Méthodes d'optimisation pour le traitement de requêtes réparties à grande échelle sur des données liées

Damla Oğuz

▶ To cite this version:

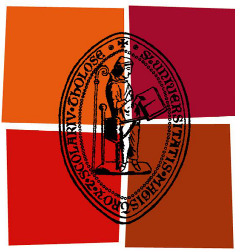
Damla Oğuz. Méthodes d'optimisation pour le traitement de requêtes réparties à grande échelle sur des données liées. Web. Université Paul Sabatier - Toulouse III, 2017. Français. NNT : 2017TOU30067 . tel-01820773

HAL Id: tel-01820773
https://theses.hal.science/tel-01820773

Submitted on 22 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*
Cotutelle internationale avec *l'Université Ege, Turquie*

Présentée et soutenue le *28/06/2017* **par :**

DAMLA OĞUZ

Méthodes d'optimisation pour le traitement de requêtes réparties à grande échelle sur des données liées

JURY

DJAMAL BENSLIMANE	Professeur à l'Université Claude Bernard Lyon 1, France	Rapporteur
ONUR DEMİRÖRS	Professeur à l'Université Technique du Moyen-Orient, Turquie	Rapporteur
LADJEL BELLATRECHE	Professeur à l'ENSMA Poitiers, France	Examinateur
ABDELKADER HAMEURLAIN	Professeur à l'Université Toulouse III, France	Directeur de Thèse
BELGIN ERGENÇ BOSTANOĞLU	Professeur Associé à l'Institut des Technologies d'Izmir, Turquie	Co-Directeur de Thèse
SHAOYI YIN	Maître de Conférences à l'Université Toulouse III, France	Encadrant

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)
Directeurs de Thèse :
Abdelkader HAMEURLAIN et *Belgin ERGENÇ BOSTANOĞLU*
Rapporteurs :
Djamal BENSLIMANE et *Onur DEMİRÖRS*

To my one and only beloved Kaya

Acknowledgments

Firstly, I would like to thank the institute director Prof. Dr. Michel DAYDÉ for welcoming me to IRIT. I would like to express my gratitude to the reviewers of this thesis, Prof. Dr. Djamal BENSLIMANE and Prof. Dr. Onur DEMİRÖRS, for their invaluable remarks. I would like to sincerely thank Prof. Dr. Ladjel BELLATRECHE for agreeing to be the examiner of this thesis.

PhD is a long and challenging period. I believe that this period has been a rewarding experience thanks to the people I had the opportunity to work with. I would like to express my sincere gratitude to my advisors Assoc. Prof. Dr. Belgin ERGENÇ BOSTANOĞLU and Prof. Dr. Abdelkader HAMEURLAIN for their continuous support, patience, and guidance. I would like to thank my co-advisor on the Turkish side, Prof. Dr. Oğuz DİKENELLİ for his valuable contributions to the thesis. I would also like to give my special thanks to my co-advisor on the French side, Dr. Shaoyi YIN, for her endless and priceless support, patience, and help in every step of this thesis. I am privileged to acquire the guidance of the mentioned distinguished advisors.

I would also like to thank The Scientific and Technological Research Council of Turkey (TÜBİTAK) for the financial support of 2214/B International Joint PhD Fellowship Programme which made this joint thesis possible.

I am grateful to Dr. Riad MOKADEM and Asst. Prof. Dr. Tolga AYAV for their

endless support during each registration term. I am also grateful to Prof. Dr. Franck MORVAN for sharing his office at IRIT Laboratory. I would like to thank Uras TOS, who took the similar steps during his PhD studies, for his friendship and support. I would specially like to thank Chiraz MOUMEN for her friendship during my Toulouse visits. I would also like to thank Burak YÖNYÜL and Emrah İNAN for their support in my studies.

I would like to extend my gratitude to my friends, colleagues, and professors at Izmir Institute of Technology, Ege University, and IRIT Laboratory. It has been a true pleasure to work in such a friendly environment.

I would like to thank all my family, starting with my mother Ayşenur DEMİRTAŞ, my father Erkan DEMİRTAŞ, and my twin sister Duygu DEMİRTAŞ GÜNER for supporting me throughout my whole life as well as in my graduate studies. I feel very lucky to have a family like them.

Finally, I would like to especially thank my husband Kaya OĞUZ for his love, patience, and encouragement in every aspect of my life. His everlasting support made this work possible.

Abstract

Optimization methods for large-scale distributed query processing on linked data

Linked Data is a term to define a set of best practices for publishing and interlinking structured data on the Web. As the number of data providers of Linked Data increases, the Web becomes a huge global data space. Query federation is one of the approaches for efficiently querying this distributed data space. It is employed via a federated query engine which aims to minimize the response time and the completion time. Response time is the time to generate the first result tuple, whereas completion time refers to the time to provide all result tuples.

There are three basic steps in a federated query engine which are data source selection, query optimization, and query execution. This thesis contributes to the subject of query optimization for query federation. Most of the studies focus on static query optimization which generates the query plans before the execution and needs statistics. However, the environment of Linked Data has several difficulties such as unpredictable data arrival rates and unreliable statistics. As a consequence, static query optimization can cause inefficient execution plans. These constraints show that adaptive query optimization should be used for federated query processing on Linked Data.

In this thesis, we first propose an adaptive join operator which aims to minimize the response time and the completion time for federated queries over SPARQL endpoints. Second, we extend the first proposal to further reduce the completion time. Both proposals can change the join method and the join order during the execution by using adaptive query optimization. The proposed operators can handle different data arrival rates of relations and the lack of statistics about them.

The performance evaluation of this thesis shows the efficiency of the proposed adaptive operators. They provide faster completion times and almost the same response times, compared to symmetric hash join. Compared to bind join, the proposed operators perform substantially better with respect to the response time and can also provide faster completion times. In addition, the second proposed operator provides considerably faster response time than bind-bloom join and can improve the completion time as well. The second proposal also provides faster completion times than the first proposal in all conditions. In conclusion, the proposed adaptive join operators provide the best trade-off between the response time and the completion time. Even though our main objective is to manage different data arrival rates of relations, the performance evaluation reveals that they are successful in both fixed and different data arrival rates.

Keywords: Distributed Query Processing, Query Optimization, Adaptive Query Optimization, Linked Data, Query Federation, Performance Evaluation

Résumé

Méthodes d'optimisation pour le traitement de requêtes réparties à grande échelle sur des données liées

Données Liées est un terme pour définir un ensemble de meilleures pratiques pour la publication et l'interconnexion des données structurées sur le Web. A mesure que le nombre de fournisseurs de Données Liées augmente, le Web devient un vaste espace de données global. La fédération de requêtes est l'une des approches permettant d'interroger efficacement cet espace de données distribué. Il est utilisé via un moteur de requêtes fédéré qui vise à minimiser le temps de réponse du premier tuple du résultat et le temps d'exécution pour obtenir tous les tuples du résultat.

Il existe trois principales étapes dans un moteur de requêtes fédéré qui sont la sélection de sources de données, l'optimisation de requêtes et l'exécution de requêtes. La plupart des études sur l'optimisation de requêtes dans ce contexte se concentrent sur l'optimisation de requêtes statique qui génère des plans d'exécution de requêtes avant l'exécution et nécessite des statistiques. Cependant, l'environnement des Données Liées a plusieurs caractéristiques spécifiques telles que les taux d'arrivée de données imprévisibles et les statistiques peu fiables. En conséquence, l'optimisation de requêtes statique peut provoquer des plans d'exécution inefficaces. Ces contraintes montrent que l'optimisation de requêtes adaptative est une nécessité pour le traitement de requêtes

fédéré sur les données liées.

Dans cette thèse, nous proposons d'abord un opérateur de jointure adaptatif qui vise à minimiser le temps de réponse et le temps d'exécution pour les requêtes fédérées sur les endpoints SPARQL. Deuxièmement, nous étendons la première proposition afin de réduire encore le temps d'exécution. Les deux propositions peuvent changer la méthode de jointure et l'ordre de jointures pendant l'exécution en utilisant une optimisation de requêtes adaptative. Les opérateurs adaptatifs proposés peuvent gérer différents taux d'arrivée des données et le manque de statistiques sur des relations.

L'évaluation de performances dans cette thèse montre l'efficacité des opérateurs adaptatifs proposés. Ils offrent des temps d'exécution plus rapides et presque les mêmes temps de réponse, comparé avec une jointure par hachage symétrique. Par rapport à bind join, les opérateurs proposés se comportent beaucoup mieux en ce qui concerne le temps de réponse et peuvent également offrir des temps d'exécution plus rapides. En outre, le deuxième opérateur proposé obtient un temps de réponse considérablement plus rapide que la bind-bloom join et peut également améliorer le temps d'exécution. Comparant les deux propositions, la deuxième offre des temps d'exécution plus rapides que la première dans toutes les conditions. En résumé, les opérateurs de jointure adaptatifs proposés présentent le meilleur compromis entre le temps de réponse et le temps d'exécution. Même si notre objectif principal est de gérer différents taux d'arrivée des données, l'évaluation de performance révèle qu'ils réussissent à la fois avec des taux d'arrivée de données fixes et variés.

Mot-clés: Traitement de requêtes distribuées, optimisation de requêtes, optimisation de requêtes adaptative, données liées, fédération de requêtes, évaluation de performances

Contents

1 Introduction **19**

1.1 Context 21

1.2 Query Federation 23

1.3 Problem Position 25

1.4 Contributions 27

1.5 Thesis Organization 29

2 State of the Art **31**

2.1 Introduction 33

2.2 Federated Query Processing on Linked Data 35

2.2.1 Data Source Selection 36

2.2.2 Join Methods 41

2.2.3 Query Optimization 46

2.2.4 Discussion on Query Federation on Linked Data 57

2.2.5 Challenges of Query Federation on Linked Data 59

2.3 Adaptive Query Optimization 63

2.3.1 Adaptive Query Optimization for Relational Databases 63

2.3.2 Adaptive Query Optimization for Query Federation on Linked Data . . . 65
2.4 Conclusion . . . 69

3 Optimization Methods for Query Federation on Linked Data 71
3.1 Introduction . . . 73
3.2 Adaptive Join Operator for Federated Queries . . . 74
3.2.1 Adaptive Join Operator for Single Join Queries . . . 76
3.2.2 Adaptive Join Operator for Multi-Join Queries . . . 80
3.3 Extended Adaptive Join Operator for Federated Queries . . . 84
3.3.1 Background . . . 85
3.3.2 Extended Adaptive Join Operator for Single Join Queries . . . 87
3.3.3 Extended Adaptive Join Operator for Multi-Join Queries . . . 92
3.4 Conclusion . . . 96

4 Performance Evaluation 99
4.1 Introduction . . . 101
4.2 Performance Evaluation of Adaptive Join Operator . . . 102
4.2.1 Performance Evaluation for Single Join Queries . . . 102
4.2.2 Performance Evaluation for Multi-Join Queries . . . 107
4.3 Performance Evaluation of Extended Adaptive Join Operator . . . 113
4.3.1 Performance Evaluation for Single Join Queries . . . 114
4.3.2 Performance Evaluation for Multi-Join Queries . . . 125
4.4 Conclusion . . . 132

5 Conclusion and Future Work 135
5.1 Thesis Review . . . 137

5.2 Future Work 140

Bibliography **141**

List of Figures

1.1 Federated query processing 26

3.1 Adaptive join operator for single join queries 77

3.2 Adaptive join operator for multi-join queries 81

4.1 AJO: Impact of data sizes for single join queries 103

4.2 AJO: Impact of data arrival rates when $card(R1) \ll card(R2)$ 105

4.3 AJO: Impact of data arrival rates when $card(R1) \gg card(R2)$ 106

4.4 AJO: Impact of data sizes for multi-join queries 109

4.5 AJO: Impact of data sizes when $card(R1) \ll card(R2) = card(R3)$ . . 110

4.6 AJO: Impact of data arrival rates when $card(R1) = card(R2) \gg card(R3)$112

4.7 EAJO: Impact of data sizes for single join queries 116

4.8 EAJO: Impact of data arrival rates when $card(R1) \ll card(R2)$ 118

4.9 EAJO: Impact of data arrival rates when $card(R1) \gg card(R2)$ 121

4.10 EAJO: Impact of bit vector size 124

4.11 EAJO: Impact of data sizes for multi-join queries 127

4.12 EAJO: Impact of data arrival rates when $card(R1) \ll card(R2) = card(R3)$ 129

4.13 EAJO: Impact of data arrival rates when $card(R1) = card(R2) \gg card(R3)$ 130

List of Tables

2.1 Data source selection methods in query federation 41

2.2 Join methods in query federation 46

2.3 Subquery building methods in query federation 50

2.4 Join ordering and join method selection in query federation 56

2.5 Comparison of Query Federation Engines 59

2.6 Comparison of adaptive query optimization in query federation 68

3.1 Intermediate results of Listing 3.1 86

3.2 Example relations R_i and R_j 90

4.1 Speedup of EAJO compared to AJO when $card(R1) \ll card(R2)$ and the data arrival rate of $R1$ is 2 Mbps 119

4.2 Speedup of EAJO compared to AJO when $card(R1) \ll card(R2)$ and the data arrival rate of $R1$ is 0.5 Mbps 120

4.3 Speedup of EAJO compared to AJO when $card(R1) \gg card(R2)$ and the data arrival rate of $R1$ is 2 Mbps 122

4.4 Speedup of EAJO compared to AJO when $card(R1) \gg card(R2)$ and the data arrival rate of $R1$ is 0.5 Mbps 122

4.5 The m/n and k combinations used for bloom filter 123

4.6 Speedup of EAJO compared to AJO when data arrival rates are fixed . 128

Chapter 1

Introduction

Abstract: This chapter introduces the context and motivations of the work presented in this thesis. We start with describing the context and then we explain our problem position. We present our proposals to the mentioned problem and discuss our contributions. Finally, we present the structure of the thesis.

Contents

1.1 Context **21**

1.2 Query Federation **23**

1.3 Problem Position **25**

1.4 Contributions **27**

1.5 Thesis Organization **29**

1.1 Context

The Web, which was proposed by Tim Berners-Lee, is one of the most important developments of 90s. Although the Web is an information space for humans, it is meaningless for machines since it consists of documents. In the early 2000s, Berners-Lee et al. (2001) proposed the Semantic Web in which information is given a well-defined meaning. In other words, the Semantic Web is an extended version of the Web which provides a data space for both humans and machines. It is often referred to as the Web of Data. In order to create such a global data space, the data should be opened, published, and related to one another according to some rules which are defined by Berners-Lee (2006). Publishing and connecting structured data on the Web in this way is defined as Linked Data. It also refers to the collection of interrelated data sources on the Web. In brief, the Semantic Web is the goal of providing both human-readable and machine-readable data, whereas Linked Data provides the means to reach that goal (Bizer et al., 2009).

As stated above, Linked Data makes the Web as a huge global data space which is referred to as the Semantic Web. Querying this distributed data space is one of the most important research problems. Therefore, we mainly focus on distributed query processing on Linked Data in this thesis.

Linked Data query processing infrastructure can be categorized as central repository and distributed repository, according to the data source location (Rakhmawati et al., 2013). In central repository infrastructure, all data from different data sources are aggregated in a single repository before query processing. In distributed repository infrastructure, the query is executed on the distributed data sources. Although central repository infrastructure provides efficient query processing, data is not always up-to-date and adding a new data source is difficult. On the other hand, data is more

up-to-date in distributed repository infrastructure. There are two main approaches for query processing based on distributed repository infrastructure which are link traversal (Hartig et al., 2009a) and query federation (Görlitz and Staab, 2011a).

Link traversal, also called follow-your-nose, can be simply defined as discovering potentially relevant data by following the links between data. Related data sources are discovered during the query execution without any data knowledge. One of the well-known examples of link traversal approach is SQUIN (Hartig et al., 2009b; Hartig, 2013). The data sources are RDF documents in this concept and the intermediate results are augmented with bindings for the common variables. The major advantage of link traversal is providing up-to-date results and using the potential of the Web by discovering data sources at run-time. However, this approach has some remarkable weaknesses. The results can change according to the starting point and a wrong starting point can increase intermediate results (Rakhmawati et al., 2013). Although some heuristic query planning methods are employed by Hartig (2011), the mentioned weaknesses cannot be solved. In other words, this approach cannot guarantee finding all results because the relevant data sources change according to the starting point.

The second approach for distributed repository infrastructure, query federation, is based on dividing a query into its subqueries and distributing them to the related data sources. These processes are performed with a federated query engine. The infrastructure is similar to mediation system architecture (Wiederhold, 1992), and thus the engine can also be called the mediator. There are two main advantages of query federation. The first one is providing up-to-date results and the other one is the capability of guaranteeing finding all results. On the other hand, queries are executed over SPARQL endpoints. For this reason, SPARQL endpoints of each relevant data source are required in order to execute a query. This can be accepted as a shortcoming

of this approach. However, (Rakhmawati et al., 2013) remarked that 68.14% of data sources provide their SPARQL endpoints and we think that this number is increasing day by day.

The common advantage of link traversal and query federation is providing up-to-date results due to executing queries on the actual data sources. However, link traversal does not guarantee complete results and has some performance problems. Because of these reasons, we turn our attention to the second approach.

In this chapter, we first introduce the query federation approach in Section 2. Then, in Section 3, we discuss the position of the problem that motivated us for the work presented in this thesis. In Section 4 and Section 5, we state the contributions of this thesis and we present the organization of the thesis, respectively.

