained and to show the contribution of our new proposal.

Query cost in distributed environments is mainly dominated by the communication cost (Ozsu & Valduriez, 2011). We conducted our experiments in the network simulator *ns-3*[2] to simulate the real network conditions and consider mainly the communication cost. 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 / *max*(cardinality of *R1*, cardinality of *R2*) (Shekita, Young, & Tan, 1993). We set the low, medium and high cardinality as 1000 tuples, 5000 tuples and 10000 tuples, respectively. Average duplication factors on the common attributes of relations are assigned randomly between 1 and 5, both inclusive. Average duplication factor = 1 means that there are not any duplicates, while average duplication factor = 5 means that there are 5 duplicates per value in average on the common attributes of the relations. For this reason, we ran each test 100 times when we assigned the duplication factors randomly. In some cases, we fixed the average duplication factors in order to understand the impact of the duplication factors as well. We used 8 bits per each element and 6 hash functions for bloom join with bloom filter. We conducted the simulations with different data arrival rates as explained in the following sections, however we always fixed their delays to 10 ms.

3.1. Performance Evaluation for Single Join Queries

In this subsection, we compare extended adaptive join operator (EAJO) with symmetric hash join (SHJ), bind join (BJ), bind-bloom join (BBJ) and adaptive join operator (AJO) 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. In addition, we compare AJO and EAJO with different m / n values and k independent hash functions where m refers to the number of bits in the bit vector, and n refers to the number of elements in the set.

3.1.1. Impact of Data Sizes

The behaviours of the SHJ, BJ, BBJ, AJO and EAJO were analyzed when the data arrival rates of both endpoints were fixed to 0.5 Mbps while the data sizes of *R1* and *R2* were changed. In order to

analyze all conditions, we evaluated the response time and the completion time when the data sizes of *R1* and *R2* were 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. Average duplication factors on the common attributes of relations were given randomly between 1 and 5, both inclusive.

As Figure 1.a shows, BBJ and BJ have the worst response time in all conditions, whereas SHJ, AJO and EAJO behave similarly. As the data size of *R1* increases, the response times of BJ and BBJ increase as well, due to waiting for the arrival of all results of *R1* and sending the unique common attributes to the endpoint of *R2*. BBJ provides a slightly better response time than BJ due to the usage of bloom filter for sending the common attributes. On the other hand, SHJ, AJO and EAJO 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 Figure 1.b, the completion time of BBJ is always shorter than BJ's due to the bloom filter usage. For this reason, we consider the completion times of BBJ instead of BJ's for comparing with others. When the cardinalities are low-medium, low-high and medium-high, (i.e., $|R1| < |R2|$), BBJ's completion time is the shortest. However, EAJO's completion time is quite similar to BBJ's because it changes the join method to BBJ when it decides that it is more efficient than SHJ or BJ. EAJO performs the best when the cardinalities of relations are medium-low, high-low and high-medium (i.e., $|R1| > |R2|$), respectively. When the cardinalities of *R1* and *R2* are the same, SHJ, AJO and EAJO provide the best performance in completion time at the same time. The data arrival rates and the cardinalities of the relations are the same in these cases. As a result, all the tuples of both relations arrive at the same time. SHJ is the most efficient join method for these cases. Therefore, both AJO and EAJO decide to continue with SHJ in such cases. To conclude the comparison of completion times, we can say that EAJO has the capability to choose the most efficient join method during the execution. For this reason, it provides or shares the best completion time in six of nine conditions. Also, it provides similar completion time to the best join method in the remaining three conditions.

Figure 1.c shows the speedup in completion time by EAJO compared to AJO. As shown in the figure, when the cardinalities of relations are different, EAJO provides speedup between 17.8% and 19.4%. The reason of the difference between the speedup percentages is based on the different average duplication factors. We can say the speedup of EAJO compared to AJO is 18.2% in average. EAJO does not provide speedup when the cardinalities of relations are the same, because both AJO and EAJO decide to continue with SHJ for the reasons explained previously.

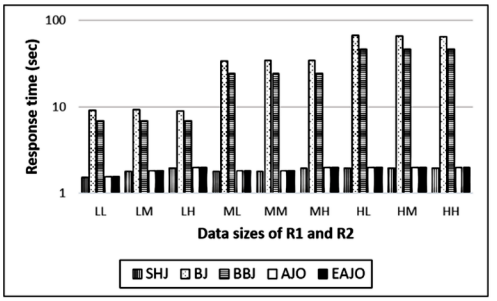
3.1.2. Impact of Data Arrival Rates

In this case, we fixed the data arrival rate of *R1* 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*. Average duplication factors on the common attributes of relations were given randomly between 1 and 5, both inclusive. However, we fixed the average duplication factors when we calculated the speedup of EAJO compared to AJO in order to understand the impact of the duplication factors as well.

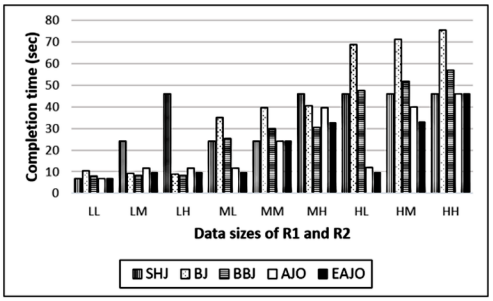
3.1.2.1. Low Cardinality of R1 and High Cardinality of R2

We conducted the simulations for two different conditions: i) when the data arrival rate of *R1* was fixed to 2 Mbps, and ii) when the data arrival rate of *R1* was fixed to 0.5 Mbps. As Figures 2.a and 2.b show, the response times of BJ and BBJ are always longer than SHJ, AJO and EAJO. The gap between the response times of BJ and BBJ; and the others increases when the data arrival rate of *R2* gets slower. SHJ provides the shortest response time in both conditions. AJO and EAJO provide almost the same response time due to beginning with SHJ. That is to say, SHJ, AJO and EAJO are the best in terms of response time at the same time.

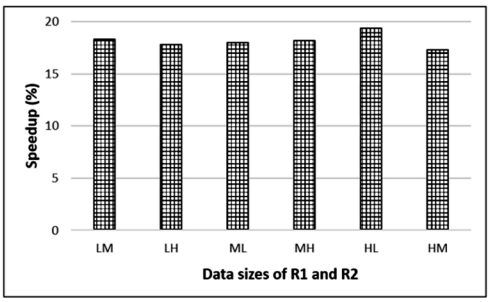
Figure 1. Data arrival rates of R1 and R2 are fixed