1.2 Query Federation

Before presenting the query federation approach, we briefly introduce some main concepts of Linked Data which are used several times in the thesis.

- Resource Description Framework (RDF) is defined as a standard model for data interchange on the Web by W3 Consortium. RDF is also called as triple model since it has a subject–predicate–object structure. The data model for Linked Data is RDF.

- Triple patterns are similar to RDF triples except that each of subject, predicate, and object may be a variable.[1] Let s, p, o_1 denote a certain subject, predicate, and object respectively, and $?o_2$ is a variable object. $tp1$ = (s, p, o_1) is a RDF

[1] https://www.w3.org/TR/rdf-sparql-query/

triple, whereas $tp2 = (s, p, ?o_2)$ is a triple pattern. $tp1$ is a triple pattern with certainty, which does not contain any variables.

- A subquery is a set of triple patterns. Listing 1.1 shows a query which finds the director and the genre of movies directed by Italians (Haase et al., 2010). ⟨*?film dbpedia-owl:director ?director*⟩ is a triple pattern since *?film* and *?director* are variables. The federated query engine decides the set of triple patterns that composes each subquery.

- SPARQL is the query language for Linked Data.

- SPARQL endpoint is an HTTP based query processing service which enables both humans and machines to query a data source via SPARQL language.

```
SELECT ?film ?director ?genre WHERE {
    ?film dbpedia-owl:director ?director .
    ?director dbpedia-owl:nationality dbpedia:Italy .
    ?x owl:sameAs ?film .
    ?x linkedMDB:genre ?genre . }
```

Listing 1.1: Query example

The main idea of query federation is quite similar to mediator-wrapper architecture (Wiederhold, 1992). In mediator-wrapper architecture, firstly the relevant data sources are selected, secondly the query is divided into its subqueries, thirdly subqueries are executed on the distributed data sources through their wrappers, and finally the results of the subqueries are combined by a mediator. Thus, a wrapper for each data source and a mediator are needed in this architecture. The architecture of query federation is similar to the mediator-wrapper architecture in integrating the information from

different data sources via a mediator. However, they are different from each other in accessing the data sources. In the mediator-wrapper architecture, wrappers are used to access the datasets due to the heterogeneous data models, whereas SPARQL endpoints are used to access the data sources without wrappers due to the common data model (RDF) in query federation. Each query is decomposed into subqueries and directed to the SPARQL endpoints of the selected data sources to be executed. The results of the subqueries are aggregated and finally returned to the user.

Figure 1.1 summarizes the working principal of a federated query engine which includes three main tasks as follows: i) data source selection, ii) query optimization, and iii) query execution. In data source selection, the relevant data sources for each triple pattern or set of triple patterns of a query are determined. The subqueries and intermediate results are transmitted over the Web of Data. Thus, query optimization is substantially important in query federation. The fundamental responsibilities of query optimization are grouping the triple patterns, deciding the join strategy, and ordering the triple patterns. Query execution part is dedicated to the execution of the query operators defined by the optimizer and preparation of the result set.

1.3 Problem Position

In the beginning of the thesis, we survey query processing approaches used in Linked Data and we turn our attention to query federation which is one of these approaches. It provides up-to-date results and has the capability of guaranteeing to find all results. As mentioned previously, query federation is performed with a federated query engine which has three basic steps as data source selection, query optimization, and query execution. Since the first step is data source selection, initially researchers have

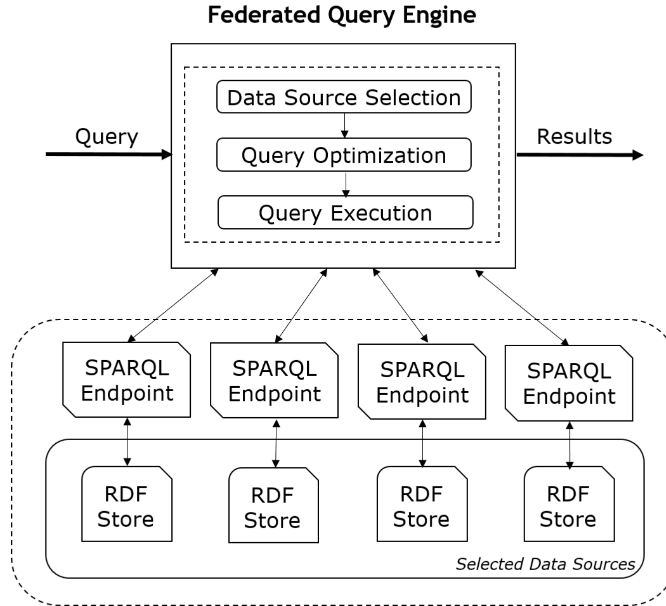


Figure 1.1: Federated query processing

mainly focused on data source selection. We aim to contribute to the subject of query optimization for query federation in this thesis.

There are naive studies about query optimization that generally focus on traditional query optimization, also called static query optimization (Selinger et al., 1979). It generates query plans before the execution and needs statistics to estimate the size of 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 (Deshpande et al., 2007) should be used in this unpredictable environment.

The objective of query optimization in federated query engines is to minimize both the response time and the completion time. Response time refers to the time to generate the first result tuple, whereas completion time refers to 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 in distributed environments, the main objective of federated query engines can be stated as to minimize the communication cost. These facts also show the importance of adaptive query optimization for query federation over Linked Data.

In conclusion, adaptive query optimization deals with unforeseen variations of run-time environment. In our domain, the run-time environment is the Web of Data, and the main objective is to minimize the response time and the completion time. Thus, adaptive query optimization is a need to manage unpredictable data arrival rates and missing statistics to minimize the response time and the completion time. Acosta et al. (2011) and Lynden et al. (2011) have shown that response time and completion time can be decreased 5-6 times and in average by using adaptive query optimization. These results show the significance of the problem. For these reasons, in this thesis, we focus on query optimization problem, more specifically, adaptive query optimization in query federation on Linked Data.

1.4 Contributions

We begin this thesis by surveying query processing approaches used in Linked Data and focus on query federation which is one of these approaches. Following this survey, we first propose a join operator which uses adaptive query optimization for federated queries over SPARQL endpoints. Second, we present an extended version of our first join operator. We present these operators for both single join and multi-join queries. The contributions of this thesis are listed as follows:

- A literature survey about federated query processing on Linked Data (Oguz et al., 2015): We synthesize the data source selection, join methods, and query opti-

mization methods of existing query federation engines. We also present the major challenges of federated query processing on Linked Data.

- Adaptive Join Operator (Oguz et al., 2016): As explained in the previous section, the objective of query optimization in federated query engines is to minimize the response time and the completion time. The first one is the time to provide the first result tuple, while the second one is the time to provide all result tuples. Adaptive join operator aims to manage different data arrival rates of relations in order to minimize both the response time and the completion time. It is able to change the join method during the execution according to remaining time estimations. Thus, it manages different data arrival rates of relations. To the best of our knowledge, there is not any study that proposes an adaptive join operator which aims to reduce both the response time and the completion time for federated queries over SPARQL endpoints. The results of the performance evaluation show that adaptive join operator provides both optimal response time and completion time for single join queries and multi-join queries. The proposed operator provides the best trade-off between the response time and the completion time in both fixed and different data arrival rates.

- Extended Adaptive Join Operator (Oguz et al., In press): Communication time has the highest effect on response time and completion time in distributed environments. Thus, we can say that the main goal of query optimization in query federation engines is to minimize the communication cost. To further reduce the communication time in completion time, we propose an extended version of the adaptive join operator through adding another join method to our candidate join methods. The new candidate join method employs a space efficient data

structure to minimize the communication cost. In conclusion, we improve our previous proposal in order to further reduce the completion time. Performance evaluation shows that the extended join operator provides optimal response time. Furthermore, the proposed operator further reduces the completion time and it has the adaptation ability to different data arrival rates.

1.5 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the steps in federated query processing, gives a detailed synthesis of studies related to query federation approach and discusses the major challenges of federated query processing on Linked Data. This chapter also analyzes the studies in relational databases and query federation which use adaptive query optimization. In Chapter 3, we concentrate on the adaptive query optimization problem in query federation which is one of the mentioned challenges in the previous chapter. We first propose an adaptive join operator for single join queries and multi-join queries for federated queries over SPARQL endpoints. Then, we extend our previous proposal to further reduce the completion time. Chapter 4 covers the results and discussions on performance evaluation of the work presented in the previous chapter. Finally, conclusions and future work are discussed in Chapter 5.

Chapter 2

State of the Art

Abstract: This chapter provides the literature survey about query processing on Linked Data. We initially give an overview of query processing approaches on Linked Data and then focus on the query federation approach. We introduce the main steps in this approach, and provide a detailed insight on them by comparing the current federated query engines. Furthermore, we present a qualitative comparison of these engines and discuss the major challenges of federated query processing on Linked Data. Then, we continue with the literature review of adaptive query optimization for relational databases. Finally, we focus on adaptive query optimization in query federation.

Contents

2.1 Introduction **33**

2.2 Federated Query Processing on Linked Data **35**

2.2.1 Data Source Selection 36

2.2.2 Join Methods 41

2.2.3 Query Optimization 46

2.2.4 Discussion on Query Federation on Linked Data 57

2.2.5 Challenges of Query Federation on Linked Data 59

2.3 Adaptive Query Optimization **63**

2.3.1 Adaptive Query Optimization for Relational Databases . . . 63

2.3.2 Adaptive Query Optimization for Query Federation on Linked Data 65

2.4 Conclusion **69**

2.1 Introduction

Bizer et al. (2009) defines Linked Data as a set of best practices for publishing and connecting structured data on the Web. These practices are known as Linked Data principles. In order to contribute to the Semantic Web, the data should be published and connected to others according to the Linked Data principles. The resulting form of the Web data is also referred to as Linked Data (Hartig, 2014). The Linked Data principles defined by Berners-Lee (2006) are as follows:

1. *Use URIs as names for things.*
2. *Use HTTP URIs so that people can look up those names.*
3. *When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).*
4. *Include links to other URIs, so that they can discover more things.*

These principles clearly show that Uniform Resource Identifier (URI) (Berners-Lee et al., 2005), Hypertext Transfer Protocol (HTTP) (Fielding et al., 1999), Resource Description Framework (RDF) (Klyne and Carroll, 2004), and RDF Query Language (SPARQL) (Harris et al., 2013) are the building stones of Linked Data. The first and the second rules declare that an entity should be identified via an HTTP URI scheme in order to be served as a globally unique identifier, and in order to provide access to a structured data representation of it. The third rule presents the data model of Linked Data and the query language for this data model which are RDF and SPARQL, respectively. RDF provides a graph-based data model that describes things including their relationships with other things. They are represented as a number of triples and each triple has three parts which are subject, predicate, and object. Thus, RDF

is referred to as triple model. Finally, the fourth rule enforces to connect data with others in order to create Web of Data (Berners-Lee, 2006; Bizer et al., 2009; Hartig and Langegger, 2010).

Linking Open Data project[1] is the most known performer of the Linked Data principles (Bizer et al., 2009). Its goal is to extend the Web of Data by identifying the existing open data sources as RDF and setting RDF links between the data items from different data sources. The number of data sources related to that project have been increased from 12 to 1,139 as of May 2007 to January 2017[2]. There are well-known organizations among the participants, such as BBC (Kobilarov et al., 2009), the New York Times[3], the UK government (Shadbolt et al., 2012), and the Library of Congress (Ford, 2013). DBpedia (Auer et al., 2007), Linked Movie Database (Hassanzadeh and Consens, 2009), and MusicBrainz (Swartz, 2002) are also some of the important participants.

To conclude, a large number of data providers publish and connect their structured data on the Web as Linked Data. Thus, the Web of Data becomes a global data space. In other words, Linked Data creates a global and distributed data space on the Web. Querying this huge data space is one of the important research questions in this research topic. Link traversal and query federation are the two approaches for querying this huge data space on the distributed data sources. Link traversal (Hartig et al., 2009b) finds the related data sources during the query execution, whereas query federation (Görlitz and Staab, 2011a) selects the related data sources before the execution. Link traversal has the disadvantage of not guaranteeing complete results. For this reason, we concentrate on query federation.

[1]https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
[2]http://lod-cloud.net/
[3]http://data.nytimes.com/

In this chapter, we review the literature of query federation in detail to understand the state of the art in this approach. In Section 2.2, we introduce the query federation approach and synthesize the existing data source selection methods, join methods, and query optimization methods in federated query processing through surveying the promising federated query engines. We also provide a qualitative comparison of these studies. In addition, we discuss the challenges of federated query processing on Linked Data. In Section 2.3, we focus on adaptive query optimization (Deshpande et al., 2007) which is one the challenges mentioned in the previous section. We first review adaptive query optimization for relational databases. Then, we focus on the studies in Linked Data which use adaptive query optimization. Finally in Section 2.4, we present our conclusions about the literature survey and we introduce the ideas motivating the work presented in Chapter 3.

2.2 Federated Query Processing on Linked Data

Federated query processing, which is also called query federation, is based on dividing a query into its subqueries and distributing the query execution of them over 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 performed with a federated query engine. The engine performs three main steps which are data source selection, query optimization, and query execution. Data source selection selects the relevant data sources for each triple pattern of a query. Query optimization is responsible for grouping the triple patterns, deciding the join method, and ordering the triple patterns. Query optimization is substantially important in query federation, because the subqueries and the intermediate results are transmitted

over the Web of Data. The last step, query execution, is dedicated to the execution of the query operators defined by the optimizer and preparation of the result set.

In the following subsections, we synthesize data source selection methods, join methods, and query optimization in query federation by surveying the promising engines in the literature.

2.2.1 Data Source Selection

We classify the data source selection methods as follows: (i) predicate-based selection, (ii) type-based selection, (iii) rule-based selection, and (iv) SPARQL ASK queries. All these methods except the last method need metadata catalogs. For this reason, we first discuss the metadata catalogs in query federation and then propose our classification.

Metadata catalogs can be defined as SPARQL endpoint descriptions that describe various properties about the data source belonging to this endpoint. The existing query federation engines use three types of metadata catalogs: (i) service descriptions (Quilitz and Leser, 2008), (ii) VoID (Vocabulary of Interlinked Datasets) descriptions (Alexander and Hausenblas, 2009), and (iii) list of predicates. We want to remark that dataset and data source are used interchangeably.

- *Service descriptions*: A service description provides metadata about the RDF data and cover some statistical information such as total triples and number of triples with a predicate. In other words, a service description specifies the information about the data source, which means a set of RDF triples, that is published by a single provider.

- *VoID descriptions*: VoID descriptions are similar to service descriptions that are used to provide metadata about the RDF data and cover some statistics about

it. Furthermore, there is another concept in VoID descriptions which is called linkset. A linkset describes a set of RDF triples where all subjects refer to one dataset and all objects belong to another dataset (Alexander and Hausenblas, 2009). Thus, VoID descriptions can be used to describe the metadata of RDF datasets with the interlinking to other datasets. Moreover, statistics about the datasets can be defined in VoID descriptions as in service descriptions. Number of triples and number of instances of a class or property are some examples of the statistics here. Cyganiak et al. (2011) proposed a VoID guide to data publishers and consumers. Besides, Charalambidis et al. (2015a) proposed an extension of VoID descriptions which introduces new concepts in order to provide more detailed descriptions.

- *List of predicates*: List of predicates is also used as a metadata catalog. Although they are useful in order to decide the relevant data sources of a query, they do not include statistical information about the data.

As mentioned in the beginning of this section, according to our classification, the data source selection methods in query federation are divided into four basic categories: predicate-based selection, type-based selection, rule-based selection, and SPARQL ASK queries.

- *Predicate-based selection:* It is based on selecting the relevant data sources of a triple pattern by matching its predicate with the covered predicates in the metadata catalog.

- *Type-based selection:* This type of data source selection uses the type definitions (*rdf:type*) in the metadata catalogs in order to select the relevant data sources.

- *Rule-based selection:* This method selects the relevant data sources according to defined rules which are generated by analyzing the relations between the triple patterns of a query. First two categories and this category are not mutually disjoint, as rule-based selection includes predicate-based and type-based selections.

- *SPARQL ASK queries:* A SPARQL ASK query returns a boolean indicating whether a query pattern matches or not[4]. Thus, data sources of a query can be selected by sending SPARQL ASK queries to the candidate endpoints. If the result of the query is TRUE, this data source is selected as a relevant data source.

DARQ (Quilitz and Leser, 2008) uses service descriptions as metadata catalogs which must be generated before the query execution. The engine employs predicate-based data source selection. Hence it compares the predicate of a triple pattern with the defined predicates of each service description. Therefore, the engine cannot support unbound predicate triple patterns.

SPLENDID (Görlitz and Staab, 2011b) uses VoID descriptions as metadata catalogs in data source selection. Data sources are indexed for every predicate and type by using VoID statistics. However, the statistics in VoID descriptions can be insufficient to select a triple pattern's relevant data source or data sources. This situation exists especially for the triples with common predicates such as *rdfs:label.* Since almost all datasets use this predicate, SPLENDID sends SPARQL ASK queries for the triple patterns with bound variables which are not covered in VoID descriptions. All data sources are selected for the triple patterns which have unbound predicates. Semagrow (Charalambidis et al., 2015b) uses VoID descriptions and SPARQL ASK queries in data source selection. The authors stated that Semagrow's data source selection is

[4]https://www.w3.org/TR/rdf-sparql-query/

pattern-wise like SPLENDID without detailed explanation. For this reason, we accept that its data source selection method is the same with SPLENDID's.

LHD (Wang et al., 2013) is another query engine that uses VoID descriptions together with SPARQL ASK queries in data source selection. It first uses VoID descriptions and then sends SPARQL ASK queries to refine the selected data sources. Its data source selection is based on predicates as DARQ. However, it can support unbound predicates without eliminating irrelevant data sources as SPLENDID.

WoDQA (Akar et al., 2012; Yönyül, 2014) also uses VoID descriptions and SPARQL ASK queries in data source selection. Akar et al. (2012) proposed different rules based on query pattern analysis, because they think that predicate-based and type-based selections are not enough in order to eliminate all irrelevant data sources due to having common predicates or types. These rules include three perspectives which are IRI-based analysis, linking analysis, and shared variable analysis. IRI-based analysis selects the relevant data sources by matching the IRIs in the triple pattern with the *void:uriSpace* and *void:vocabulary* properties of VoID descriptions. Therefore, IRI-based analysis includes the predicate-based and type-based selection methods. By this means, WoDQA does not only selects the data sources according to the predicates or types involved in a query, it considers all the IRIs in a query. In linking analysis, WoDQA takes into consideration the linkset definitions in the VoID descriptions. Lastly, in the shared variables analysis, WoDQA considers that triple patterns with shared variables can affect their related data sources. In other words, shared variables analysis aims to eliminate the irrelevant data sources.

List of predicates can also be used as a metadata catalog, as stated previously. ANAPSID (Acosta et al., 2011) keeps a list of predicates, the execution timeout property of the endpoint, and the statistics as a metadata catalog. The endpoints' execu-

tion timeouts and statistics are collected by an adaptive sampling technique (Blanco et al., 2012; Vidal et al., 2010) or they can be collected during the query execution. ANAPSID uses predicate-based selection and chooses the endpoints whose timeouts are longer than the estimated execution time of triple patterns. However, the details are not given in their publication so it is not clear how ANAPSID estimates the triple patterns' execution times. ADERIS (Lynden et al., 2010, 2011) also uses list of predicates as metadata catalogs and employs predicate-based selection. It sends SPARQL SELECT queries with DISTINCT keyword to each endpoint to find out the unique predicates. Besides, ADERIS adds data sources manually when it is impossible to do that automatically[5].

FedX (Schwarte et al., 2011) sends SPARQL ASK queries for each triple pattern of a query in order to decide if it can be answered by the endpoint or not. It also caches the relevance of each triple pattern with each data source in order to minimize the SPARQL ASK queries.

Table 2.1 shows the data source selection methods in the existing federated query engines. FedX (Schwarte et al., 2011) just sends SPARQL ASK queries in order to select the data sources. Although SPLENDID (Görlitz and Staab, 2011b) uses VoID descriptions to select the data sources based on predicates and types, it sends SPARQL ASK queries when the descriptions cannot help to select the relevant data sources. DARQ (Quilitz and Leser, 2008) selects the data sources based on predicates by using service descriptions. However, it does not send SPARQL ASK queries when the service description fail to select the related data sources. LHD (Wang et al., 2013) employs predicate-based selection via VoID descriptions and sends SPARQL ASK queries in order to eliminate the irrelevant data sources. ANAPSID (Acosta et al., 2011) and

[5]http://code.google.com/p/sparql-aderis/

Table 2.1: Data source selection methods in query federation

	Predicate-based selection	Type-based selection	Rule-based selection	SPARQL ASK queries
DARQ	✓			
FedX				✓
SPLENDID	✓	✓		✓
ANAPSID	✓			
ADERIS	✓			
LHD	✓			✓
WoDQA			✓	✓
Semagrow	✓	✓		✓

ADERIS (Lynden et al., 2010, 2011) use predicate-based selection as DARQ. However, ANAPSID also considers the execution timeout information of endpoints as well. Different from other engines, WoDQA (Akar et al., 2012; Yönyül, 2014) aims to eliminate all irrelevant data sources and employs rule-based selection which includes predicate-based and type-based selections. It also uses SPARQL ASK queries.

In conclusion, data source selection is a difficult task without metadata catalogs. In this case, SPARQL ASK queries are used in order to select the relevant data sources. In addition, Saleem et al. (2016) stated that caching the results of the SPARQL ASK queries greatly reduces the data source selection time.

2.2.2 Join Methods

The second step in federated query processing is query optimization which covers subquery building, join method selection, and join ordering. In order to improve the coherence of the thesis, we present the join methods in this subsection and we will discuss the query optimization in the following subsection.

Join methods in the existing engines can be categorized as follows: (i) bind join,

(ii) nested loop join, (iii) merge join, (iv) hash join, (v) symmetric hash join, and (vi) multiple hash join.

Bind Join

Bind join (Haas et al., 1997) passes the bindings of the intermediate results of the outer relation to the inner relation in order to filter the result set. It is substantially efficient when the intermediate results are small.

Bind join, which is also called bound join, is commonly used by federated query engines. Schwarte et al. (2011) proposed a bind join technique for FedX which uses SPARQL UNION[6] constructs to group a set of mappings in a subquery to be sent to the relevant data sources in a single remote request. WoDQA (Akar et al., 2012; Yönyül, 2014) uses bind join as well. Different from FedX, WoDQA employs bind join method with SPARQL FILTER[7] expression. In addition, SPARQL 1.1 Query Language[8] proposes SERVICE[9] keyword to explicitly execute certain subqueries on different SPARQL endpoints, and WoDQA takes the advantage of SERVICE keyword in its bind join. Charalambidis et al. (2015b) tested bind join with both UNION and VALUES[10] expressions for Semagrow. Although bind join with UNION expression requires additional processing in order to map the binding variables and their original names, the authors stated that it provides faster completion time than VALUES expression with the query they tested. Semagrow employs UNION expressions in a parallel fashion.

DARQ (Quilitz and Leser, 2008), ANAPSID (Acosta et al., 2011), ADERIS (Lynden

[6]https://www.w3.org/TR/rdf-sparql-query/#alternatives
[7]https://www.w3.org/TR/sparql11-query/#expressions
[8]https://www.w3.org/TR/sparql11-query/
[9]https://www.w3.org/TR/2013/REC-sparql11-service-description-20130321/
[10]https://www.w3.org/TR/sparql11-query/#inline-data

et al., 2011), SPLENDID (Görlitz and Staab, 2011b), and LHD (Wang et al., 2013) use bind join as well. We will discuss their usage later. Different from others, DARQ employs bind join when the data sources have limitations on access patterns (Florescu et al., 1999). Data sources with limited access patterns need some variables in a query to be bound in order to answer the query (Quilitz and Leser, 2008). For this reason, DARQ keeps the definition of limitations on access patterns in service descriptions.

Nested Loop Join

Nested loop join, as understood from its name, performs two nested loops over the relations. The inner relation is scanned for every binding in the outer relation while the bindings which provide the join condition are included in the result. Nested loop join is used by DARQ (Quilitz and Leser, 2008) when there is no limitation on access patterns. ADERIS (Lynden et al., 2010, 2011) applies index nested loop join method in query execution which uses an index on join attributes. Hence it provides an efficient access path for the inner relation.

As mentioned previously, WoDQA (Akar et al., 2012; Yönyül, 2014) uses bind join. However, it employs nested loop join in order to join the intermediate results of the relations locally. In other words, WoDQA uses nested loop join as a complementary part of bind join.

Merge Join

Merge join is based on merging two sorted relations on the join attribute. Hence this method needs both relations sorted and the join type should be an equi-join that uses only equality comparisons on the join attribute. Consider two relations with n_1 and n_1 tuples, respectively. The cost of nested loop join is proportional to $n_1 * n_2$, while

the cost of merge join is proportional to $n_1 + n_2$. Besides, the cost of sorting n pages is proportional to n log n. As a result, merge join is useful when there is an equi-join and when the relations are previously sorted. In general, sorting the relations and employing merge join is efficient when the cardinalities of relations are high (Ozsu and Valduriez, 2011). Semagrow (Charalambidis et al., 2015b) can employ merge join method besides bind join. It calculates the costs of both join methods and chooses the method with the lower cost.

Hash Join

Hash join is another join method used in federated query processing. It consists two phases. A hash table of one of the relations, generally the relation with the lower cardinality, is created in the first phase. In the second phase, the other relation's tuples are read, hashed and compared with the values in the hash table. These phases are also referred to as build phase and probe phase, respectively. A result tuple is generated when a match is found.

SPLENDID (Görlitz and Staab, 2011b) and LHD (Wang et al., 2013) use hash join which requests the results of the join argument in parallel and joins them locally. Although hash join is a symmetric join method conceptually, it is asymmetric in its operands (Wilschut and Apers, 1991).

Symmetric Hash Join

Symmetric hash join (Wilschut and Apers, 1991) is one of the earliest symmetric join algorithms. It supports pipelining in parallel database systems by maintaining a hash table for each relation. In other words, symmetric hash join creates two hash tables instead of generating single hash table as in hash join method. Thus, symmetric hash

join is a non-blocking join method which produces the output of tuples as early as possible. When a tuple arrives from a relation, it is probed in the other relation's hash table. Besides, the tuple is added to its own hash table to be used later in the process.

Double pipelined hash join (Ives et al., 1999) and XJoin (Urhan and Franklin, 2000) are the extended versions of symmetric hash join. Different from symmetric hash join, double pipelined hash join adapts its execution when the memory is insufficient and XJoin moves some parts of hash tables to the secondary storage when the memory is full.

Acosta et al. (2011) proposed a non-blocking join method, called adaptive group join (agjoin), which is based on symmetric hash join and XJoin. By this means, ANAPSID can produce results even when an endpoint becomes blocked and can hide delays from users. The authors also proposed another join method called adaptive dependent join (adjoin) which is an extended version of dependent join (Florescu et al., 1999). It sends requests to the data sources in an asynchronous fashion and hides delays from the user. In other words, it sends the request to the second data source when tuples from the first source are received. Therefore, adjoin can be accepted as bind join, because it needs the bindings in order to answer the query. Both agjoin and adjoin flush to the secondary memory when the memory is full as XJoin does.

Multiple Hash Join

LHD (Wang et al., 2013) uses multiple hash tables in order to integrate subqueries in parallel. The result of a relation is stored in its hash table and it is probed against the hash tables of other relations. Although using multiple hash tables is similar to multi-way symmetric hash join (Viglas et al., 2003), their operations are different. LHD uses these hash tables in order to execute the subqueries in a parallel fashion. Multi-way

Table 2.2: Join methods in query federation

	Bind join	Nested loop join	Merge join	Hash join	Symmetric hash join	Multiple hash join
DARQ	✓	✓				
FedX	✓					
SPLENDID	✓			✓		
ANAPSID	✓				✓	
ADERIS	✓	✓				
LHD	✓					✓
WoDQA	✓					
Semagrow	✓		✓			

symmetric hash join creates and uses them as the tuples from the relations arrive.

Besides, LHD employs bind join when pre-computed bindings are used. It separates the input bindings via a hash table on the dependent variable. If there is only one binding in the query, the variables in the query is replaced by the values of the binding. Otherwise, the bindings are specified with VALUES[11] syntax.

Table 2.2 shows the join methods used by federated query engines. As the table shows, bind join is the most popular join method among the federated query engines. Different from others, ANAPSID (Acosta et al., 2011) uses a non-blocking join method which is an extended version of symmetric hash join and XJoin. However, it uses its own data structure instead of hash tables.

2.2.3 Query Optimization

In this subsection we will discuss the query optimization methods in query federation. The goal of query optimization in federated query processing is to minimize the response time and the completion time which include communication time, I/O time, and CPU

[11] https://www.w3.org/TR/sparql11-query/#inline-data

time. The communication time dominates the others and it is directly proportional to the amount of intermediate results. Join method and join order affect the number of intermediate results. Therefore, join method selection and join ordering are the two essential parts of query optimization in federated query processing. In addition, the number of sent HTTP requests to the SPARQL endpoints affects the communication time as well. For this reason, grouping the appropriate triple patterns and sending them together to the related endpoint is important in order to reduce the communication time.

Consequently, query optimizer of a federated query engine covers three main decisions which are subquery building, join method selection, and join ordering. Following the query optimization, the last step in federated query processing is the query execution in which subqueries are executed over the SPARQL endpoints of the selected data sources according to the decisions made in query optimization. In the following of this section, we will discuss these decisions.

2.2.3.1 Subquery Building

A subquery of a SPARQL query comprises a set of triple patterns. Subquery building refers to grouping the triple patterns of a query in order to decrease the number of HTTP requests and intermediate results. We classify the subquery building methods used in federated query engines as follows: (i) exclusive grouping, (ii) exclusive grouping considering shared variables, and (iii) *owl:sameAs* grouping. We first define these methods and then explain their roles in the existing engines.

- *Exclusive grouping:* Exclusive grouping is a heuristic which groups the triple patterns if they have one and only one relevant data source. The grouped triple patterns are called exclusive groups. In other words, the triple patterns in an

exclusive group must refer to a single data source. This heuristic aims to reduce both the HTTP requests and the intermediate results.

- *Exclusive grouping considering shared variables:* This heuristic is an extended version of exclusive grouping. It creates different exclusive groups for the triple patterns without shared variables. Exclusive grouping method can group triple patterns which do not have shared variables, hence it causes redundant intermediate results.

- *owl:sameAs grouping:* Consider <*tp1 = ?x foaf:knows ?y .*> and <*tp2 = ?y owl:sameAs ?z*> (Schwarte et al., 2011). This method creates a subquery for the triple pattern which has *owl:sameAs* predicate with an unbound subject variable (*tp2*) and the triple pattern with the same unbound variable (*tp1*). It is used when there is an assumption that this predicate is used in order to indicate the internal resources of a dataset.

The idea behind the *exclusive grouping* was proposed by Quilitz and Leser (2008) for DARQ. However, the method was titled as *exclusive grouping* by Schwarte et al. (2011) for FedX. Although ANAPSID (Acosta et al., 2011) does not use the name of *exclusive grouping*, it groups the triple patterns which refer to the same endpoint. SPLENDID (Görlitz and Staab, 2011b) uses both *exclusive grouping* and *owl:sameAs grouping*. The assumption about the *owl:sameAs grouping* here is that all data sources describe *owl:sameAs* links for their data. This grouping can be employed when third party datasets with external *owl:sameAs* links do not exist in the federation. Although Semagrow (Charalambidis et al., 2015b) uses *exclusive grouping* as well, it is a configuration option which can be disabled. WoDQA (Akar et al., 2012; Yönyül, 2014) is the only engine which uses *exclusive grouping considering shared variables*.

As presented in data source selection, ADERIS (Lynden et al., 2010, 2011) generates metadata catalogs which cover distinct predicate values of each data source. ADERIS utilizes from them in subquery building. It groups the subqueries if their predicates are covered in the same data source. The main idea of this grouping is the same with *exclusive grouping*.

In conclusion, there are three methods for subquery building which are *exclusive grouping*, *exclusive grouping considering shared variables* and *owl:sameAs grouping*. Although some engines group the triple patterns which refer to the same data source, they do not name this method as *exclusive grouping*. On the other hand, *owl:sameAs grouping* is used when there is an assumption that this predicate is used for the resources of one dataset. Each triple pattern of a query is accepted as a subquery without using these methods.

Table 2.3 shows the subquery building methods in query federation. In order to decrease the HTTP requests, DARQ (Quilitz and Leser, 2008), FedX (Schwarte et al., 2011), SPLENDID (Görlitz and Staab, 2011b), ANAPSID (Acosta et al., 2011), ADERIS (Lynden et al., 2010, 2011), and Semagrow (Charalambidis et al., 2015b) use *exclusive grouping*, whereas WoDQA (Akar et al., 2012; Yönyül, 2014) employs *exclusive grouping considering shared variables* with the aim of decreasing the redundant intermediate results as well. SPLENDID (Görlitz and Staab, 2011b) uses *owl:sameAs grouping* and *exclusive grouping* together. However, *owl:sameAs grouping* cannot be used when other datasets use *owl:sameAs* predicate to define that the resource in their datasets indicates to the same resource in other datasets. LHD (Wang et al., 2013) does not group the triple patterns, it sends them in a parallel fashion.

Table 2.3: Subquery building methods in query federation

	Exclusive grouping	Exclusive grouping shared variables	*owl:sameAs* grouping
DARQ	✓		
FedX	✓		
SPLENDID	✓		✓
ANAPSID	✓		
ADERIS	✓		
WoDQA		✓	
Semagrow	✓		

2.2.3.2 Join Ordering

Ibaraki and Kameda (1984) stated that finding an optimization cost for a query is admitted as computationally intractable. Starting from this point of view, Gardarin and Valduriez (1990) specified that heuristics are necessary for optimizing the cost functions. Due to huge and distributed data space, query processing on Linked Data is a difficult task. Thus, using heuristic methods for join ordering in federated query processing is an expected case. FedX (Schwarte et al., 2011) and WoDQA (Akar et al., 2012; Yönyül, 2014) employ various heuristics for join ordering. We name these heuristics as follows:

- *Free variables heuristic (FVH):* Considers the number of free and bound variables. The number of free variables of triple patterns and groups are counted with considering the already bound variables from the earlier iterations. In other words, the free variables which have become bound from the earlier iterations are accepted as bound variables.