(a) Response time

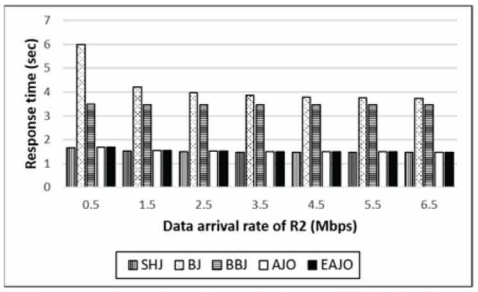


(b) Completion time

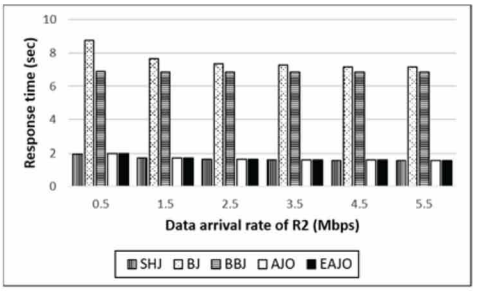


(c) Speedup of EAJO compared to AJO

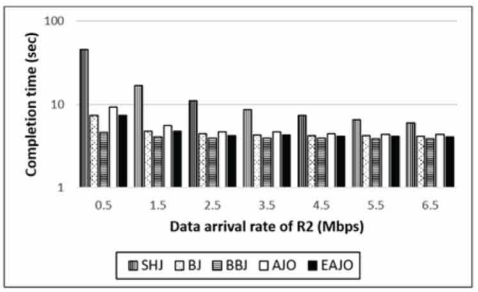
Figure 2. Data sizes of R1 and R2 are fixed with card(R1) << card(R2)



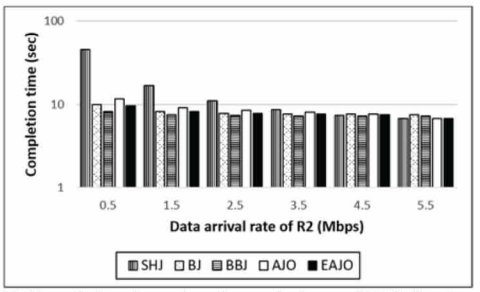
(a) Response time when data arrival rate of *R1* is fixed to 2 Mbps and data arrival rate of *R2* is changed



(b) Response time when data arrival rate of *R1* is fixed to 0.5 Mbps and data arrival rate of *R2* is changed



(c) Completion time when data arrival rate of *R1* is fixed to 2 Mbps and data arrival rate of *R2* is changed



(d) Completion time when data arrival rate of *R1* is fixed to 0.5 Mbps and data arrival rate of *R2* is changed

As shown in Figure 2.c, the completion time of BBJ is always shorter than BJ due to the bloom filter usage. For this reason, we use the completion time of BBJ instead of the completion time of BJ when we compare the completion times of operators. BBJ provides the shortest completion time for all conditions because the first relation's cardinality is low and its data arrival rate is relatively fast. As the data arrival rate of the second relation gets faster, EAJO provides similar completion time with BBJ. The completion time of EAJO is always faster than SHJ and AJO. Figure 2.d shows the completion time comparison when the first relation's data arrival rate is fixed to 0.5 Mbps. BBJ provides the shortest completion time until the second relation's data arrival rate is 4.5 Mbps. However, EAJO has almost the same completion time with BBJ due to its ability to change the join method to BBJ during the execution. When the second relation's data arrival rate is faster or equal to 5.5 Mbps, SHJ provides the shortest completion time. In these cases, AJO and EAJO have the same completion time due to continuing with SHJ. That is to say, the winner of completion time is changed according to the data arrival rates. However, EAJO can choose the best join method during the execution.

Table 2 shows the speedup in completion time of EAJO compared to AJO when the data arrival rate of *R1* is fixed to 2 Mbps and the data arrival rate of *R2* is changed. The used average duplication factors are 1, 2 and 5, respectively where 1 means there are not any duplicates. For each data arrival rate of *R2*, AJO and EAJO change the join method to BJ and BBJ, respectively. Although EAJO provides speedup in all cases due to decreasing the data size of unique common attributes by using a bloom filter, the speedup decreases as the second relation's data arrival rate increases. The reason of this decrease in the speedup is because of the effect of the decrease in the size of the sent data as the network speed increases. Another key point to remember is that the speedup remains quite similar after a certain point due to the same reason. Table 3 shows the speedup gained by EAJO when the

Table 2. Speedup for card(R1) << card(R2) when data arrival rate of R1 is 2 Mbps

Data Arrival Rate of R2 in Mbps	Average Duplication Factors		
	1	2	3
0.5	35.28%	22.53%	11.57%
1.5	25.65%	13.99%	7.07%
2.5	20.39%	10.71%	5.70%
3.5	15.99%	8.47%	4.80%
4.5	12.51%	7.44%	4.47%
5.5	10.55%	6.81%	4.32%
6.5	9.64%	6.37%	4.24%

Table 3. Speedup for card(R1) << card(R2) when data arrival rate of R1 is 0.5 Mbps

Data Arrival Rate of R2 in Mbps	Average Duplication Factors		
	1	2	3
0.5	30.61%	18.62%	9.16%
1.5	17.29%	8.72%	4.20%
2.5	12.41%	6.08%	3.12%
3.5	9.75%	4.91%	2.71%
4.5	-	4.27%	2.51%

first relation's data arrival rate is fixed to 0.5 Mbps. In this case, EAJO provides speedup until the second relation's data arrival rate is equal or faster than 4.5 Mbps, because both AJO and EAJO decide to continue with SHJ after this data arrival rate. As shown in both Table 2 and Table 3, the speedup decreases as the duplication factor increases.

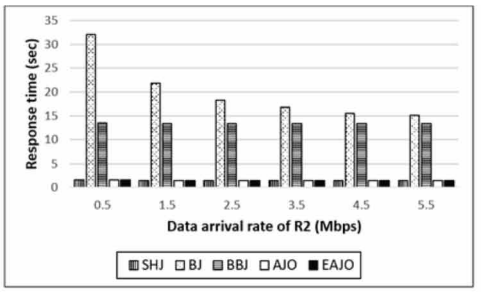
3.1.2.2. High Cardinality of R1 and Low Cardinality of R2

We again conducted the simulations for two different conditions: i) when the data arrival rate of *R1* was fixed to 2 Mbps, and ii) when the data arrival rate of *R1* was fixed to 0.5 Mbps. The results observed from Figure 3.a and Figure 3.b are similar to the results in Figure 2.a and Figure 2.b. Since the cardinality of the first relation is high in this case, response times of BJ and BBJ are dramatically longer than SHJ and also longer than AJO and EAJO as expected. The response times of SHJ, AJO and EAJO are nearly the same.

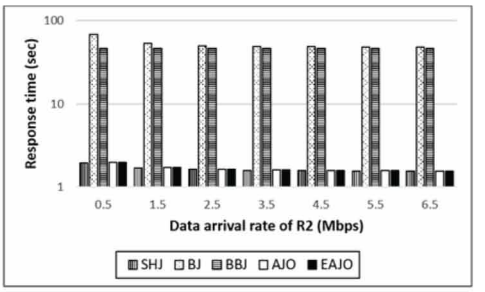
As shown in Figure 3.c, the completion time of EAJO is the best in all conditions. SHJ, BJ and BBJ wait the arrival of all tuples related to the first relation whose cardinality is high. However, AJO and EAJO can change the join method and the join order when the second relation's tuples all arrive. Compared to AJO, EAJO has the advantage of changing the join method to BBJ. Figure 3.d shows the completion time comparison when the first relation's data arrival rate is fixed to 0.5 Mbps. The results are similar to the previous one. EAJO's completion time is the shortest once again. The gap between the others is even higher.

Table 4 and Table 5 show the gained speedup in completion time by EAJO compared to AJO. In all conditions, both AJO and EAJO change the join order as $R2 \bowtie R1$. The gained time of EAJO compared to AJO remains the same, because the unique common attributes are sent to the endpoint of *R1* and its data arrival rate is fixed. However, overall time decreases up to a certain value as the data arrival rate of *R2* increases. For this reason, the speedup increases up to that certain value for both conditions as the data arrival rate of *R2* increases. The speedup also increases as the duplication factor decreases.