- *Exclusive group priority heuristic (EGPH):* Gives priority to exclusive groups which are presented in Section 2.2.3.1.

- *Position and type based selectivity heuristic (PTSH):* Calculates the heuristic selectivity value of each triple pattern as multiplying the calculated coefficients of each node according to their positions with the calculated coefficients of each node according to their types.

- *Shared variables heuristic (SVH):* Reorders the join order by considering the shared variables between triples patterns.

FedX (Schwarte et al., 2011) orders both triple patterns and groups of triple patterns by using *free variables heuristic* and *exclusive group priority heuristic*. Triple patterns and groups are chosen with the lowest cost iteratively. WoDQA (Akar et al., 2012; Yönyül, 2014) orders the triple patterns of each query by using *position and type based selectivity heuristic* after creating the exclusive groups of each query. The coefficient of position and types are assigned according to their selectivities. It considers that subjects are more selective than objects, and objects are more selective than predicates. A similar strategy is used by Stocker et al. (2008) for the Jena ARQ optimizer in which they categorize this estimation as heuristics without pre-computed statistics. Although (Stocker et al., 2008) state that there are more triples matching with a predicate than a subject or an object in a typical data source, they specify that making a distinction between subject and object is more difficult. On the other hand, WodQA orders the selectivities of types as URIs, literals, and variables.

After ordering the triple patterns in an exclusive group by employing *position and type based selectivity heuristic*, WoDQA employs *shared variables heuristic* for ordering exclusive groups. The triple patterns which do not have shared variables, are changed with the next triple pattern to process the related joins as early as possible. After ordering the triple patterns of each exclusive group, WoDQA orders the exclusive groups. It calculates the mean selectivity of each group by using *position and type based selec-*

tivity heuristic. Lastly, this order is updated by employing *shared variables heuristic* for exclusive groups. The exclusive group which has more shared variables than the consequent group, moves up in the order of exclusive groups. The aim of ordering the exclusive groups is to decrease intermediate results as well.

DARQ (Quilitz and Leser, 2008), SPLENDID (Görlitz and Staab, 2011b), ADERIS (Lynden et al., 2010, 2011), LHD (Wang et al., 2013), and Semagrow (Charalambidis et al., 2015b) use cost-based methods for join ordering. DARQ, SPLENDID, LHD, and Semagrow use dynamic programming (DP), whereas ADERIS employs greedy algorithm (GA) for the search strategy. Dynamic programming is breadth-first, while greedy algorithm is depth-first. Hence dynamic programming builds all possible plans before choosing the best one, whereas greedy algorithm builds only one plan (Ozsu and Valduriez, 2011). The cost functions of DARQ, SPLENDID, and LHD are explained in the following subsection. All these engines consider the cardinality estimations, cost for sending a triple pattern and cost for receiving a result. Although Semagrow considers the cardinality estimations and communication costs, it assigns a unique communication cost factor to each data source such as 10%. Charalambidis et al. (2015b) stated that different communication cost factors can be employed assuming that this information is available in the metadata catalogs. DARQ estimates the cardinalities by using the statistics in the service descriptions. On the other hand, SPLENDID, LHD, and Semagrow use VoID statistics in order to estimate the cardinalities.

LHD classifies the execution of joins as follows: (i) joins which do not require input bindings (plain access plan) and (ii) joins which require pre-computed bindings (dependent access plan). The joins in the first class can be executed in a parallel fashion, whereas the second one should be executed in a sequence due to the need of bindings. LHD uses plain access plans for the triple patterns which have concrete subject

or object. We refer to that heuristic as *concrete subject or object heuristic* (CSOH). Therefore, it first executes these triple patterns and then uses dynamic programming. Secondly, it determines the actual order of triple patterns to execute them in parallel by considering the type of access plans, bound variables, and the already bound variables from the previous iterations. It executes the triple patterns with plain access plan concurrently, while the triple patterns with the dependent access are executed as soon as its bindings are ready.

The first version of ADERIS (Lynden et al., 2010) builds predicate tables and adaptively joins two tables as they become complete while the other predicate tables are being generated. The second version of ADERIS (Lynden et al., 2011) uses an adaptive cost model for query optimization. Equation 2.1 (Lynden et al., 2011) shows the cost model of ADERIS where $incard$ is the estimated input cardinality for each iteration, $lookupTime$ refers to the average time taken to probe a given table t and R is the remaining set of tables that need to be joined to the current plan. Furthermore, join ordering is based on a greedy algorithm. The engine estimates cardinality at each stage for join ordering. In brief, ADERIS supports adaptive query processing.

$$cost(t) = incard \cdot lookupTime(t) \cdot \sum_{i \in R} lookupTime(i) \cdot cardEst(t) \tag{2.1}$$

2.2.3.3 Join Method Selection

As stated in Section 2.2.2, DARQ, SPLENDID, ANAPSID, ADERIS, LHD, and Semagrow implement two different join methods. We classify join method selection methods as follows: (i) binding limitation-based, (ii) cost-based, and (iii) time constrained-based.

ANAPSID (Acosta et al., 2011) employs bind join when a binding is required by a

data source, while DARQ (Quilitz and Leser, 2008) uses bind join when the data sources have limitations on access patterns. We refer to this method as *binding limitation-based* (BLB). However, DARQ uses cost models for join method selection when there is no binding limitation. We refer to this selection method as *cost-based* (CB).

SPLENDID (Görlitz and Staab, 2011b) uses cost models for join method selection as DARQ. Equation 2.2 (Quilitz and Leser, 2008) and Equation 2.3 (Quilitz and Leser, 2008) are the cost functions of DARQ for nested loop join and bind join, respectively, where q and p are the relations, $R(q)$ is the result size of q, c_t is the transfer cost for one tuple, and c_r is the transfer cost for one query. q_2' is the relation with the bindings of q_1. SPLENDID uses the same cost functions (Equation 2.2 and Equation 2.3) for hash join and bind join, respectively. Although DARQ and SPLENDID consider transfer costs, they ignore the different data arrival rates of relations.

$$cost(q_1 \bowtie_{NLJ} q_2) = |R(q_1)| \cdot c_t + |R(q_2)| \cdot c_t + 2 \cdot c_r \tag{2.2}$$

$$cost(q_1 \bowtie_{BJ} q_2) = |R(q_1)| \cdot c_t + |R(q_1)| \cdot c_r + |R({q_2}')| \cdot c_t \tag{2.3}$$

As mentioned previously, Semagrow can employ merge join and bind join. When the join type is equi-join, it calculates their costs and chooses the join method that has a lower cost. The engine estimates these cost by using the statistics in the VoID descriptions. Hence the join method selection of Semagrow is cost-based, too.

LHD is yet another federated query engine which selects the join method according to the cost functions. It proposes two different plans according to the usage of bindings in query execution, which are plain access plan and dependent access plan. The plain access plan of a triple pattern executes the triple pattern directly. Therefore, these

joins, such as hash join and nested loop join, do not need precomputed bindings. The dependent access plan uses the intermediate bindings in order to execute the triple pattern. Bind join is an example of these type of joins. Equations 2.4, 2.5, 2.6, and 2.7 (Wang et al., 2013) show the cost functions where a plain access plan of triple pattern t is denoted as $acc(t)$, and dependent access plan with bindings of q is represented as $acc(q, t)$. Also rt_q is the time of sending a triple pattern or a precomputed result to a data source, and rt_t is the time of receiving a result. These cost functions are quite similar to the cost models of DARQ and SPLENDID.

After data source selection, ADERIS generates predicate tables for each predicate in the query where the tables include subject and object values as the columns. These predicate tables are joined by using index nested loop join. A predicate table can be missing when an endpoint may refuse to answer the queries due to the timeouts. In that case, the engine sends a subquery with bindings for the subject or object values to the corresponding endpoint, hence the join method becomes bind join. We refer to this selection method as *time constrained-based* (TCB).

$$cost(q \bowtie p) = maximum(cost(q), cost(p)) \tag{2.4}$$

$$cost(q \bowtie_B p) = cost(q) + cost(acc(card(q), t)) \tag{2.5}$$

$$cost(acc(t)) = rt_q + card(t) \cdot rt_t \tag{2.6}$$

$$cost(acc(q, t)) = card(q) \cdot rt_q + card(q \bowtie t) \cdot rt_t \tag{2.7}$$

Table 2.4 shows the join method selection and join ordering methods in query federation which are substantially related with each other. FedX (Schwarte et al., 2011) and WoDQA (Akar et al., 2012; Yönyül, 2014) use heuristics, whereas DARQ (Quilitz and

Leser, 2008), SPLENDID (Görlitz and Staab, 2011b), ADERIS (Lynden et al., 2011), LHD (Wang et al., 2013), and Semagrow (Charalambidis et al., 2015b) propose cost functions for join ordering. FedX orders the joins by using *free variable heuristic* (FVH) and *exclusive grouping priority heuristic* (EGPH). WoDQA first orders the triple patterns by using *position and type based selectivity heuristic* (PTSH) and *shared variable heuristic* (SVH). Second, it orders the exclusive groups by SVH. DARQ, SPLENDID, and Semagrow (Charalambidis et al., 2015b) use *cost-based* (CB) method and dynamic programming (DP) for join method selection and join ordering, respectively. DARQ also uses *binding limitation-based* (BLB) method for join method selection. ANAPSID (Acosta et al., 2011) selects the join method by employing BLB as well. LHD uses *concrete subject or object heuristic* (CSOH) and employs dynamic programming for join ordering. It uses CB method for join method selection. ADERIS uses greedy algorithm (GA) for join ordering and employs *time constrained-based* (TCB) method for join method selection.

Table 2.4: Join ordering and join method selection in query federation

	Join ordering							Join method selection		
	FVH	EPGH	PTSH	SVH	CSOH	DP	GA	BLB	TCB	CB
DARQ						✓		✓		✓
FedX	✓	✓								
SPLENDID						✓				✓
ANAPSID								✓		
ADERIS							✓		✓	
LHD					✓	✓				✓
WoDQA			✓	✓						
Semagrow						✓				✓

2.2.4 Discussion on Query Federation on Linked Data

In this section, we summarize the main results from the previous section with a qualitative comparison and we state the challenges in query federation.

We compare the federated query engines qualitatively according to the following criteria:

- *No preprocessing per query:* Data source selection without using a metadata catalog might cause some performance problems due to the need of preprocessing for each query.

- *Unbound predicate queries:* Predicates are less selective than subjects and objects in a typical data source (Stocker et al., 2008). Therefore, selecting the data source of a query with an unbound predicate is difficult. However, the data source selection methods of some engines are based on predicates. These engines might have some problems to handle queries with unbound predicates.

- *Parallelisation:* Parallelisation is another fact which improves the performances of engines due to the concurrent query processing. It can be achieved in two forms which are inter-operator parallelism and intra-operator parallelism. More than one operations of a query are executed concurrently in inter-operator parallelism, whereas a single operator is executed by multiple processors in intra-operator parallelism.

- *Adaptive query processing:* Adaptive query processing (Deshpande et al., 2007) is a form of dynamic query processing which reacts to unforeseen variations of run-time environment (Ozsu and Valduriez, 2011). Since federated query processing is done on the Web, adaptive query processing is required in order to manage the

changing conditions such as different data arrival rates, endpoint unavailability, and timeouts.

Table 2.5 shows the qualitative comparison of the engines with the mentioned criteria. All engines use metadata catalogs in the data source selection except FedX (Schwarte et al., 2011). For this reason, it needs preprocessing per query before query processing. It sends SPARQL ASK queries to data sources for each query. However, it caches these results to be used later. Other engines primarily employ metadata catalogs. Actually, ADERIS (Lynden et al., 2010, 2011) sends SELECT DISTINCT queries to decide which predicates are covered by each data source. However, it does not send individual SPARQL ASK queries for each triple pattern in the query.

DARQ (Quilitz and Leser, 2008), ANAPSID (Acosta et al., 2011), and Semagrow(Charalambidis et al., 2015b) cannot manage unbound predicate queries, because their data source selection methods are predicate based only. Although data selection methods of ADERIS, SPLENDID (Görlitz and Staab, 2011b), and LHD (Wang et al., 2013) are based on predicates as well, they handle queries with unbound predicates by selecting all available data sources. They might cause some performance problems but the queries with unbound predicates can be supported by this way.

FedX, ANAPSID, LHD, and Semagrow execute the triple patterns in a parallel fashion. FedX integrates a parallelisation infrastructure to execute subqueries at different endpoints concurrently and uses a pipelining approach to send intermediate results to the next operator as they are ready. ANAPSID executes the triple patterns in parallel by proposing a join method based on symmetric hash join and XJoin. LHD separates the query plans and the communication with data sources for parallelisation. Several threads are used for sending triple patterns to a data source and for receiving results from a data source. It also considers the type of access plans, bound variables

Table 2.5: Comparison of Query Federation Engines

	No preprocessing per query	Unbound predicate queries	Parallelisation		Adaptive query processing
			Inter-operator	Intra-operator	
DARQ	✓				
FedX		✓	✓		
SPLENDID	✓	✓			
ANAPSID	✓			✓	✓
ADERIS	✓	✓			✓
LHD	✓	✓	✓	✓	
WoDQA	✓	✓			
Semagrow	✓		✓		

and the already bound variables from the previous iterations to adopt parallelisation. Furthermore, it uses multiple hash joins. FedX and Semagrow provide inter-operator parallelism, whereas ANAPSID employs intra-operator parallelism. LHD affords both inter-operator and intra-operator parallelism.

Only ANAPSID and ADERIS employ adaptive query processing. ANAPSID proposes a non-blocking join method, whereas ADERIS changes the join order dynamically. Besides, Semagrow uses reactive paradigm for union operators, in which the basic idea is based on notifying the operators when the data is available. Although it provides a kind of adaptivity, it can be accepted as a pipelining approach.

2.2.5 Challenges of Query Federation on Linked Data

During surveying the studies in query federation on Linked Data, we have noticed that there are some challenges and open research issues in this field, which are metadata management, caching results, and adaptive query processing. In this subsection, we state the first two challenges and suggest some ideas to handle them. We will discuss

the third challenge in the following section in detail.

2.2.5.1 Metadata Management

Metadata catalogs are useful in data source selection in query federation. As mentioned previously, service descriptions, VoID descriptions, and predicate lists are the examples of metadata catalogs used by the federated query engines. Service descriptions were presented by Quilitz and Leser (2008) for DARQ and VoID descriptions were proposed by Cyganiak et al. (2011) as a vocabulary which allows to define linked RDF datasets. In other words, VoID descriptions aim to provide a standard metadata publishing approach for RDF data. In addition, statistics about the entire dataset or the linkset can be expressed in the VoID descriptions and these statistics can be used in query optimization. Therefore, VoID descriptions are more appropriate for generating metadata catalogs due to providing the metadata of the data sources and their relations with the other data sources. However, there are some open research questions as follows. How is this metadata catalog generated? How often is this metadata catalog updated? More generally, how is metadata management supported?

A few number of data sources provide their metadata descriptions in practice, although most of the engines use metadata catalogs for data source selection. Thus, existing engines should generate these descriptions before query processing. Actually, VoID descriptions provide a standardized vocabulary to express the metadata about the dataset or the linkset and the statistics about the dataset. However, how are the metadata information and the statistics obtained? In addition, just generating a metadata catalog is not enough in practice. Keeping it up-to-date is an important aspect to be considered. The changes in the datasets should be covered in the catalogs as well. To conclude, there are two main challenges in metadata catalog usage. The first one

is generating the metadata catalogs. The other one is keeping the metadata catalogs up-to-date. In order to overcome these challenges, a metadata catalog management framework should be generated which can perform the following tasks: (i) gathering and expressing the metadata of the datasets and maybe the statistics with VoID descriptions, (ii) monitoring the changes in datasets, and (iii) updating the metadata catalogs according to these changes. Thus, an up-to-date, standardized metadata catalog management can be provided and it can be both used in data source selection and query optimization in federated query processing.

2.2.5.2 Caching Results

Caching has an important role in improving the performance in distributed query processing. Adali et al. (1996) proposed a query result caching mechanism by using the invariants which define the certain relationships between two different queries. When a query is covered by another query, the results of the covered query can be found from the cache by employing the invariants. The invariants should be decided through the knowledge about the data sources in the queries. Another option is employing quite general invariants with few information about the data source. Adali et al. (1996) stated that caching provides savings in time, and invariants is useful when the query is not explicitly cached. Caching is also employed by search engines to improve the response time as well. Gan and Suel (2009) discussed the studies for caching in search engines and Cambazoglu et al. (2012) classified the methods of caching as result, similarity, semantic, and rank caching.

Martin et al. (2010) proposed an approach as a proxy layer between Semantic Web applications and the SPARQL endpoints for caching. When the query is sent by the user, the cache is checked and if the results of this query exists in the cache, the answer

is returned without executing it over the endpoint. If the query is not cached previously, it is executed over its endpoint and then cached. Another proposal for caching of Linked Data was presented by Williams and Weaver (2011), which uses the last modification date information and the up-to-dateness in the HTTP headings. Although these studies (Martin et al., 2010; Williams and Weaver, 2011) provide caching results for the same queries, they cannot manage queries with small variations.

Lorey and Naumann (2013) proposed another caching approach for SPARQL queries with assuming that similar queries are executed over a SPARQL endpoint. For this reason, its strategy is based on prefetching which allows to gather data that is potentially useful for subsequent queries. In other words, this approach caches the results of query patterns to be used later. However, when the query pattern is too general, prefetching can be inefficient on large-scale datasets due to gathering large amount of data (Yönyül, 2014). Also, this approach cannot find all candidate subgraph matches of a SPARQL query (Papailiou et al., 2015).

In our literature survey (Oguz et al., 2015), we stated that combining caching results and live query results can decrease the query processing time of query federation engines as in distributed mediator systems. We also remarked that this caching mechanism should cover the subsets of a query and should have an updating strategy in order to make federated query processing more efficient. Besides, we discussed the usage of caching and live querying by employing hybrid query processing, which was used for link traversal (Umbrich et al., 2012a,b).

Papailiou et al. (2015) proposed an approach for adaptive indexing and caching frequent query patterns by monitoring the workload queries. It uses a canonical labeling technique for SPARQL queries without apriori knowledge about the dataset and the workload queries. The queries with different triple pattern orders or variables have the

same canonical label which is used a key for the cached results. Moreover, Papailiou et al. (2015) stated that their approach is applicable to all systems.

FedX (Schwarte et al., 2011) caches the results of the SPARQL ASK queries in the data source selection. WoDQA (Yönyül, 2014) caches the results of queries without considering subquery macthing. AVALANCHE (Basca and Bernstein, 2014), which is a technique for querying Web of Data, caches partial results of a query during the execution which can be used for the same subquery. We think that federated query engines should extend their caching mechanism in order to minimize their response and completion times.

2.3 Adaptive Query Optimization

In this section, we discuss the research on adaptive query optimization in both relational databases and query federation over Linked Data.

2.3.1 Adaptive Query Optimization for Relational Databases

Query optimization, which is performed by a query optimizer, refers to the process of generating an execution plan for the query. Therefore, the query optimizer is essential for a database management system engine. The query optimizer consists three components: a search space, a cost model, and a search strategy. The search space refers to the possible execution plans for the query. The cost model estimates the cost of a given execution plan. The search strategy explores the search space and selects the best plan with respect to the cost model. In other words, the query optimizer selects the execution plan which has the lowest cost according to the cost model (Ozsu and Valduriez, 2011; Yin et al., 2015).

Traditional query optimization (Selinger et al., 1979) can be inefficient in distributed systems due to the strong variations in the environment. Different from traditional query processing, adaptive query processing covers monitoring, assessing, and reacting activities in order to handle unforeseen variations of run-time conditions. Therefore, adaptive query processing has a feedback loop between the execution environment and the query optimizer (Ozsu and Valduriez, 2011). Adaptive query processing and adaptive query optimization are often used interchangeably.

Adaptive query processing for relational databases has been studied in detail by the database community. There are various degrees of adaptivity from evolutionary methods to revolutionary methods (Laddhad, 2006). Evolutionary methods focus on generating plans that can be switched during the execution according to delays or estimation errors. Their level of modification is inter-operator in which the feedback is collected from different physical operators (Gounaris et al., 2002). Some known examples of evolutionary methods are query scrambling (Amsaleg et al., 1998), mid-query re-optimization (Kabra and DeWitt, 1998), Tukwilla / ECA rules (Ives et al., 1999), proactive re-optimization (Babu et al., 2005), and progressive query optimization (Han et al., 2007; Markl et al., 2004; Kache et al., 2006). Revolutionary methods are more recent and their level of modification is intra-operator in which the feedback is collected during the evaluation of a physical operator (Gounaris et al., 2002). First group of intra-operator methods are adaptive operators like double pipelined hash join (Ives et al., 1999), XJoin (Urhan and Franklin, 2000), and mobile join (Arcangeli et al., 2004; Ozakar et al., 2005). The operators in this group is able to adapt its execution according to the variations during the execution. Second group of intra-operator methods come up with the invention of eddies which enable researchers to optimize the query processing from fine-grained to tuple-level (Avnur and Hellerstein,

2000; Raman et al., 2003; Deshpande, 2004; Deshpande and Hellerstein, 2004; Bizarro et al., 2005; Zhou et al., 2005).

2.3.2 Adaptive Query Optimization for Query Federation on Linked Data

Query federation over Linked Data is done on the distributed data sources on the Web. Hence data arrival rates of relations are unpredictable and most of the statistics are missing or unreliable. Therefore, we think that adaptive query optimization (Deshpande et al., 2007) is a necessity in order to handle such strong variations of this environment.

ANAPSID (Acosta et al., 2011) and ADERIS (Lynden et al., 2010, 2011) are the two federated query engines which use adaptive query optimization over SPARQL endpoints. ANAPSID proposes a non-blocking join method based on symmetric hash join (Wilschut and Apers, 1991) and Xjoin (Urhan and Franklin, 2000). ADERIS (Lynden et al., 2010) joins two predicate tables as they become complete, whereas ADERIS (Lynden et al., 2011) uses a cost model for dynamically changing the join order. Also, AVALANCHE (Basca and 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. The proposals of this thesis, namely AJO (Oguz et al., 2016) and EAJO (Oguz et al., 2016), also consider adaptive query optimization.

Table 2.6 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. A triple pattern fragment (Verborgh et al., 2014) is a Linked Data Fragment with three components which are selector, count metadata, and controls. A selector is a single triple pattern, count metadata refers to as metadata with total triple count, and controls provide retrieving any other triple pattern fragment of the same dataset.

- *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)*.
- *Type of Statistics (ToS):* States of the collection time of statistics which has the following values: *run-time (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. (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. (2002).

- *Type of Modification (ToM):* Can be employed as *rescheduling (rs)*, *dynamic operator (do)*, and *rescheduling and replacement (rs & rp)*.

As shown in Table 2.6, ADERIS (Lynden et al., 2011), ANAPSID (Acosta et al., 2011), AVALANCHE (Basca and Bernstein, 2014), AJO (Oguz et al., 2016), and EAJO (Oguz et al., 2016) use adaptive query optimization for queries over SPARQL endpoints. On the other hand, nLDE (Acosta and Vidal, 2015) proposes a client-side engine against triple pattern fragment servers which is similar to distributed eddies (Tian and DeWitt, 2003). Hence nLDE uses adaptive query optimization for queries over triple pattern fragments.

The proposals for SPARQL endpoints prefer to collect the statistics in run-time due to unreliable or missing statistics. Therefore, up-to-dateness of statistics is provided. On the other hand, nLDE uses metadata catalogs for the statistical information because triple pattern fragments contain both data, metadata, and controls. The second parameter in Table 2.6 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 an extended version of dependent join (Florescu et al., 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 2.6, 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. AVALANCHE

Table 2.6: Comparison of adaptive query optimization in query federation

	S	JM	ToS	FoF	ToE	LP	PP	ToM
ADERIS	se	inlj/bj	rt	inter	any	op_ro	uao	rs
ANAPSID	se	shj/bj	rt	intra	dar	no	uao	do
AVALANCHE	se	bj/bbj	rt	inter	dar	op_ro	no	rs
nLDE	tpfs	shj/nlj	md	intra	any	op_ro	no	rs
AJO	se	shj/bj	rt	intra	dar	rf	op_rep	rs&rp
EAJO	se	shj/bj/bbj	rt	intra	dar	rf	op_rep	rs&rp

and EAJO, in brief, 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 our proposals (AJO and 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 2.6, 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.

2.4 Conclusion

In the first section of this chapter, we have analyzed and synthesized the fundamental components of federated query processing which are data source selection, join methods, and query optimization. We have compared the existing federated query engines according to our proposed classifications. We have also stated the major challenges in federated query processing which are metadata management, caching results, and adaptive query processing; and we have discussed the first two challenges. Since we believe that the third challenge is the most crucial one, we have discussed it separately in the second section.

Linked Data environment has strong difficulties such as unpredictable data arrival rates and unreliable statistics. Most of the studies of query optimization in query federation focus on static query optimization (Selinger et al., 1979) which generates query execution plans before the execution and needs statistics (Quilitz and Leser, 2008; Schwarte et al., 2011; Görlitz and Staab, 2011b; Wang et al., 2013; Charalambidis et al., 2015b). However, static query optimization can cause inefficient execution plans. We think that adaptive query optimization (Deshpande et al., 2007) can handle the

mentioned difficulties of Linked Data environment.

ANAPSID (Acosta et al., 2011) and ADERIS (Lynden et al., 2010, 2011) use adaptive query optimization for federated queries over SPARQL endpoints. ANAPSID aims to minimize the response time, while ADERIS intends to minimize the completion time. However, to the best of our knowledge, there is not any study that aims to minimize both the response and the completion times when the query is executed over SPARQL endpoints.

In this thesis, we propose adaptive join operators which aim to minimize the response time and the completion time for federated queries over SPARQL endpoints. Both of our proposals can change the join method and the join order during the execution by using adaptive query optimization. Different from other studies which consider adaptive query optimization in query federation, our proposals can reformulate the remaining plan by replacing the join operator or changing the join order.

In the following chapter, we first present our operators which aim to handle the variations of Linked Data environment with considering the main goal of query optimization in federated query processing.

Chapter 3

Optimization Methods for Query Federation on Linked Data

Abstract: In this chapter, we present two proposals which aim to contribute to the query optimization of federated query processing on Linked Data. Since the objective of query optimization in federated query engines is to minimize the response time and the completion time, we first propose an adaptive join operator for federated queries over SPARQL endpoints with this goal. The proposed operator can change the join method during the execution by using adaptive query optimization. The operator can also change the join order in order to minimize the completion time. It can handle unexpected data arrival rates of relations and missing statistics. To the best of our knowledge, adaptive join operator is the first study which aims to minimize both the response time and the completion time for federated queries over SPARQL endpoints. Second, we propose an extension of the adaptive join operator, namely extended adaptive join operator, which aims to further reduce the completion time by employing an additional join method.

Contents

3.1 Introduction **73**

3.2 Adaptive Join Operator for Federated Queries **74**

3.2.1 Adaptive Join Operator for Single Join Queries 76

3.2.2 Adaptive Join Operator for Multi-Join Queries 80

3.3 Extended Adaptive Join Operator for Federated Queries **84**

3.3.1 Background 85

3.3.2 Extended Adaptive Join Operator for Single Join Queries . . 87

3.3.3 Extended Adaptive Join Operator for Multi-Join Queries . . 92

3.4 Conclusion **96**

3.1 Introduction

So far, we have presented the context, the position of the problem that motivated us for the work in this thesis, and the current state of the art. As mentioned in the previous chapters, federated query processing is performed with a federated query engine which distributes the subqueries to the SPARQL endpoints of the selected data sources to execute and then integrates the results of the subqueries to generate the final result set. The objective of query optimization in these engines is to minimize the response and the completion times. Response time refers to the time to produce the first result tuple, while completion time refers to the time to provide all result tuples. Communication cost is the dominant cost in both response time and completion time. Thus, the main goal of federated query engines can be stated as to minimize the communication cost.

To summarize the existing studies, Schwarte et al. (2011) use heuristics in query optimization, whereas Quilitz and Leser (2008), Görlitz and Staab (2011a) and Wang et al. (2013) concentrate on static query 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, and due to this, data arrival rates are unpredictable. 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. However, there are only two engines, ANAPSID (Acosta et al., 2011) and ADERIS (Lynden et al., 2010, 2011) which consider adaptive query optimization for query federation. Acosta et al. (2011) proposed a non-blocking join method based on symmetric hash join (Wilschut and Apers, 1991) and Xjoin (Urhan and Franklin, 2000) to minimize the response time, whereas Lynden et al. (2011) proposed a cost model for dynamically changing the join order to minimize the completion time. In

addition to these federated query engines, Basca and Bernstein (2010, 2014) proposed a technique called AVALANCHE which gathers statistics on the fly before query execution and produces only the first k results with the aim of minimizing the response time. To the best of our knowledge, there is not any study that exploits an adaptive join operator that aims to minimize both the response time and the completion time for federated queries over SPARQL endpoints. In addition, communication time has the highest effect on overall cost as mentioned earlier. 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.

The contribution of this chapter is as follows. First, 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. Second, we propose an extended version of our previous operator, called extended adaptive join operator, which aims to further reduce the completion time.

In Section 3.2, we propose the adaptive join operator and present the algorithms for single join queries and multi-join queries in detail. In Section 3.3, we introduce the extended adaptive join operator and its algorithms for both single and multi-join queries.

3.2 Adaptive Join Operator for Federated Queries

Join method selection plays an important role in query optimization. Symmetric hash join (Wilschut and Apers, 1991) is a join method which maintains a hash table for each relation. Therefore, it is defined as a non-blocking join method and produces the first result tuple as early as possible. Bind join (Haas et al., 1997), which is the most

popular join method among the federated query engines (Oguz et al., 2015), passes the bindings of the intermediate results of the outer relation to the inner relation in order to filter the result set. In brief, symmetric hash join provides short response time, whereas bind join provides short completion time when the cardinality of the intermediate results is low.

Equation 3.1 and Equation 3.2 are the cost functions of symmetric hash join and bind join, respectively. Equation 3.2 is a variation of the formula used by (Quilitz and Leser, 2008) in which they assume that the transfer costs of different relations are the same. However, we consider different transfer costs of relations. R_i and R_j are the relations while $card(R)$ is the number of tuples in R. The transfer costs of R_i and R_j for one result tuple are c_{t_i} and c_{t_j}, respectively. $R_j{}'$ is the relation with the bindings of R_i. Actually, $card(R_j{}')$ means $card(R_i \bowtie R_j)$ when we assume that the common attribute values are unique.

$$cost(R_i \bowtie_{SHJ} R_j) = max\Big(\big(card(R_i) \cdot c_{t_i}\big), \big(card(R_j) \cdot c_{t_j}\big)\Big) \tag{3.1}$$

$$cost(R_i \bowtie_{BJ} R_j) = card(R_i) \cdot c_{t_i} + card(R_i) \cdot c_{t_j} + card(R_j{}') \cdot c_{t_j} \tag{3.2}$$

Static query optimization decides the join method before the query execution and thus it can cause inefficient query plans due to unpredictable data arrival rates and missing statistics. The join cardinality, $card(R_i \bowtie R_j)$, 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 R_i and R_j, without waiting for all tuples of R_i to arrive. However, if the

cardinality of R_j is very high while the join cardinality is low, the query completion time of symmetric hash join can be longer than the completion time of bind join.

Since the data arrival rates of relations are known after a short time of execution, the remaining completion times can be estimated. For these reasons, we propose to set the join method as symmetric hash join in the beginning in order to minimize the response time, and to use cost functions after having information about the data arrival rates of endpoints to minimize the completion time. We decide whether to change the join method to bind join according to the cost estimations. In order to learn the cardinalities of relations, we send count queries in the beginning of the execution. As mentioned before, the communication time dominates the I/O time and CPU time. Hence the costs of count queries are 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 and the join cardinalities with the aim of minimizing both the response time and the completion time.

3.2.1 Adaptive Join Operator for Single Join Queries

In this subsection, we first present the algorithm of the adaptive join operator for single join queries. Second, we propose the join cardinality estimation formula and the cost estimations for symmetric hash join and bind join.

Figure 3.1 summarizes the adaptive join operator for single join queries. Our operator always begins with symmetric hash join and it calculates the estimated remaining times for both join methods when all the tuples of a relation arrive. It changes the join method to bind join if the remaining time of bind join is less than the remaining time of symmetric hash join. Adaptive join operator not only can change the join method, but also has the ability to change the join order.

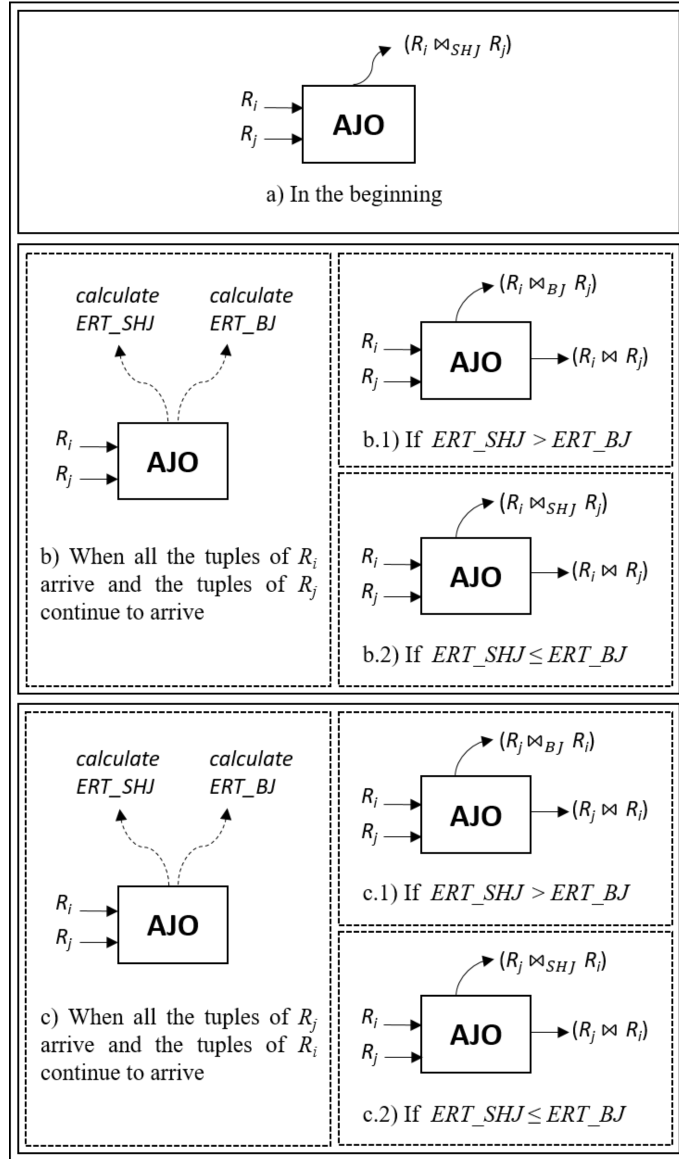


Figure 3.1: Adaptive join operator for single join queries

The algorithm of the adaptive join operator for single join queries is depicted in Algorithm 1. Firstly, the adaptive operator sends count queries to the SPARQL endpoints of data sources R_i and R_j in order to learn their cardinalities. The operator always begins with symmetric hash join in order to produce the first result tuple as early as possible. In other words, it always sets the join method as symmetric hash

join in the beginning in order to minimize the response time. During the execution, when all the tuples from one data source arrive and the tuples from the other data source continue to arrive, the adaptive join operator estimates the remaining time of continuing with symmetric hash join and the remaining time of switching to bind join. It selects the join method according to these cost estimations. If the operator switches to bind join, it emits the duplicate results of symmetric hash join and bind join. The cardinality estimation formula and the remaining time estimation formulas will be presented in the following of this subsection. We use the term "cardinality" instead of "number of triple patterns" in the rest of the paper.