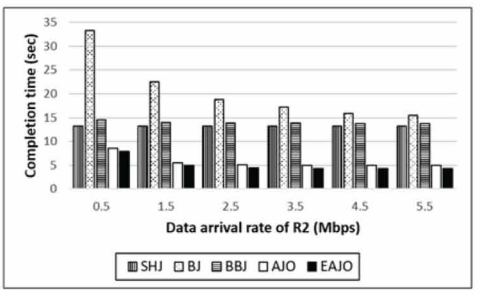
Figure 3. Data sizes of R1 and R2 are fixed with card(R1) >> card(R2)



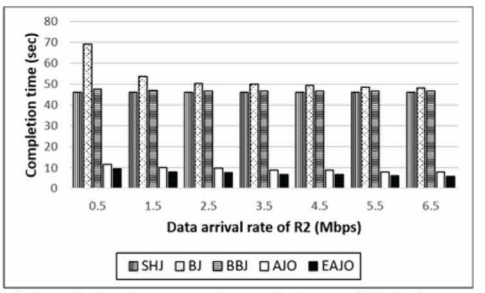
(a) Response time when data arrival rate of *R1* is fixed to 2 Mbps and data arrival rate of *R2* is changed



(b) Response time when data arrival rate of *R1* is fixed to 0.5 Mbps and data arrival rate of *R2* is changed



(c) Completion time when data arrival rate of *R1* is fixed to 2 Mbps and data arrival rate of *R2* is changed



(d) Completion time when data arrival rate of *R1* is fixed to 0.5 Mbps and data arrival rate of *R2* is changed

Table 4. Speedup for card(R1) >> card(R2) when data arrival rate of R1 is 2 Mbps

Data Arrival Rate of R2 in Mbps	Average Duplication Factors		
	1	2	3
0.5	14.47%	7.12%	3.53%
1.5	20.92%	10.89%	5.58%
2.5	22.80%	12.08%	6.26%
3.5	23.24%	12.37%	6.42%
4.5	23.24%	12.37%	6.42%
5.5	23.24%	12.37%	6.42%

Table 5. Speedup for card(R1) >> card(R2) when data arrival rate of R1 is 0.5 Mbps

Data Arrival Rate of R2 in Mbps	Average Duplication Factors		
	1	2	3
0.5	30.61%	18.62%	9.16%
1.5	33.51%	21.00%	10.60%
2.5	35.28%	22.53%	11.57%
3.5	37.37%	24.40%	12.81%
4.5	37.37%	24.40%	12.81%
5.5	39.80%	26.69%	14.39%
6.5	39.80%	26.69%	14.39%

3.1.3. Impact of Bit Vector Size

As explained in the previous sections, a bloom filter represents a set $S = \{e_1, e_2, e_3, \ldots, e_n\}$ of n elements in a vector v of m bits. Initially all the bits are set to 0. Then, k independent hash functions, h_1, h_2, . . ., h_k, with range $\{1, 2, \ldots, m\}$ are used. In this part, we analyze the impact of m / n by changing it between 2 and 22. In each m/n value, we used the number of hash functions, k, which minimizes the false positive rate (Fan et al., 2000). The m/n and k combinations used in our experiments can be seen from Table 6. We fixed the data arrival rates of both endpoints to 2 Mbps and the cardinalities of relations to low and high, respectively. First, average duplication factors on the common attribute of relations were given randomly between 1 and 5, both inclusive. Second, the average duplication factors were set to 2.

Figure 4.a shows the achieved speedup in completion time by EAJO compared to AJO in different m/n values when the average duplication factors are random. The results observed from the experiment appears to suggest that the gained speedup is not affected by the m/n value when it is between 6 and 20, inclusively. The best performance is provided when the m/n is equal to 8.

Figure 4.b shows the gained speedup in completion time by EAJO when the average duplication factors are set to 2. The results are similar to the results in Figure 4.a. Since the m/n is between 8 and 16, the speedup values are almost the same.

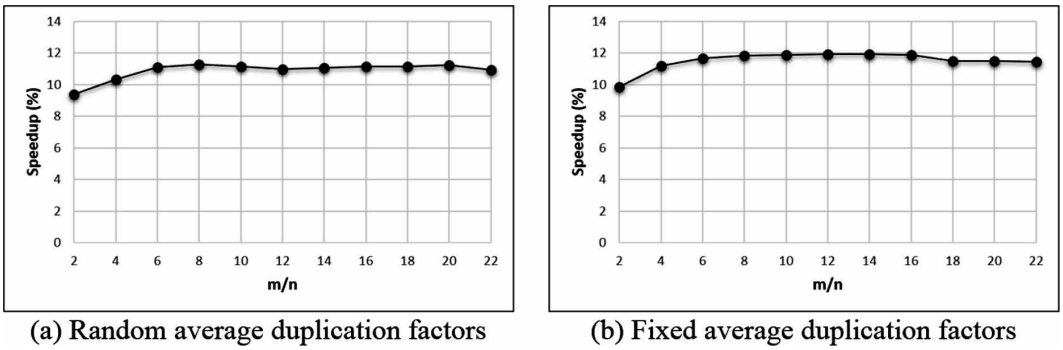
3.1.4. Discussion on the Performance Evaluation for Single Join Queries

The simulation results demonstrated that SHJ provides the best response time performance in all conditions due to being a non-blocking join operator which produces the first result tuple as early as

Table 6. The m/n and k combinations used for bloom filter

m/n	k
2	1
4	3
6	4
8	6
10	7
12	8
14	10
16	11
18	12
20	14
22	15

Figure 4. Speedup of EAJO compared to AJO when m/n and k combinations used



(a) Random average duplication factors (b) Fixed average duplication factors

possible. Our previous and current proposals, AJO and EAJO respectively, provide almost the same response time with SHJ due to setting the join method as SHJ in the beginning. The response times of BJ and BBJ are dramatically longer because of waiting for all tuples of the first relation to arrive.

On the other hand, BJ or BBJ can provide better completion times when the first relation's cardinality is low and the second relation's cardinality is high. However, our previous proposal AJO can change the join method to BJ, and our new proposal EAJO can change the join method to BJ or BBJ in this condition.

EAJO provides the best completion time when the first relation's cardinality is high and the second relation's cardinality is low. This conclusion is valid in all data arrival combinations that we have tested.

To conclude, SHJ is the most successful join method in response time. However, the best join method in completion time can differ according to the relations' cardinalities and their data arrival rates. In addition, the results showed that BBJ provides better completion times than BJ in all conditions. These results seem to suggest that using bloom filters in bind join is a necessity. Our proposal, EAJO, provides an optimal response time by beginning with SHJ. It also provides an optimal completion time by changing the join method or join order during the execution. In brief, EAJO gives the best tradeoff between the response time and the completion time. Another key fact to remember is that EAJO always provides better completion time than AJO.

3.2. Performance Evaluation for Multi-Join Queries

In this subsection, we compare EAJO with multi-way symmetric hash join (MSHJ), BJ, BBJ, and AJO 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 . }
}
```

3.2.1. Impact of Data Sizes

Since our aim in this case is to show the impact of data sizes, we fixed the data arrival rates of all relations to 0.5 Mbps and the delays to 10 milliseconds. We conducted our experiments when the data sizes of *R1*, *R2*, *R3* were low-low-low (LLL); low-medium-high (LMH); low-high-high (LHH); high-medium-low (HML); high-high-low (HHL); and high-high-high (HHH).

As shown in Figures 5.a, 5.b and 5.c, in all cases, MSHJ, AJO and EAJO provide the best response time whereas BJ performs the worst response time and BBJ follows it. When the cardinality of *R1* is high, the response times of BJ and BBJ become dramatically longer due to waiting for the arrival of all results of *R1*. As the duplication factor increases, the response times of BJ and BBJ shorten due to the decrease in the number of unique common attribute values. In other words, the number of attribute values to send to the other endpoints is decreased as the average duplication factor increases. Although the response times of BJ and BBJ decrease as the average duplication factor increases, their response times are dramatically longer than MSHJ, AJO and EAJO.

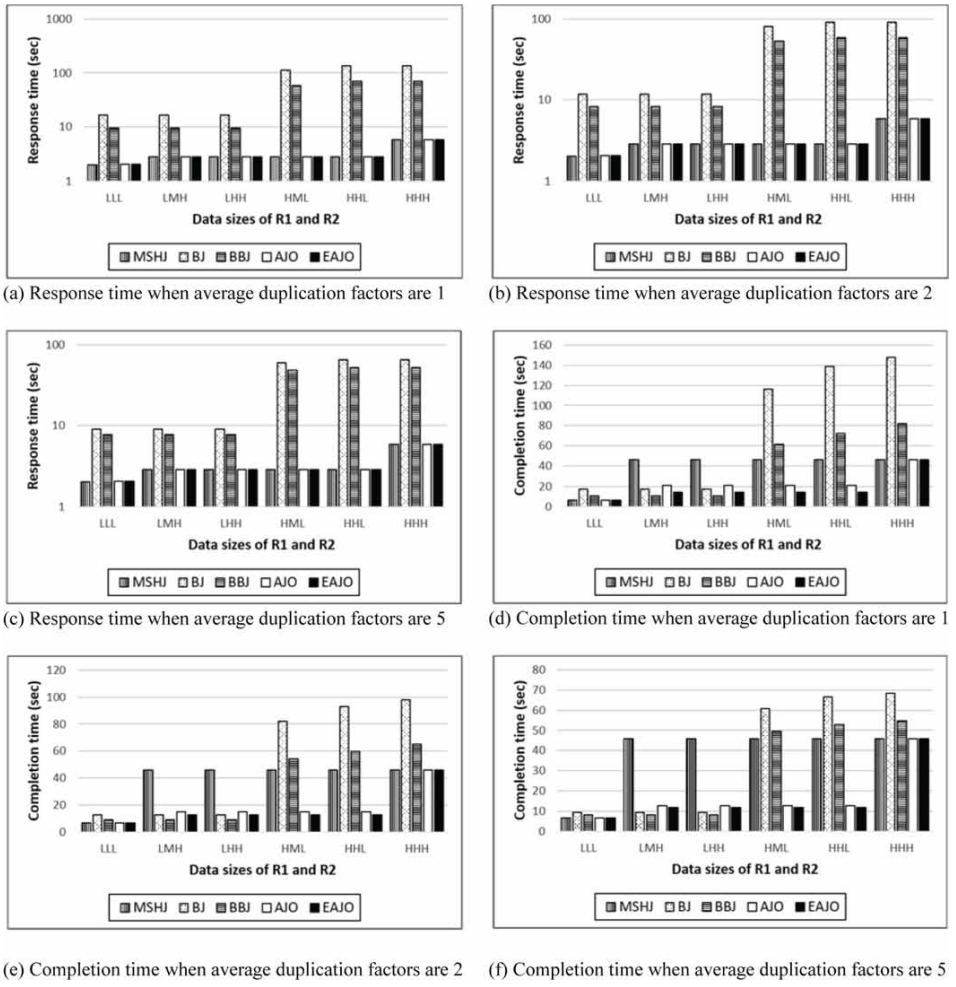
Figures 5.d, 5.e and 5.f show the completion times of MSHJ, BJ, BBJ, AJO and EAJO when the data arrival rates of all relations are fixed. When the cardinalities are HML or HHL, EAJO performs the best completion time and AJO has the closest completion time to it. The difference between EAJO and others, except AJO, is dramatically high. When the cardinalities of all relations are the same, namely LLL or HHH, MSHJ, AJO and EAJO share the best completion time whereas BJ performs the worst. When the cardinalities are LMH or LHH, BBJ performs the shortest completion time. EAJO's completion time is the second best when the average duplication factors are 1. BJ performs slightly better than EAJO when the average duplication factors are 2 or 5. To conclude, EAJO performs or shares the best completion time in four of six cases due to having the adaptation ability.

Table 7 shows the speedup in completion time of EAJO compared to AJO when the data arrival rates of *R1*, *R2* and *R3* are fixed. EAJO provides speedup from 6.40% to 31.33 when the cardinalities of relations are different. Although the speedup is not affected by the cardinalities of relations, it increases as the average duplication factors decrease. EAJO does not provide speedup when the cardinalities of relations are the same, because both AJO and EAJO decide to continue with MSHJ.

3.2.2. Impact of Data Arrival Rates

In order to show the impact of data arrival rates on MSHJ, BJ, BBJ, AJO and EAJO, 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 high cardinality of *R3* (LHH); ii) high cardinality of *R1*, high cardinality of *R2*, and low cardinality of *R3* (HHL). LHH and HHL are chosen because EAJO performs the worst and the best completion times among their results with other combinations in the previous section. Since we showed the effect of average duplication factors previously, we fixed the average duplication factors to 2 in these cases.

Figure 5. Data arrival rates of R1, R2 and R3 are fixed



(a) Response time when average duplication factors are 1

(b) Response time when average duplication factors are 2

(c) Response time when average duplication factors are 5

(d) Completion time when average duplication factors are 1

(e) Completion time when average duplication factors are 2

(f) Completion time when average duplication factors are 5

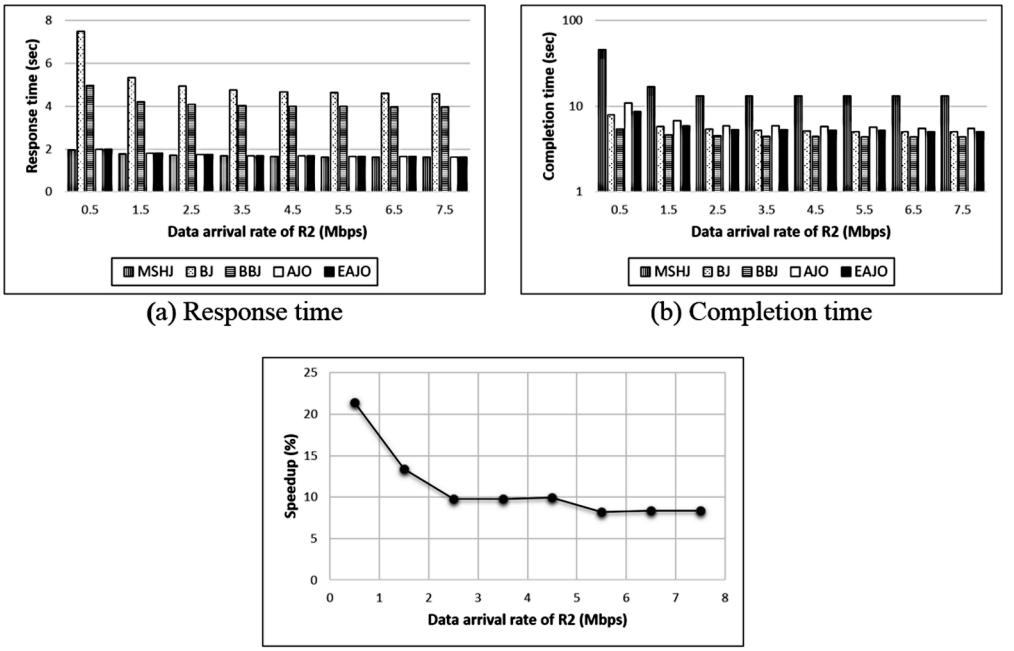
Table 7. Speedup of EAJO compared AJO when data arrival rates are fixed

Data Sizes ofR1, R2 and R3	Average Duplication Factors		
	1	2	3
LMH	31.33%	16.55%	6.40%
LHH	31.33%	16.55%	6.40%
HML	31.33%	16.55%	6.40%
HHL	31.33%	16.55%	6.40%

3.2.2.1. Low Cardinality of R1, High Cardinality of R2, High Cardinality of R3

Figure 6.a shows the response times of MSHJ, BJ, BBJ, AJO and EAJO when the cardinalities of relations are low, high and high respectively. As shown in the figure, response times of MSHJ, AJO and EAJO are almost the same whereas BJ's and BBJ's response times are highly slower.

Figure 6. Data sizes of R1, R2 and R3 are fixed with card(R1) << card(R2) = card(R3)



(a) Response time

(b) Completion time

(c) Speedup of EAJO compared to AJO

Figure 6.b indicates that the completion times in ascending order are of BBJ, BJ, EAJO, AJO and MSHJ. When the first relation's cardinality is low and its data arrival is relatively fast, BBJ and BJ provide better completion times. The completion time of MSHJ is the worst one in all cases due to having the disadvantage of waiting all the tuples of *R2* and *R3*. However, AJO and EAJO change their join methods to BJ and BBJ, respectively, when the tuples of the first relation all arrive. Thus, EAJO performs almost the same completion time to BJ, and provides slightly worse completion time than BBJ. BBJ's and BJ's both response times and completion times would increase, if the first relation's cardinality were medium or high.

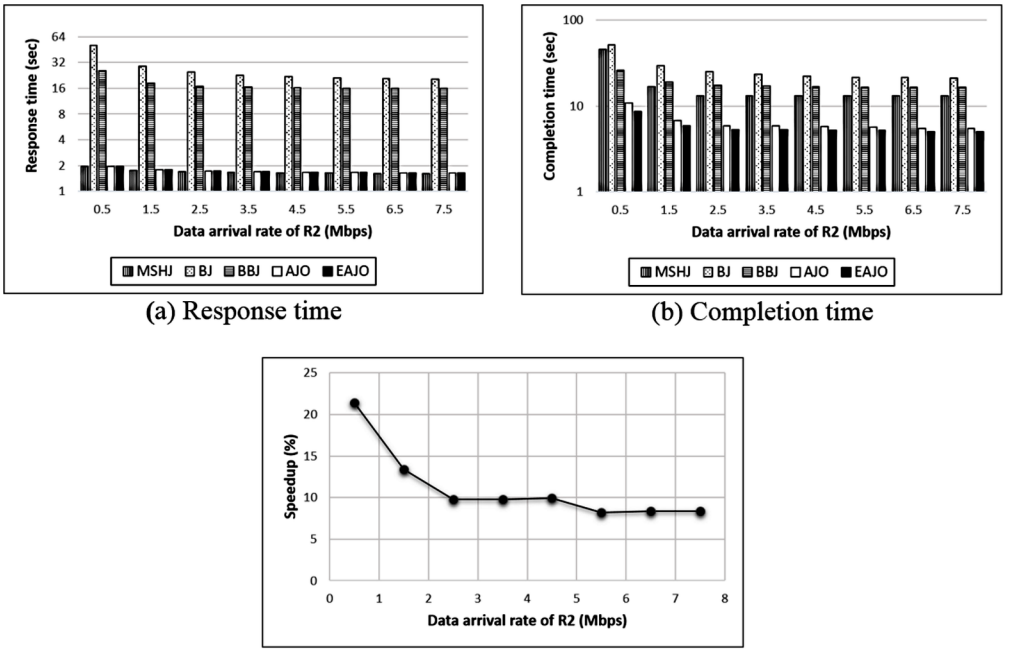
Figure 6.c shows the speedup in completion time of EAJO compared to AJO when the data arrival rate of *R1* is fixed to 2 Mbps and the data arrival rate of *R2* is changed with $card(R1) << card(R2) = card(R3)$. As shown in the figure, the speedup decreases as the second relation's data arrival rate increases. The reason of this decrease in the speedup is because of the effect of the decrease in the size of the sent data as the network speed increases.

3.2.2.2. High Cardinality of R1, High Cardinality of R2, Low Cardinality of R3

The results observed from Figure 7.a are similar to the results in Figure 6.a. BJ and BBJ have the worst response time again, whereas MSHJ, AJO and EAJO have almost the same response time. Since the cardinality of the first relation is high in this case, response times of BJ and BBJ are dramatically longer than others.