Algorithm 1: Adaptive join operator for single join queries

1 $|R_i| \longleftarrow cardinality\ of\ R_i\ received\ from\ the\ COUNT\ query$
2 $|R_j| \longleftarrow cardinality\ of\ R_j\ received\ from\ the\ COUNT\ query$
3 $|R_{i_arrived}| \longleftarrow cardinality\ of\ arrived\ R_i\ tuples$
4 $|R_{j_arrived}| \longleftarrow cardinality\ of\ arrived\ R_j\ tuples$
5 Set JOIN method as Symmetric Hash Join (SHJ)
6 **while** $(|R_{i_arrived}| < |R_i|\ or\ |R_{j_arrived}| < |R_j|)$ **do**
7   **if** $(|R_{i_arrived}| == |R_i|\ and\ |R_{j_arrived}| < |R_j|\ or$ $|R_{j_arrived}| == |R_j|\ and\ |R_{i_arrived}| < |R_i|)$ **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**

Cardinality and Remaining Time Estimations

In this subsection, we explain our cardinality and remaining time estimations which are used in the decision of the join method for the rest of the execution. These estimates are calculated when all the tuples of a relation arrive from its SPARQL endpoint.

Equation 3.3 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_i} is the transfer cost of R_i for one result tuple, and c_{t_j} is the transfer cost of R_j for one result tuple. $R_j{}'$ is the relation with the bindings of R_i. Hence $|R_j{}'|$ is the cardinality of R_j which is reduced by the bindings of R_i. $|R_j{}'|$ is equal to the join cardinality, $|R_i \bowtie R_j|$, when we assume that the common attribute values are unique.

$$cost(R_i \bowtie_{BJ} R_j) = |R_i| \cdot c_{t_i} + |R_i| \cdot c_{t_j} + |R_j{}'| \cdot c_{t_j} \tag{3.3}$$

Equation 3.4 is the cardinality estimation formula for the second relation reduced with the bindings of the first relation. $|R_i \bowtie R_{j_arrived}|$ 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. We use this formula 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.

$$|R_j_estimation'| = \frac{|R_i \bowtie R_{j_arrived}| \cdot |R_j|}{|R_{j_arrived}|} \tag{3.4}$$

As stated earlier, when all the tuples of R_i arrive, the algorithm estimates the remaining time if the adaptive join operator continues with symmetric hash join and the remaining time if it changes the join method to bind join. We have an idea about the data arrival rate of R_j during the execution, so the estimation is possible. Equation 3.5

shows the estimated remaining time if the adaptive operator continues with symmetric hash join, ERT_{SHJ}, 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.

$$ERT_{SHJ} = \frac{(|R_j| - |R_{j_arrived}|) \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.5}$$

Equation 3.6 show the estimated remaining time if the algorithm switches to bind join, ERT_{BJ}, where $|R_i|$ is the cardinality of R_i, t_{ST} is the time for sending one result tuple to the SPARQL endpoint of R_j ($\approx \frac{t_{Rj_arrived}}{|R_{j_arrived}|}$), $|R_j_estimation'|$ is the estimated 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. 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 $R_j{}'$ from the endpoint of R_j.

$$ERT_{BJ} = (|R_i| \cdot t_{ST}) + \frac{|R_j_estimation'| \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.6}$$

3.2.2 Adaptive Join Operator for Multi-Join Queries

In this subsection, we introduce the adaptive join operator for multi-join queries which means there are more than two relations in the query. In other words, the query is comprised of more than two subqueries.

Figure 3.2 and Algorithm 2 explain the working principle of the adaptive join operator for multi-join queries. The operator uses multi-way symmetric hash join (Viglas et al., 2003) in the beginning instead of symmetric hash join since there are more than two relations to be joined. When all the tuples of a relation arrive, called R_i, the algorithm estimates the remaining time if the 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 following costs: i) estimated remaining time if it changes the join method to bind join for R_i and R_j and continues with multi-way symmetric hash join for other relations, ii) estimated remaining time if the operator continues with multi-way symmetric hash join for all relations. The adaptive join operator chooses the minimum cost and the above procedure is repeated every time a relation is completely received.

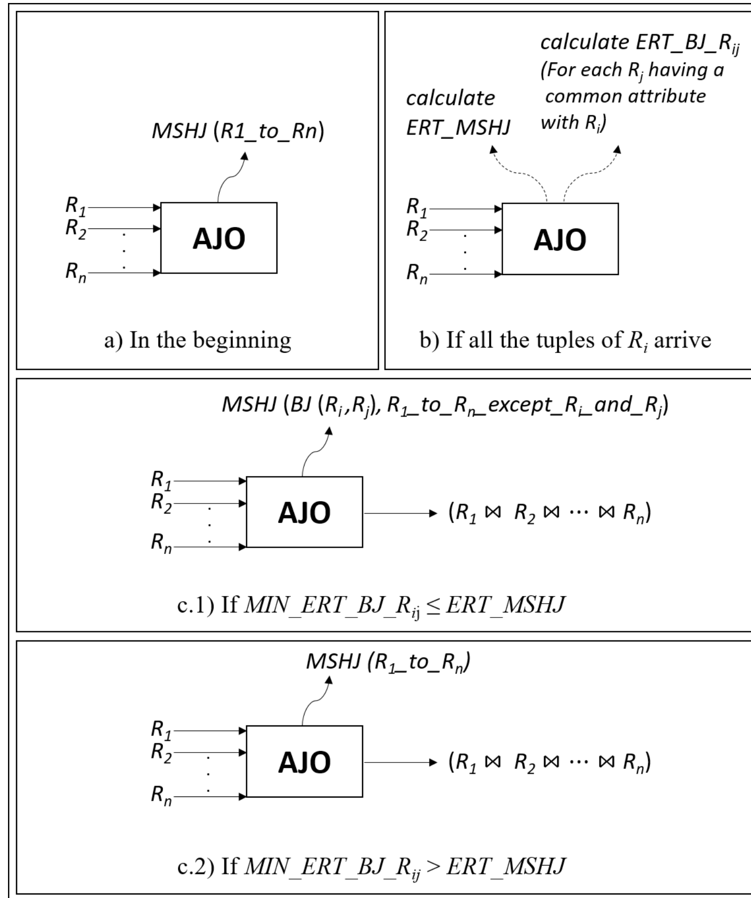


Figure 3.2: Adaptive join operator for multi-join queries

Algorithm 2: Adaptive join operator for multi-join queries

1. $S \longleftarrow \{R_1, R_2, R_3, \ldots, R_n\}$
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**

Cardinality and Remaining Time Estimations

Let R_1, R_2, ... and R_n are the relations of the query. When all tuples of a relation, called R_i arrive, we calculate the estimated remaining times if the adaptive join operator changes the join method to bind join for each relation which has a common attribute with R_i. Let R_j is the relation to be joined with R_i. We use Equation 3.7 for the estimated cardinality of the second relation which is reduced by the bindings of the first relation, called $R_j{}'$ $|R_i \bowtie R_{j_arrived}|$ 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. We use this formula in order to calculate the estimated cardinality of $R_j{}'$ when all the tuples of R_i arrive. We need this estimation in order to calculate the estimated remaining time whether the adaptive join operator switches to bind join for R_i and R_j or it continues with multi-way symmetric hash join for all relations. In fact, we use the same cardinality estimation for single join queries and multi-join queries.

$$|R_j_estimation'| = \frac{|R_i \bowtie R_{j_arrived}| \cdot |R_j|}{|R_{j_arrived}|} \tag{3.7}$$

The estimated remaining time for multi-way symmetric hash join is shown in Equation 3.8, where $|R_k|$ is the cardinality of R_k, $|R_{k_arrived}|$ is the cardinality of arrived tuples of R_k, and $t_{Rk_arrived}$ is the time for $R_{k_arrived}$ tuples to arrive. The completion time of multi-way symmetric hash join is equal to the maximum completion time of the relations which are involved in the query.

$$ERT_{MSHJ} = max\Big(\frac{(|R_k| - |R_{k_arrived}|) \cdot t_{Rk_arrived}}{|R_{k_arrived}|}\Big) \text{ where } k \in [1, \ldots, n] \tag{3.8}$$

Equation 3.9 shows the estimated remaining time if the adaptive join operator

uses bind join for R_i and R_j, and employs multi-way symmetric hash join for the other relations of the query. $|R_i|$ is the cardinality of R_i, t_{ST} is the time for sending one query to the SPARQL endpoint of $R_j(\approx \frac{t_{Rj_arrived}}{|R_{j_arrived}|})$, $|R_i \bowtie R_j|$ is the estimated cardinality of $R_i \bowtie R_j$, $|R_{j_arrived}|$ is the cardinality of arrived tuples of R_j, and $t_{Rj_arrived}$ 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 as shown in Equation 3.10 where $k \in [1, \ldots, n]$, $k \neq i$ and $k \neq j$.

$$ERT_{BJ_R_{ij}} = max\Big((|R_i| \cdot t_{ST} + \frac{|R_i \bowtie R_j| \cdot t_{Rj_arrived}}{|R_{j_arrived}|}), ERT_{rest}\Big) \tag{3.9}$$

$$ERT_{rest} = max\Big(\frac{(|R_k| - |R_{k_arrived}|) \cdot t_{Rk_arrived}}{|R_{k_arrived}|}\Big) \tag{3.10}$$

3.3 Extended Adaptive Join Operator for Federated Queries

In this section, we propose an extended version of the adaptive join operator which is improved with bind-bloom join (Basca and Bernstein, 2014; Groppe et al., 2015) to further reduce the communication time and, consequently, to minimize the completion time.

We first summarize the symmetric hash join and the bind join which are explained in the previous sections and then we explain the principles of bloom filter and bind-bloom join. Second, we present the extended adaptive join operator for single join queries and multi-join queries.

3.3.1 Background

We have explained the principles of symmetric hash join and bind join, and we have introduced their cost functions in the previous sections. Symmetric hash join provides short response time since it operates the subqueries in a parallel fashion. On the other hand, bind join passes the intermediate results of the first relation to the second relation in order to filter the result set. Hence bind join is successful with respect to completion time when the cardinalities of the first relation and the intermediate results are low.

As mentioned earlier, communication cost is the dominant cost in distributed environments. In order to reduce the communication cost, a space efficient data structure called bloom filter (Bloom, 1970) is widely used in relational databases (Mackert and Lohman, 1986; Mullin, 1990; Michael et al., 2007; Ives and Taylor, 2008). It is utilized in different Linked Data subjects such as identity reasoning (Williams, 2008) and data source selection (Hose and Schenkel, 2012). Bloom filter is also used to reduce the communication cost in two studies of Linked Data (Basca and Bernstein, 2014; Groppe et al., 2015). We briefly explain the bloom filter before presenting our proposal which uses it in order to reduce the communication cost.

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 $S = \{e_1, e_2, \ldots, e_n\}$ 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 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 et al., 2000). Independent of the size of the elements, less than 10 bits per element are required for

a 1% false positive probability (Bonomi et al., 2006).

We propose to use b bits per each element and k hash functions in order 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 $h1$(*?commonAttribute*), $h2$(*?commonAttribute*), $\ldots$, hk(*?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 3.1. 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 3.2. The intermediate results from *:service1* are shown in Table 3.1. 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 3.3 where *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. As a result, bloom filter decreases the size of the intermediate results.

```
SELECT * WHERE {
 SERVICE <:s1> { ?student :name :studentName . }
 SERVICE <:s2> { ?student :enroll ?course . } }
```

Listing 3.1: Federated query example

Table 3.1: Intermediate results

Line	student
1	*student_1*
...	...
n	*student_n*

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

Listing 3.2: Bind query

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

Listing 3.3: Bind query with bloom filter

Although bind-bloom join reduces the size of sent data to the second relation, 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. We will present the extended adaptive join operator for single join queries and multi-join queries in the following of this section.

3.3.2 Extended Adaptive Join Operator for Single Join Queries

Algorithm 3 shows the pseudo code of the extended adaptive join operator for single join queries. Firstly, we send count queries to the endpoints of data sources $R1$ 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 a data source arrive and the tuples from the other data source 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 of this subsection.

Algorithm 3: Extended 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         $ERT_{SHJ} \longleftarrow estimated\ remaining\ time\ (ERT)\ if\ continued\ with\ SHJ$
9         $ERT_{BJ} \longleftarrow ERT\ if\ switched\ to\ Bind\ Join\ (BJ)$
10         $ERT_{BBJ} \longleftarrow ERT\ if\ switched\ to\ Bind - Bloom\ Join\ (BBJ)$
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**

Cardinality and Remaining Time Estimations

Equation 3.11 shows the cost function of bind join where R_i and R_j are relations, $|R|$ is the number of tuples in R, and c_t is the transfer cost of R for one result tuple. $R_j{}'$ 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 $R_j{}'$. In the adaptive join operator, we assume that the common attribute values are unique. In this case, we consider the possibility of including duplicate values on the common attributes of the relations.

$$cost(R_i \bowtie_{BJ} R_j) = |R_i| \cdot c_{t_i} + |R_i| \cdot c_{t_j} + |R_j{}'| \cdot c_{t_j} \tag{3.11}$$

Before presenting our cardinality and remaining time estimations, we want to clarify and define the average duplication factor of a relation. Let R_i and R_j are the two relations which have a common attribute. Average duplication factor of R_i on R_j, $ADF(R_i, R_j)$, is the average duplication factor value 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 3.12 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 define an average duplication factor since we cannot guarantee a constant duplication factor for each attribute.

$$ADF(R_i, R_j) = \frac{|R_i|}{|R_i_uca|} \tag{3.12}$$

Assume that the relations, namely R_i and R_j, contains the attribute values which are shown in Table 3.2. The common attribute between R_i and R_j is a as indicated in the table. $ADF(R_i, R_j)$ calculation for the example relations can be seen below:

$$ADF(R_i, R_j) = \frac{|R_i|}{|R_{i}_uca|} = \frac{7}{4} = 1.75$$

Table 3.2: Example relations R_i and R_j

R_i	
a	b
a1	b1
a1	b2
a2	b3
a2	b4
a3	b5
a4	b6
a4	b7

R_j	
a	c
a1	c1
a2	c2
a5	c3
a6	c4
a7	c5
a8	c6
a9	c7

Equation 3.13 is used to estimate the cardinality of the second relation which is reduced by the bindings of the first relation. $|R_i \bowtie R_{j_arrived}|$ 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 extended adaptive join operator uses the estimated cardinality in order to estimate the remaining times of bind join and bind-bloom join. In other words, the operator employs Equation 3.13 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.

$$|R_{j}_estimation'| = \frac{|R_i \bowtie R_{j_arrived}| \cdot |R_j|}{|R_{j_arrived}|} \Big/ ADF(R_i, R_j) \tag{3.13}$$

As stated in the beginning of this subsection, when all the tuples of R_i arrive, the algorithm estimates three remaining times as follows: (i) the remaining time if the

extended adaptive join operator continues with symmetric hash join, (ii) the remaining time if it changes the join method to bind join, and (iii) the remaining time if it changes the join method to bind-bloom join. During the execution, we have an idea about the data arrival rate of R_j, and thus the estimation is possible. Equation 3.14 shows the estimated remaining time for symmetric hash join, ERT_{SHJ}, 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.

$$ERT_{SHJ} = \frac{\left(|R_j| - |R_{j_arrived}|\right) \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.14}$$

Equation 3.15 shows the estimated remaining time if the algorithm switches to bind join, namely ERT_{BJ}. $|R_i_uca|$ 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 ($\approx \frac{t_{Rj_arrived}}{|R_{j_arrived}|}$), 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.

$$ERT_{BJ} = \left(|R_i_uca| \cdot t_{ST}\right) + \frac{|R_j_estimation'| \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.15}$$

Equation 3.16 shows the estimated remaining time if the algorithm switches to bind-bloom join, namely ERT_{BBJ}, 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 ($\approx \frac{s(|R_{j_arrived}|)}{|R_{j_arrived}|}$, 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.

$$ERT_{BBJ} = \frac{b \cdot |R_i_uca|}{dr_j} + \frac{\left(|R_j_estimation'| + |fp|\right) \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.16}$$

3.3.3 Extended Adaptive Join Operator for Multi-Join Queries

In multi-join queries, we begin with a non-blocking join method in order to minimize the response time as in single join queries. In this case, we use multi-way symmetric hash join (Viglas et al., 2003) since there are more than two relations. The algorithm of the extended operator for multi-join queries is depicted in Algorithm 4. When all the tuples from a relation arrive, called R_i, the algorithm estimates the remaining times if the extended adaptive 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. Then, the algorithm compares the following estimated times: (i) the remaining time if the operator continues with multi-way symmetric hash join for all relations belonging to the query, (ii) the remaining time if the operator changes the join method to bind join for $R_i \bowtie R_j$ and uses multi-way symmetric hash join for the other relations, (ii) the remaining time if the operator changes the join method to bind-bloom join for $R_i \bowtie R_j$ and uses multi-way symmetric hash join for the other relations. The above procedure is repeated every time a relation is completely received.