As shown in Figure 7.b, EAJO provides the best completion time in all cases. The completion times in ascending order are of EAJO, AJO, MSHJ, BBJ, and BJ when the second relation's data arrival rate is equal or faster than 1.5 Mbps. EAJO and AJO have the advantage of using BJ or BBJ when the tuples of *R3* all arrive whose cardinality is low. EAJO outperforms AJO in all cases due to the usage of bloom filter for sending the common attributes.

Figure 7. Data sizes of R1, R2 and R3 are fixed with card(R1) = card(R2) >> card(R3)



(a) Response time

(b) Completion time

(c) Speedup of EAJO compared to AJO

Figure 7.c shows the speedup in completion time of EAJO compared to AJO when the data arrival rate of *R1* is fixed to 2 Mbps and the data arrival rate of *R2* is changed with *card*(*R1*) = *card*(*R2*) >> *card*(*R3*). The speedup is gained due to the usage of bloom filter and hence sending less data size through the network. The speedup decreases as the second relation's data arrival rate increases, because the effect of the decrease in the size of the sent data decreases as the network speed increases. The results are the same with the results in Figure 6.c. The cardinalities of *R1*, *R2* and *R3* are low-high-high and high-high-low in these cases, respectively. The common attributes exist between *R1* - *R2*; and *R2* - *R3*. In the first case, when the cardinalities are low-high-high, first the tuples of *R1* all arrive, and AJO and EAJO change the join method for *R1* and *R2* to BJ or BBJ, respectively. In the second case, when the cardinalities are high-high-low, first the tuples of *R3* all arrive, and AJO and EAJO change the join method for *R3* and *R2* to BJ or BBJ, respectively. For this reason, the achieved speedups are the same in both cases.

3.2.3. Discussion on the Performance Evaluation for Multi-Join Queries

The simulation results showed that MSHJ, which is a non-blocking join method, provides the best response time in all conditions. AJO and EAJO provide almost the same response time with MSHJ due to setting the join method as MSHJ at the beginning. The response times of BJ and BBJ are dramatically longer because of waiting the arrival of all tuples belonging to the first relation.

The results also demonstrated that BBJ provides the best completion time when the first relation's cardinality is low and the other relations' cardinalities are medium or high. However, EAJO can change the join method to BBJ in these conditions. On the other hand, EAJO provides the best completion time when the first relation's cardinality is high. This conclusion is valid in all data arrival combinations that we have tested.

In conclusion, MSHJ is the best join method in response time. However, the best join method in completion time differs according to the relations' cardinalities and data arrival rates. EAJO provides

an optimal response time by beginning with MSHJ and an optimal completion time by changing the join method or join order during the execution. We can conclude that EAJO gives the best tradeoff between the response time and the completion time. We also emphasize that EAJO always provides better completion time than AJO.

4. RELATED WORK

Linked data contains two aspects: i) a way of publishing and connecting structured data on the web, and ii) the collection of interrelated data sources on the web. There are two main approaches to query these data sources which are link traversal (Hartig et al., 2009) and query federation (Görlitz & Staab, 2011a). Both approaches have the advantage of providing up-to-date results due to distributed query processing. However, link traversal has the weakness of not guaranteeing finding all results and has some performance problems. Because of these reasons, we turn our attention to the second approach.

Query federation is performed via an engine that distributes the query execution over a federation of SPARQL endpoints and has the following main steps: i) data source selection, ii) query optimization, and iii) query execution. Data source selection is responsible for selecting the relevant data sources for each triple pattern which composes the query. Query optimization groups the triple patterns, decides the join strategy and the join order. The last step is responsible for the execution of the query plan which is decided by the query optimizer.

The objective of the federated query engines can be stated as to minimize the response time and the completion time which include communication time, I/O time and CPU time. The communication time dominates the others in distributed environments. Since the subqueries and intermediate results are transmitted over the web of data, the communication cost is affected by the amount of intermediate results. It is substantially affected by the join order and the join method which are decided in the query optimization phase.

Static query optimization and heuristics are widely used in query federation (Quilitz & Leser, 2008; Görlitz & Staab, 2011b; Schwarte, Haase, Hose, Schenkel, & Schmidt, 2011; Wang, Tiropanis, & Davis, 2013). 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 (Deshpande et al., 2007) is a need in this unpredictable environment. ANAPSID (Acosta et al., 2011) and ADERIS (Lynden et al., 2010, 2011) are the two query federation engines which use adaptive query optimization. ANAPSID uses a non-blocking join method based on symmetric hash join (Wilschut & Apers, 1991) and Xjoin (Urhan & Franklin, 2000). ADERIS (Lynden et al., 2010) joins two predicate tables as they become complete, whereas ADERIS (Lynden et al., 2011) employs a cost model for dynamically changing the join order. Also, AVALANCHE (Basca & Bernstein, 2010, 2014) considers adaptivity. It collects statistical information about relevant data sources and then generates its execution plan to provide the first k tuples. Our previous proposal (Oguz et al., 2016), to the best of our knowledge, is the first study that considers an adaptive join operator that aims to reduce both the response time and the completion time when query execution is done over SPARQL endpoints. In this paper, we present the improved version of it which achieves to further minimize the completion time.

Table 8 shows the comparison of adaptive query optimization in query federation depending on the following criteria:

- **Server (S):** Indicates the type of the server for publication and querying of linked data. SPARQL Endpoints *(se)* and triple pattern fragment servers *(tpfs)* are the possible values;