Algorithm 4: Extended adaptive join operator for multi-join queries

1 $S \longleftarrow \{R_1, R_2, \ldots, R_n\}$
2 Send COUNT queries to the endpoints of $R_1, R_2, \ldots, R_n$
3 $MIN_ERT_{BJ} = MIN_ERT_{BBJ} \longleftarrow \infty$
4 $MIN_ET_{BJ} = MIN_ET_{BBJ} \longleftarrow \infty$
5 $BJ_Candidate = BBJ_Candidate \longleftarrow \Phi$
6 Start $MSHJ(S)$
7 **while** (S *is not empty*) **do**
8   **if** (*all the tuples of* R_i *arrive*) **then**
9         $ERT_{MSHJ} \longleftarrow ERT\ if\ continued\ with\ MSHJ$
10   **foreach** R_j *having a common attribute with* R_i **do**
11             $ERT_{BJ_R_{ij}} \longleftarrow ERT\ if\ switched\ to\ BJ\ for\ R_i\ and\ R_j$
                $ERT_{BBJ_R_{ij}} \longleftarrow ERT\ if\ switched\ to\ BBJ\ for\ R_i\ and\ R_j$
                $ET_{BJ_R_{ij}} \longleftarrow estimated\ time\ for\ BJ\ between\ R_i\ and\ R_j$
                $ET_{BBJ_R_{ij}} \longleftarrow estimated\ time\ for\ BBJ\ between\ R_i\ and\ R_j$
12   **if** ($ERT_{BJ_R_{ij}} < MIN_ERT_{BJ}$) **then**
13                 $MIN_ERT_{BJ} \longleftarrow ERT_{BJ_R_{ij}}$
14                 $MIN_ET_{BJ} \longleftarrow ET_{BJ_R_{ij}}$
15                 $BJ_Candidate \longleftarrow \{R_i, R_j\}$
16   **end**
17   **if** ($ERT_{BBJ_R_{ij}} < MIN_ERT_{BBJ}$) **then**
18                 $MIN_ERT_{BBJ} \longleftarrow ERT_{BBJ_R_{ij}}$
19                 $MIN_ET_{BBJ} \longleftarrow ET_{BBJ_R_{ij}}$
20                 $BBJ_Candidate \longleftarrow \{R_i, R_j\}$
21   **end**
22   **end**
23   **if** ($MIN_ERT_{BJ} <= ERT_{MSHJ}$) **then**
24   **if** ($ET_{BBJ_R_{ij}} < ET_{BJ_R_{ij}}$) **then**
25                 $\acute{R_i} \longleftarrow BBJ(R_i, R_j)$
26                 $S \longleftarrow S - BBJ_Candidate + \{\acute{R_i}\}$
27   Run $MSHJ(S)$ and eliminate duplicate results
28   **end**
29             $\acute{R_i} \longleftarrow BJ(R_i, R_j)$
30             $S \longleftarrow S - BJ_Candidate + \{\acute{R_i}\}$
31   Run $MSHJ(S)$ and eliminate duplicate results
32   **end**
33   **end**
34 **end**

Cardinality and Remaining Time Estimations

We use the same formula for single join queries and multi-join queries to estimate the cardinality of the second relation reduced by the bindings of the first relation. Therefore, we use Equation 3.13 which is shown in Section 3.3.2 for multi-join queries as well. We need this estimation in order to calculate the estimated remaining times for the following cases: (i) if the operator switches to bind join for $R_i \bowtie R_j$, (ii) if the operator switches to bind-bloom join for $R_i \bowtie R_j$, and (ii) if the operator continues with multi-way symmetric hash join.

Equation 3.17 shows the estimated remaining time if the extended adaptive join operator continues with multi-way symmetric hash join. Completion time is equal to the maximum completion time of the relations belonging to the query.

$$ERT_{MSHJ} = max\Big(\frac{(|R_k| - |R_{k_arrived}|) \cdot t_{Rk_arrived}}{|R_{k_arrived}|}\Big) \text{ where } k \in [1, \dots, n] \tag{3.17}$$

Equation 3.18 shows the estimated remaining time if the extended adaptive join operator employs bind join for R_i and R_j, and uses multi-way symmetric hash join for the other relations belonging to the query. It is equal to the maximum time between $ET_{BJ_R_{ij}}$ and ERT_{rest}. $ET_{BJ_R_{ij}}$ is the estimated time if the operator employs bind join for R_i and R_j. ERT_{rest} is the estimated remaining time for the rest of other relations to arrive. $ET_{BJ_R_{ij}}$ is shown in Equation 3.19. $|R_i_uca|$ 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 ($\approx \frac{t_{Rj_arrived}}{|R_{j_arrived}|}$), 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, $|R_{j_arrived}|$ is the cardinality of arrived tuples of R_j, and $t_{Rj_arrived}$ is the

time for $R_j_arrived$ tuples to arrive. ERT_{rest} is calculated by using Equation 3.20 where $k \in [1, \ldots, n]$, $k \neq i$ and $k \neq j$. $|R_k|$ is the cardinality of R_k, $|R_{k_arrived}|$ is the cardinality of arrived tuples of R_k, and $t_{Rk_arrived}$ is the time for $R_{k_arrived}$ tuples to arrive.

$$ERT_{BJ_R_{ij}} = max(ET_{BJ_R_{ij}}, ERT_{rest}) \tag{3.18}$$

$$ET_{BJ_R_{ij}} = (|R_i_uca| \cdot t_{ST}) + \frac{|R_j_estimation'| \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.19}$$

$$ERT_{rest} = max\Bigg(\frac{\Big(|R_k| - |R_{k_arrived}|\Big) \cdot t_{Rk_arrived}}{|R_{k_arrived}|}\Bigg) \tag{3.20}$$

Equation 3.21 shows the estimated remaining time if the extended operator switches to bind-bloom join for R_i and R_j, and uses multi-way symmetric hash join for the other relations belonging to the query. It is equal to the maximum time between $ET_{BBJ_R_{ij}}$ and ERT_{rest}. $ET_{BBJ_R_{ij}}$ is the estimated time if the operator employs bind-bloom join for R_i and R_j. ERT_{rest} is the estimated remaining time for the rest of other relations to arrive. $ET_{BBJ_R_{ij}}$ is calculated by using Equation 3.22. 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 ($\approx \frac{s(|R_{j_arrived}|)}{|R_{j_arrived}|}$, 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, and $t_{Rj_arrived}$ is the time for $R_{j_arrived}$ tuples to arrive. We use Equation 3.13 and Equation 3.20 in order to calculate $|R_j_estimation'|$ and ERT_{rest}, respectively.

$$ERT_{BBJ_R_{ij}} = max\Big(ET_{BBJ_R_{ij}}, ERT_{rest}\Big) \tag{3.21}$$

$$ET_{BBJ_R_{ij}} = \frac{b \cdot |R_i_uca|}{dr_j} + \frac{\Big(|R_j_estimation'| + |fp|\Big) \cdot t_{Rj_arrived}}{|R_{j_arrived}|} \tag{3.22}$$

3.4 Conclusion

Query optimization in query federation aims to minimize the response time and the completion time. Query federation distributes the subqueries of a query to the relevant SPARQL endpoints to be executed and then aggregates their results. However, the data arrival rates of relations are unpredictable since the execution is done on the distributed data sources on the Web. Moreover, the most of the statistics are missing. These constraints show that adaptive query optimization (Deshpande et al., 2007) is a need for query federation over SPARQL endpoints.

In this chapter, we presented two proposals which use adaptive query optimization for SPARQL query federation in order to minimize both the response time and the completion time. Since the communication cost mainly dominates the other costs in distributed environments, we focused on the minimization of the communication cost.

First proposal, namely adaptive join operator, initially sends count queries to the endpoints of relations in order to learn their cardinalities. The operator always begins with symmetric hash join and multi-way symmetric hash join for single join queries and multi-join queries, respectively, with the aim of minimization of the response time. The data arrival rates of relations are known after a short time of execution. For single join queries, the operator estimates the remaining times for symmetric hash join and bind join when all the tuples of a relation arrive. Different from single join queries, the operator chooses the minimum bind join cost between the received relation and the

relation which has a common attribute with this relation. Then, the operator compares the following cases: i) remaining time of continuing with multi-way symmetrich hash join for all relations, and ii) the remamining time of using bind join for the relations with the minimum bind join cost and using multi-way symmetric hash join for the rest of the relations. According to the remaining time estimations, the operator decides whether to change the join method to bind join or not.

In the second study, we proposed the extended adaptive join operator which is the improved version of our previous proposal. We aimed to further reduce the communication cost. For this reason, we included the bind-bloom join, which is a kind of bind join enhanced with bloom filter, to the candidate join methods. Since a bloom filter can contain a low rate of false positives, we keep bind join in our candidate join methods. The extended join operator again begins with symmetric hash join and multi-way symmetric hash join for single and multi-join queries, respectively. When all the tuples of a relation arrive, the remaining time estimations are calculated for symmetric hash join (or multi-way hash join), bind join and bind-bloom join.

The goal of the proposed operators are as follows: i) minimization of both the response time and the completion time, ii) managing with different data arrival rates, iii) handling the problem of missing statistics. The proposed adaptive join operators use a non-blocking join method in the beginning and they can change the join method during the execution to minimize the completion time. Moreover, both operators can change the join order as well. Therefore, both of our proposals aim to provide the best trade-off between the response time and the completion time.

Chapter 4

Performance Evaluation

Abstract: This section provides the performance evaluations of our proposals, namely adaptive join operator and extended adaptive join operator. We use response time and completion time as evaluation metrics. First, we evaluate and discuss the performance evaluation of the adaptive join operator for single join queries and multi-join queries. We compare the proposed operator with symmetric hash join and bind join. We discuss the impact of data sizes and the data arrival rates. Second, we present the results and discussions on the performance evaluation of the extended adaptive join operator for single join queries and multi-join queries. We evaluate the performances of the extended operator, symmetric hash join, bind join, bind-bloom join, and adaptive join operator. We again discuss the impact of data sizes and the data arrival rates. In addition, we show the impact of bit vector size for the extended adaptive join operator. We also present the speedup of the extended adaptive join operator compared to the adaptive join operator with respect to the completion time.

Contents

4.1 Introduction 101

4.2 Performance Evaluation of Adaptive Join Operator 102

4.2.1 Performance Evaluation for Single Join Queries 102

4.2.2 Performance Evaluation for Multi-Join Queries 107

4.3 Performance Evaluation of Extended Adaptive Join Operator 113

4.3.1 Performance Evaluation for Single Join Queries 114

4.3.2 Performance Evaluation for Multi-Join Queries 125

4.4 Conclusion 132

4.1 Introduction

This chapter includes two main sections as follows. Section 4.2 presents the performance evaluation of adaptive join operator and Section 4.3 provides the performance evaluation of extended adaptive join operator. The performances of both operators are evaluated for single join queries and multi-join queries. There are two relations in single join queries, while there are three relations in multi-join queries.

As stated in the previous chapters, the goal of query optimization in query federation is to minimize the response time and the completion time. For this reason, we used them as evaluation metrics. Both of them include communication time, I/O time, and CPU time. We mentioned earlier that query cost in distributed environments is mainly defined by the 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 analyze sample result sizes and consequently 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 $0.5/\Big(max($cardinality of $R1$, cardinality of $R2)\Big)$ (Shekita et al., 1993). We set the low, medium, and high cardinality as 1000 tuples, 5000 tuples, and 10000 tuples, respectively. We analyze the data arrival rates of 28 endpoints to assign the range of data arrival rates of relations in simulations. We conducted the simulations with different data arrival rates as explained in the following sections, however we always fixed their delays to 10 ms.

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

4.2 Performance Evaluation of Adaptive Join Operator

In this section, we present the evaluation results on the performances of symmetric hash join (or multi-way symmetric hash join), bind join, and adaptive join operator for single join queries and multi-join queries. The reason of comparing our proposal with symmetric hash join and bind join is as follows. Symmetric hash join provides efficient response time by being a non-blocking join method. Bind join provides efficient completion time under some conditions, as mentioned in previous chapters. Besides, it is the most popular join method among the query federation engines.

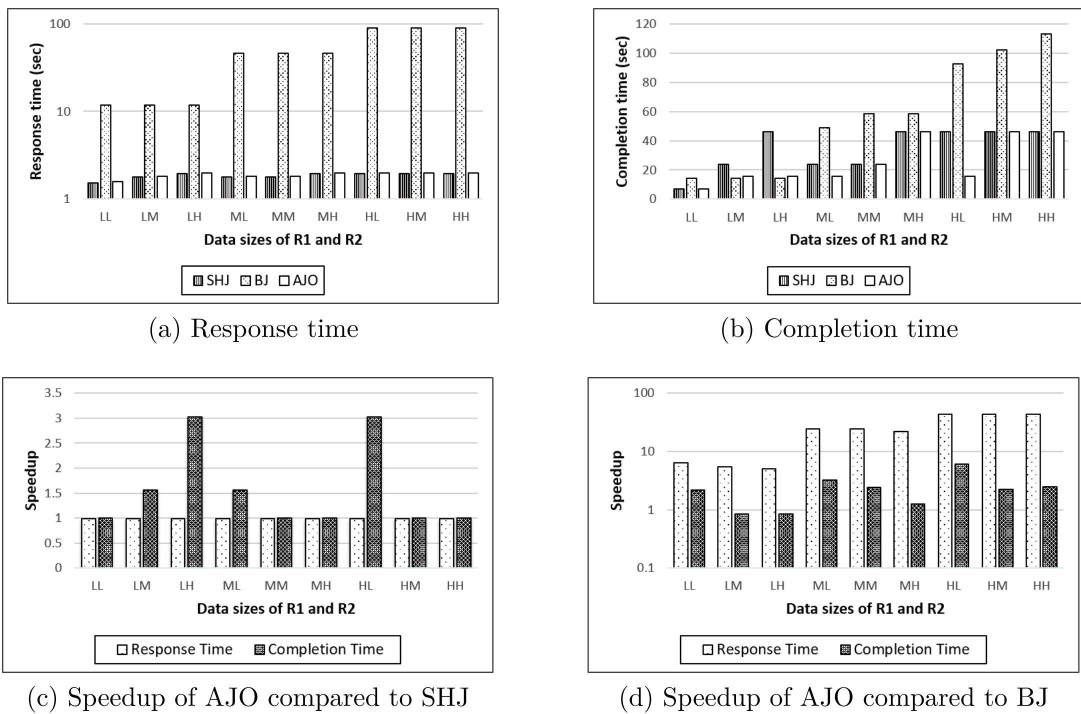
4.2.1 Performance Evaluation for Single Join Queries

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 and data arrival rates in the first and the second case, respectively.

4.2.1.1 Impact of Data Sizes

In this case, we fixed the data arrival rates of both endpoints to 0.5 Mbps. In order to analyze the impact of data sizes on the behaviours of SHJ, BJ, and AJO, we calculated their response times and completion times 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.

Figure 4.1 depicts the behaviours of SHJ, BJ, and AJO with different data size conditions while the data arrival rates of both relations are fixed. As shown in Figure



(a) Response time

(b) Completion time

(c) Speedup of AJO compared to SHJ

(d) Speedup of AJO compared to BJ

Figure 4.1: Data arrival rates of $R1$ and $R2$ are fixed

4.1.a, BJ has the worst response time for all conditions, while SHJ and AJO behave similar to each other. As the data size of $R1$ increases, the response time of BJ increases as well due to waiting for the arrival of all tuples 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.

Completion time of BJ is shorter than others when the cardinality of $R1$ is low and the cardinality of $R2$ is medium or high, as shown in Figure 4.1.b. 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 Figure 4.1.c. 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 Figure 4.1.d displays, compared to BJ, AJO provides speedup in response time from 5.9 times to 45.5 times. AJO also provides speedup in completion time up to 6 times except two cases.

4.2.1.2 Impact of Data Arrival Rates

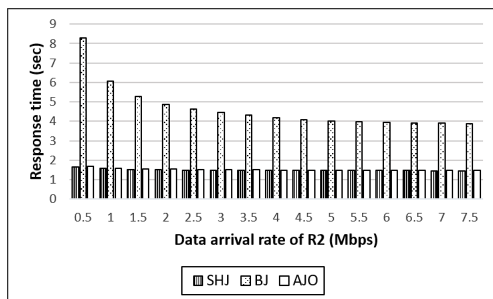
In this case, we fixed the data arrival rate of $R1$ to 2 Mbps and changed the data arrival rate of $R2$. We conducted the simulations for two different cardinality options: i) low cardinality of $R1$ and high cardinality of $R2$; ii) high cardinality of $R1$ and low cardinality of $R2$.