- **Join Method (JM):** Shows the used join methods in the studies which are categorized as nested loop join (*nlj*), index nested loop join *(inlj)*, symmetric hash join *(shj)*, bind join *(bj)*, and bind-bloom join *(bbj)*;

Table 8. Comparison of adaptive query optimization in query federation

	S	JM	ToS	FoF	ToE	LP	PP	ToM
ADERIS (Lynden et al., 2011)	se	inlj/bj	rt	inter	any	op_ro	uao	rs
ANAPSID (Acosta et al., 2011)	se	shj/bj	rt	intra	dar	no	uao	do
AVALANCHE (Basca & Bernstein, 2014)	se	bj/bbj	rt	inter	dar	op_ro	no	rs
nLDE (Acosta & Vidal, 2015)	tpfs	shj/nlj	md	intra	any	op_ro	no	rs
AJO (Oguz et al., 2016)	se	shj/bj	rt	intra	dar	rf	op_rep	rs&rp
EAJO	se	shj/bj/bbj	rt	intra	dar	rf	op_rep	rs&rp

- **Type of Statistics (ToS):** States of the collection time of statistics which has the following values: runtime *(rt)* and metadata *(md)*;
- **Frequency of Feedback (FoF):** Shows the level of modification and has two possible values: inter-operator *(inter)* and intra-operator *(intra)*;
- **Type of Event (ToE):** Shows the case triggering the decision and has two values which are data arrival rates *(dar)* and *any*;
- **Logical Plan (LP):** Displays the query plan modifications at the logical level and are categorized as reformulation of the remaining plan *(rf)*, operator reordering *(op_ro)*, and no effects *(no)* for adaptive query optimization in relational databases by Gounaris et al. (Gounaris, Paton, Fernandes, & Sakellariou, 2002). Reformulation of the remaining plan includes the operator reordering;
- **Physical Plan (PP):** Represents the query plan modifications at the physical level and are categorized as usage of adaptive operators *(uao)*, operator replacement *(op_rep)*, and no effects *(no)* for relational databases by Gounaris et al. (Gounaris et al., 2002);
- **Type of Modification (ToM):** Can be employed as rescheduling *(rs)*, dynamic operator *(do)*, and rescheduling and replacement *(rs & rp)*.

As shown in Table 8, ADERIS, ANAPSID, AVALANCHE, AJO and EAJO use adaptive query optimization for the queries over SPARQL endpoints, whereas nLDE employs adaptive query optimization for queries over triple pattern fragments. The proposals for the SPARQL endpoints prefer to collect the statistics in runtime due to unreliable or missing statistics. Therefore, up-to-dateness of statistics is provided. On the other hand, nLDE uses metadata catalogs for the statistics because triple pattern fragments contain both data, metadata and controls.

The second parameter in Table 8 is the join method. Bind join is used by all the studies, except nLDE, and nested loop join is employed by ADERIS and nLDE. ANAPSID proposes two join methods which are agjoin and adjoin. The first one is a non-blocking join method which is based on symmetric hash join and XJoin. The second one is the extended version of dependent join (Florescu, Levy, Manolescu, & Suciu, 1999) which sends the request to the second data source when tuples from the first source are received. Adjoin can be accepted as a bind join because it needs the bindings. As illustrated in Table 8, ANAPSID, AJO, nLDE and EAJO have the opportunity to produce results incrementally since they use symmetric hash join. AVALANCHE defines its join method as distributed join and it employs bloom filter optimised joins to reduce communication cost. The difference between distributed join and bind join is not explained in their papers. We categorize its join methods as bind join and bind-bloom join. In brief, AVALANCHE and EAJO can use bind-bloom join which has the advantage of decrease the completion time.

The third parameter for the comparison is the frequency of feedback. The studies in inter-operator level collect feedback from different physical operators and react to the execution of them according to the feedback. On the other hand, feedback is collected during the processing of the physical operator in the intra-operator level. The limit of collection can vary from a single tuple to a block of tuples (Gounaris et al., 2002). ADERIS and AVALANCHE have the inter-operator feedback frequency, whereas ANAPSID, nLDE, AJO and EAJO have the intra-operator one. ANAPSID's feedback belongs to using an adaptive operator. The difference between the intra-operator of nLDE and AJO/EAJO is based on the amount of accumulated data before reacting. Although nLDE checks the feedback for each tuple, AJO and EAJO do it when all tuples of a relation arrive. The next parameter is the type of event. ANAPSID, AVALANCHE, AJO and EAJO focus on data arrival rates, whereas ADERIS and nLDE check their decisions at each step.

AJO and EAJO distinguish from others when we consider the sixth and seventh parameters in Table 8, namely logical plan and physical plan. Different from others, AJO and EAJO provide reformulation of the remaining plan at the logical level, and operator replacement at the physical level by the ability of changing both the join order and the join method.

The last comparison parameter is the type of modification. ANAPSID's type of modification belongs to a dynamic operator, whereas the types of modification of ADERIS, AVALANCHE and nLDE are rescheduling due to changing the join order for the rest of the query. AJO and EAJO, besides rescheduling, cover replacement which has the meaning of changing the join method.

5. CONCLUSION

In this paper, we presented an adaptive join operator for single join queries and multi-join queries which is an extended version of our previous work (Oguz et al., 2016). We improved our previous adaptive join operator to further reduce the communication cost. For this reason, we integrated bind-bloom join to our operator. Our new proposal always begins with symmetric hash join (multi-way symmetric hash join for multi-join queries) in order to provide optimal response time. It can change the join method to bind join or bind-bloom join when it decides that the candidate join method is more efficient than symmetric hash join for the rest of the query.

The results of the performance evaluation showed the efficiency of the proposed join operator. Compared to symmetric hash join and multi-way symmetric hash join, it provides faster completion times and almost the same response times. Compared to bind join and bind-bloom join, the extended operator performs substantially better with respect to the response time and it can also improve the completion time. Furthermore, the extended operator provides faster completion time than our previous operator in all conditions, because it uses a bloom filter for sending the common attributes to the other endpoint. Experimental results also showed that bind-bloom join provides better completion times than bind join in all conditions. These results allow us to suggest using bloom filters in bind join.

ACKNOWLEDGMENT

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

REFERENCES

Acosta, M., & Vidal, M.-E. (2015). Networks of linked data eddies: An adaptive Web query processing engine for RDF data. In The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA (pp. 111–127). Springer International Publishing.

Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., & Ruckhaus, E. (2011). *ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In The Semantic Web – ISWC 2011, LNCS* (Vol. 7031, pp. 18–34). Springer Berlin Heidelberg.

Babu, S., & Bizarro, P. (2005). Adaptive Query Processing in the Looking Glass. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA (pp. 238–249).

Basca, C., & Bernstein, A. (2010, November 9). Avalanche: Putting the Spirit of the Web back into Semantic Web Querying. *Proceedings of the ISWC 2010 Posters & Demonstrations Track: Collected Abstracts*, Shanghai, China.

Basca, C., & Bernstein, A. (2014). Querying a messy web of data with Avalanche. *Journal of Web Semantics*, *26*, 1–28. doi:10.1016/j.websem.2014.04.002

Bloom, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, *13*(7), 422–426. doi:10.1145/362686.362692

Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., & Varghese, G. (2006). An Improved Construction for Counting Bloom Filters. In *Algorithms – ESA 2006: 14th Annual European Symposium*, Zurich, Switzerland (pp. 684–695). Berlin, Heidelberg: Springer. doi:10.1007/11841036_61

Deshpande, A., Ives, Z., & Raman, V. (2007). Adaptive Query Processing. *Found. Trends Databases*, *1*(1), 1–140. doi:10.1561/1900000001

Fan, L., Cao, P., Almeida, J., & Broder, A. Z. (2000). Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, *8*(3), 281–293. doi:10.1109/90.851975

Florescu, D., Levy, A., Manolescu, I., & Suciu, D. (1999). Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Record*, *28*(2), 311–322. doi:10.1145/304181.304210

Görlitz, O., & Staab, S. (2011a). Federated Data Management and Query Optimization for Linked Open Data. In New Directions in Web Data Management 1 (Vol. 331, pp. 109–137). Springer Berlin Heidelberg. doi:10.1007/978-3-642-17551-0_5

Görlitz, O., & Staab, S. (2011b, October 23). SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. *Proceedings of the Second International Workshop on Consuming Linked Data (COLD '11)*, Bonn, Germany.

Gounaris, A., Paton, N. W., Fernandes, A. A. A., & Sakellariou, R. (2002). Adaptive Query Processing: A Survey. In *Advances in Databases: 19th British National Conference on Databases, BNCOD 19, Sheffield*, UK (pp. 11–25). Berlin, Heidelberg: Springer. doi:10.1007/3-540-45495-0_2

Gounaris, A., Tsamoura, E., & Manolopoulos, Y. (2013). Adaptive Query Processing in Distributed Settings. In B. Catania & L. C. Jain (Eds.), Advanced Query Processing (Vol. 1, pp. 211–236). Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-28323-9_9

Groppe, S., Heinrich, D., & Werner, S. (2015). Distributed join approaches for W3C-conform SPARQL endpoints. *Open Journal of Semantic Web*, *2*(1), 30–52.

Haas, L. M., Kossmann, D., Wimmers, E. L., & Yang, J. (1997). Optimizing Queries Across Diverse Data Sources. *Proceedings of the 23rd International Conference on Very Large Data Bases* (pp. 276–285). Morgan Kaufmann Publishers Inc.

Hartig, O., Bizer, C., & Freytag, J.-C. (2009). *Executing SPARQL Queries over the Web of Linked Data. In the Semantic Web - ISWC 2009, LNCS* (Vol. 5823, pp. 293–309). Springer Berlin Heidelberg.

Hogenboom, A., Frasincar, F., & Kaymak, U. (2013). Ant colony optimization for {RDF} chain queries for decision support. *Expert Systems with Applications*, *40*(5), 1555–1563. doi:10.1016/j.eswa.2012.08.074

Hogenboom, A., Milea, V., Frasincar, F., & Kaymak, U. (2009). RCQ-GA: RDF Chain Query Optimization Using Genetic Algorithms. In *E-Commerce and Web Technologies: 10th International Conference, EC-Web 2009*, Linz, Austria (pp. 181–192). Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-03964-5_18

Hose, K., & Schenkel, R. (2012). Towards Benefit-based RDF Source Selection for SPARQL Queries. *Proceedings of the 4th International Workshop on Semantic Web Information Management* (pp. 2:1–2:8). New York, USA: ACM. doi:10.1145/2237867.2237869

Ives, Z. G., & Taylor, N. E. (2008). Sideways Information Passing for Push-Style Query Processing. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering* (pp. 774-783). doi:10.1109/ICDE.2008.4497486

Kalayci, E. G., Kalayci, T. E., & Birant, D. (2015). An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns. *Information Systems*, *50*, 51–68. doi:10.1016/j.is.2015.01.013

Lynden, S., Kojima, I., Matono, A., & Tanimura, Y. (2010). Adaptive Integration of Distributed Semantic Web Data. *Proceedings of the 6th International Conference on Databases in Networked Information Systems* (pp. 174–193). Springer-Verlag. doi:10.1007/978-3-642-12038-1_12

Lynden, S., Kojima, I., Matono, A., & Tanimura, Y. (2011). ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints. *Proceedings of the 2011th Confederated International Conference on the Move to Meaningful Internet Systems* (pp. 808–817). Springer-Verlag.

Mackert, L. F., & Lohman, G. M. (1986). R* Optimizer Validation and Performance Evaluation for Local Queries. *SIGMOD Record*, *15*(2), 84–95. doi:10.1145/16856.16863

Michael, L., Nejdl, W., Papapetrou, O., & Siberski, W. (2007). Improving distributed join efficiency with extended bloom filter operations. *Proceedings of the 21st International Conference on Advanced Information Networking and Applications (AINA '07)* (p. 187-194). doi:10.1109/AINA.2007.80

Morvan, F., & Hameurlain, A. (2009). Dynamic Query Optimisation*: Towards Decentralised Methods. Int. J. Intell. Inf. Database Syst.*, *3*(4), 461–482. doi:10.1504/IJIIDS.2009.030440

Mullin, J. K. (1990). Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, *16*(5), 558–560. doi:10.1109/32.52778

Oguz, D., Ergenc, B., Yin, S., Dikenelli, O., & Hameurlain, A. (2015). Federated query processing on linked data: A qualitative survey and open challenges. *The Knowledge Engineering Review*, *30*(5), 545–563. doi:10.1017/S0269888915000107

Oguz, D., Yin, S., Hameurlain, A., Ergenc, B., & Dikenelli, O. (2016). Adaptive Join Operator for Federated Queries over Linked Data Endpoints. In *Advances in Databases and Information Systems: 20th East European Conference, ADBIS 2016*, Prague, Czech Republic (pp. 275–290). Cham: Springer International Publishing. doi:10.1007/978-3-319-44039-2_19

Oren, E., Guéret, C., & Schlobach, S. (2008). Anytime Query Answering in RDF Through Evolutionary Algorithms. *Proceedings of the 7th International Conference on The Semantic Web* (pp. 98–113). Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-540-88564-1_7

Ozsu, M. T., & Valduriez, P. (2011). *Principles of Distributed Database Systems*. New York: Springer.

Quilitz, B., & Leser, U. (2008). Querying Distributed RDF Data Sources with SPARQL. *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications* (pp. 524–538). Springer-Verlag. doi:10.1007/978-3-540-68234-9_39

Schwarte, A., Haase, P., Hose, K., Schenkel, R., & Schmidt, M. (2011). FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *the Semantic Web - ISWC 2011 - 10th International Semantic Web Conference*, Bonn, Germany (pp. 601–616).

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access Path Selection in a Relational Database Management System. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (pp. 23–34). New York, NY, USA: ACM. doi:10.1145/582095.582099

Shekita, E. J., Young, H. C., & Tan, K.-L. (1993). Multi-Join Optimization for Symmetric Multiprocessors. *Proceedings of the 19th International Conference on Very Large Data Bases* (pp. 479–492). Morgan Kaufmann Publishers Inc.

Urhan, T., & Franklin, M. J. (2000). XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Eng. Bull.*, *23*(2), 27–33.

Viglas, S. D., Naughton, J. F., & Burger, J. (2003). Maximizing the Output Rate of Multi-way Join Queries over Streaming Information Sources. *Proceedings of the 29th International Conference on Very Large Data Bases* (pp. 285–296). VLDB Endowment. doi:10.1016/B978-012722442-8/50033-1

Wang, X., Tiropanis, T., & Davis, H. C. (2013, May 14). LHD: Optimising Linked Data Query Processing Using Parallelisation. *Proceedings of the WWW2013 Workshop on Linked Data on the Web*, Rio de Janeiro, Brazil.

Williams, G. T. (2008). Supporting identity reasoning in SPARQL using bloom filters. Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008).

Wilschut, A. N., & Apers, P. M. G. (1991). Dataflow Query Execution in a Parallel Main- Memory Environment. *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (pp. 68–77). IEEE Computer Society Press. doi:10.1109/PDIS.1991.183069

ENDNOTES

[1] Speedup of x compared to y (%) = (completion time of y - completion time of x) / (completion time of y) * 100

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

Damla Oguz received her BSc degrees in Software Engineering (2008) and Industrial Systems Engineering (2009) from Izmir University of Economics. She received her MSc degree in Computer Engineering from Izmir Institute of Technology in 2012. She is currently a PhD student under joint supervision between Paul Sabatier University and Ege University. Her current research interests are query optimization in large-scale distributed environments and linked data. She has been working as a research assistant in Department of Computer Engineering at Izmir Institute of Technology since December 2009.

Shaoyi Yin conducted her PhD at the University of Versailles, France, under an INRIA doctoral contract and defended in June 2011. She then worked as a post-doc at the University of Cergy-Pontoise, in ETIS Laboratory. Since September 2012, she works at Paul Sabatier University, in the Pyramid team of IRIT Laboratory, as an associate professor. Her current research interests mainly include query optimization in parallel and large-scale distributed environments.

Belgin Ergenç received the Diploma Degree in Computer Science from Middle East Technical University, Ankara, Turkey in 1983. She worked with different titles and responsibilities in IT industry during 1983 - 2000. She received the MS Degree in Computer Engineering from Izmir Institute of Technology, Turkey in 2002 and Ph.D. degree from Paul Sabatier University of Toulouse, France, in 2008 respectively. Since 2008, she is an associate professor in the Department of Computer Engineering at Izmir Institute of Technology, Turkey. Her main research interests are distributed databases and data mining. She likes reading, travelling, photography, cinema and jazz music. Her favorite sports are walking, swimming and pilates.

Abdelkader Hameurlain is full professor in Computer Science at Paul Sabatier University, Toulouse, France. He is a member of the Institute of Research in Computer Science of Toulouse (IRIT). His current research interests are in query processing and optimization in parallel and large-scale distributed environments, mobile databases, and database performance. Prof. Hameurlain has been the general chair of the International Conference on Database and Expert Systems Applications (DEXA'02, DEXA'2011 and DEXA'2017). He is co-editors in Chief of the International Journal "Transactions on Large-scale Data and Knowledge Centered Systems" (LNCS, Springer). He was guest editor of three special issues of "International Journal of Computer Systems Science and Engineering on "Mobile Databases", "Data Management in Grid and P2P Systems", and "Elastic Data Management in Cloud Systems".

Oguz Dikenelli received his BSEE degree in electric and electronic engineering from the Middle East Technical University, Ankara, Turkey in 1988 and the MS and PhD degrees in computer engineering from the Ege University, İzmir, Turkey in 1991 and 1995 respectively. He was a visiting researcher in the computer science department in Peen State and SMU, Texas, USA during 1993-1994. He is currently a professor in computer engineering department at Ege University. His research interests include agent-oriented software engineering, multi-agent systems and semantic data processing. Dr. Dikenelli is the author or co-author of more than 100 papers and edited 4 books describing his research and serves as a committee member in various national and international conferences.