Low Cardinality of R1 and High Cardinality of R2

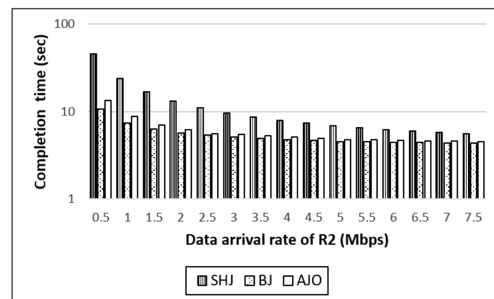
As Figure 4.2.a shows, SHJ and AJO provide almost the same response time. On the other hand, the response time of BJ is always longer than SHJ's and AJO's. The gap between the response times of BJ and the others increases when the data arrival rate of $R2$ gets slower.

Figure 4.2.b displays the completion times of SHJ, BJ, and AJO. BJ provides shorter completion times than others in all data arrival rate conditions because the first relation's cardinality is low. However, AJO always provides shorter completion time than SHJ due to changing the join method as BJ during the execution. As the data arrival rate of the second relation gets faster, the difference between BJ and others decreases.

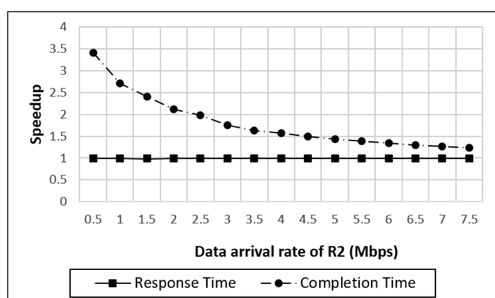
[2] Speedup of x compared to y (response time) = response time of y / response time of x
Speedup of x compared to y (completion time) = completion time of y / completion time of x



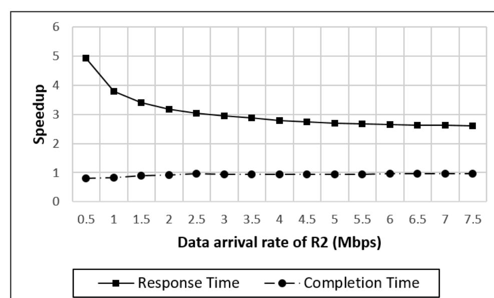
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to SHJ



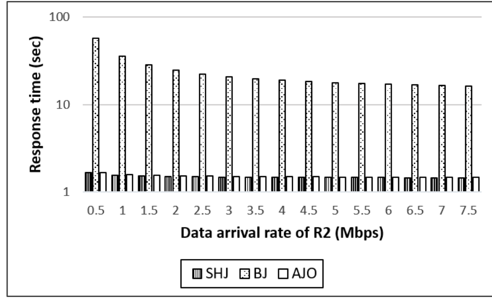
(d) Speedup of AJO compared to BJ

Figure 4.2: Data sizes of $R1$ and $R2$ are fixed with $card(R1) \ll card(R2)$

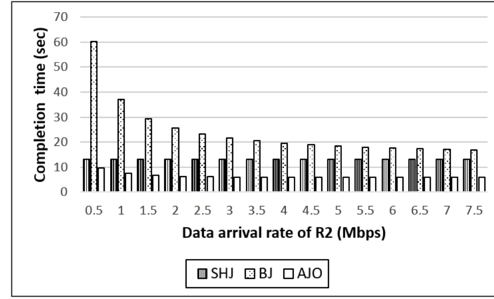
As shown in Figure 4.2.c, compared to SHJ, AJO has almost the same response time, however it can provide speedup in completion time up to 3.4 times. Although the speedup decreases while the second relation's data arrival rate increases, we expect it to be nearly 1 in the worst case. The reason of this is based on the working principal of AJO. It changes the join method to BJ when it estimates that BJ is more efficient than SHJ. Otherwise, AJO does not change the join method; it continues with SHJ. 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 illustrated in Figure 4.2.d.

High Cardinality of R1 and Low Cardinality of R2

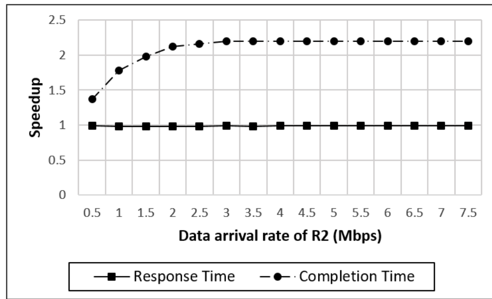
The results observed from Figure 4.3.a are similar to the results in Figure 4.2.a. Since the cardinality of the first relation is high in this case, the response time of BJ is



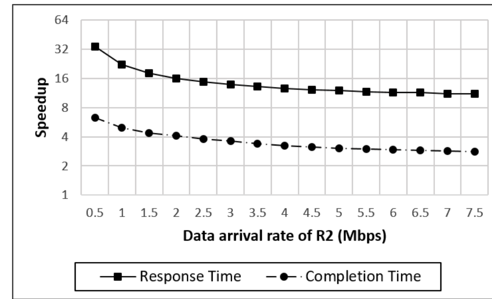
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to SHJ



(d) Speedup of AJO compared to BJ

Figure 4.3: Data sizes of $R1$ and $R2$ are fixed with $card(R1) \gg card(R2)$

dramatically longer than SHJ's and AJO's. The response times of SHJ and AJO are nearly the same.

As shown in Figure 4.3.b, 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 data arrival rate conditions. It changes the join method to BJ and the join order when all the tuples of the second relation arrive.

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 illustrated in Figure 4.3.c. Compared to BJ, AJO improves both the response time and the completion time as displayed in 4.3.d. 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.

4.2.1.3 Discussion on the Performance Evaluation

Simulation results showed that SHJ performs the best response time because it can generate the first result tuple as soon as possible. AJO has the same advantage in response time since it always uses SHJ in the beginning. BJ provides longer response time because it has the disadvantage of waiting the results of the first relation. As the cardinality of the first relation increases, this disadvantage becomes more evident.

BJ can provide shorter completion time when the cardinality of the first relation is low. The gap between SHJ and BJ increases as the cardinality of the other relation increases. On the other hand, AJO can change the join method to BJ in these cases.

To conclude, SHJ provides the shortest response time, whereas the owner of the best performance in completion time is changed according to the cardinalities of relations and data arrival rates. AJO provides optimal response time due to beginning with SHJ. On the other hand, AJO can change the join method to BJ if it decides that it provides shorter completion time than SHJ. It can also change the join order in order to minimize the completion time. In brief, AJO provides optimal response time and completion time for single join queries.

4.2.2 Performance Evaluation for Multi-Join Queries

In this subsection, we analyze the performances of multi-way symmetric hash join (MSHJ), BJ and AJO when there are three relations in the query.

Listing 4.1 displays a query example that we use in our experiments. $R1$ ($service1$) 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 . } }
```

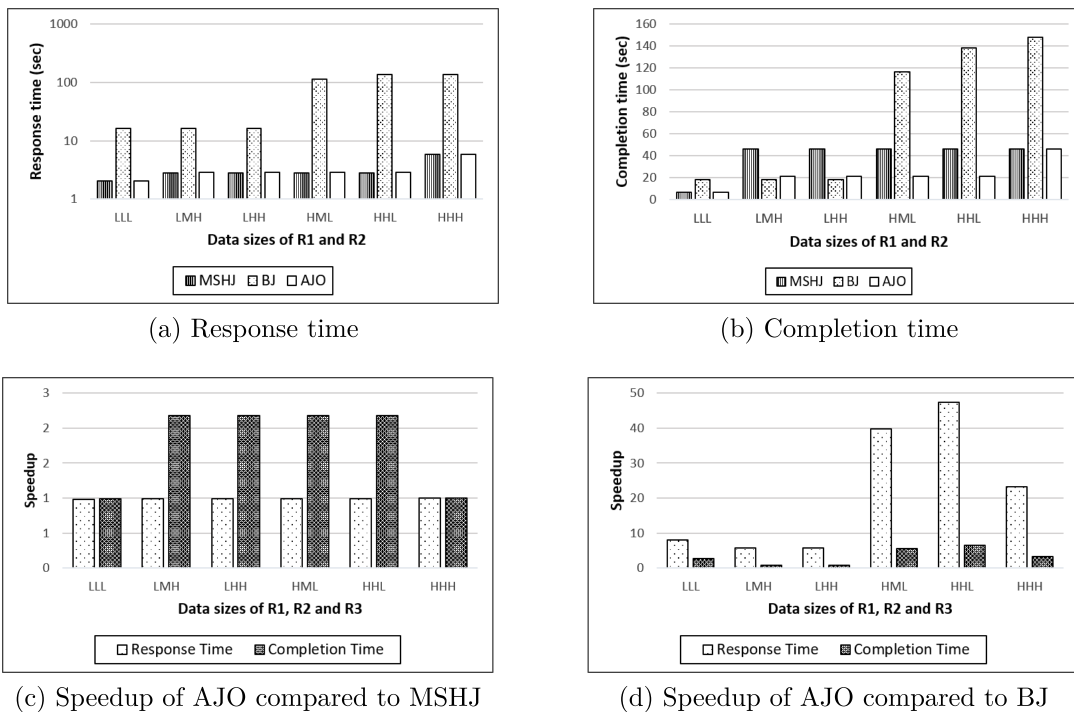
Listing 4.1: Query example

4.2.2.1 Impact of Data Sizes

In order to show the impact of data sizes on the behaviours of MSHJ, BJ, and AJO, we fixed the data arrival rates of all relations to 0.5 Mbps. 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 Figure 4.4.a shows, the response times of MSHJ and AJO are almost the same, whereas BJ's response time is substantially longer in cardinality conditions. On the other hand, as illustrated in Figure 4.4.b, BJ provides the best completion time when the first relation's cardinality is low. However, AJO's completion time is quite similar because it can change the join method to bind join in these conditions. When the first relation's cardinality is high, BJ's completion time becomes substantially longer while AJO has the best performance due to changing the join order.

As shown in Figure 4.4.c, compared to MSHJ, AJO has almost the same response time, however it can provide speedup in completion time up to 2.2 times. Speedup comparison between AJO and BJ is displayed in Figure 4.4.d. Compared to BJ, AJO degrades completion time 0.85 times when the cardinalities are LMH and LHH, however



(a) Response time

(b) Completion time

(c) Speedup of AJO compared to MSHJ

(d) Speedup of AJO compared to BJ

Figure 4.4: Data arrival rates of $R1$, $R2$ and $R3$ are fixed

it provides speedup in completion times in other conditions. The speedup value differs from 2.62 times to 6.56 times. In addition, AJO provides speedup in response time in all conditions, which is between 5.75 and 47.38 times.

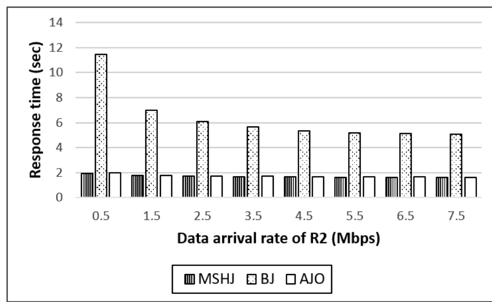
4.2.2.2 Impact of Data Arrival Rates

Our aim in this case is to show the effect of different data arrival rates on the performances of MSHJ, BJ, and AJO. For this reason, we fixed the data arrival rates of $R1$ and $R3$ to 2 Mbps and changed the data arrival rate of $R2$

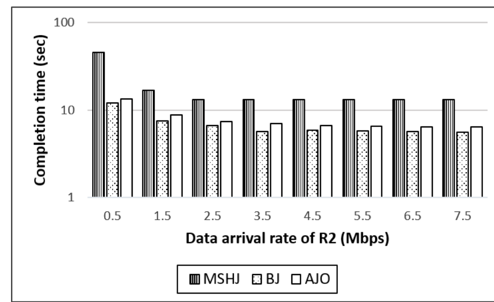
We conducted the simulations for two different cardinality options: i) low cardinality of $R1$, high cardinality of $R2$, and high cardinality of $R3$ (LHH); ii) high cardinality of $R1$, high cardinality of $R2$ and low cardinality of $R3$ (HHL).

Low Cardinality of R1, High Cardinality of R2, and High Cardinality of R3

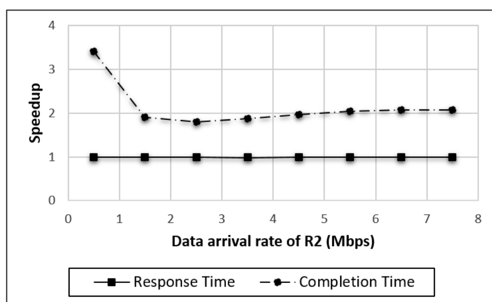
In this case, as displayed in Figure 4.5.a, BJ has the worst response time in all data arrival rates of $R2$, while MSHJ and AJO have almost the same response time. However, as shown in Figure 4.5.b, BJ's completion time is shorter than MSHJ's completion time which has the disadvantage of waiting all 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 changes the join method to BJ for $R1$ and $R2$, and ($R1 \bowtie R2$) and $R3$ during the execution. The reason of the success of BJ in completion time is related to the low cardinality of the first relation.



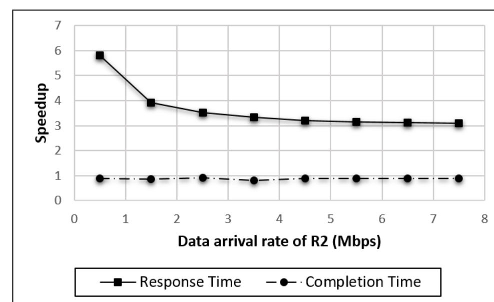
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to MSHJ



(d) Speedup of AJO compared to BJ

Figure 4.5: Data sizes of $R1$, $R2$, $R3$ are fixed with $card(R1) \ll card(R2) = card(R3)$

Figure 4.5.c illustrates the speedup of AJO compared to MSHJ with respect to the response time and the completion time. MSHJ's completion time is related to the time

of the latest arrival of the relation. Hence MSHJ's completion time is related to the arrival times of $R2$ and $R3$ because the data arrival rates of $R1$ and $R2$ are the same (2 Mbps), while the cardinalities of them are low and high, respectively. Its completion time does not change when the data arrival of $R2$ is equal or faster than 2 Mbps. In other words, the completion time of MSHJ remains the same after this data arrival rate of $R2$ because the tuples of $R3$ arrive lastly. On the other hand, AJO's completion time decreases as the data arrival of $R2$ increases. Compared to MSHJ, AJO has almost the same response time but it can provide speedup in completion time up to 3.4 times.

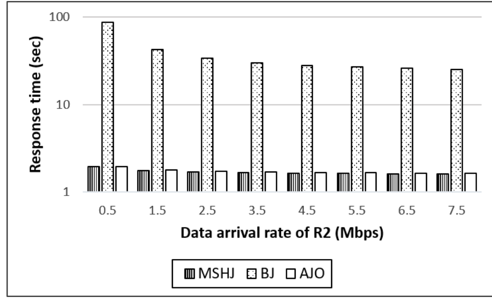
As Figure 4.5.d shows, compared to BJ, AJO degrades the completion time up to 0.8 times, it can improve the response time up to 3.9 times. Although BJ provides shorter completion time since the first relation cardinality is low, AJO can decide to change the join method to BJ during the execution.

High Cardinality of R1, High Cardinality of R2, and Low Cardinality of R3

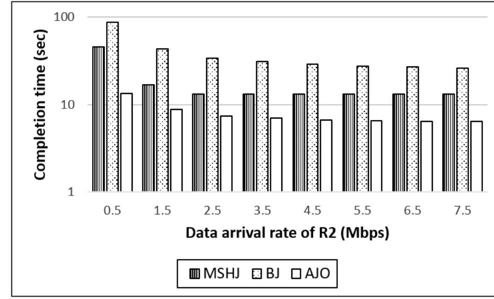
The results observed from Figure 4.6.a are similar to the results in Figure 4.5.a. BJ performs the worst response time again, whereas MSHJ and AJO have almost the same response time. However, the gap between the response times of BJ and the others' are dramatically high because the first relation's cardinality is high. As the data arrival rate of the second relation increases, response times of all of them decreases.

Figure 4.5.b compares performances of MSHJ, BJ, and AJO with respect to their completion times. AJO has the best completion time in all conditions. The completion times of AJO and BJ decreases as the data arrival rate of $R2$ increases. On the other hand, the completion time of MSHJ remains constant when the data arrival rate of $R2$ is more than 2 Mbps since $R1$'s cardinality is high and its data arrival rate is 2 Mbps.

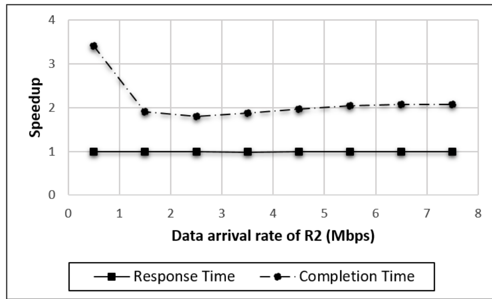
Compared to MSHJ, AJO has almost the same response time but it can provide



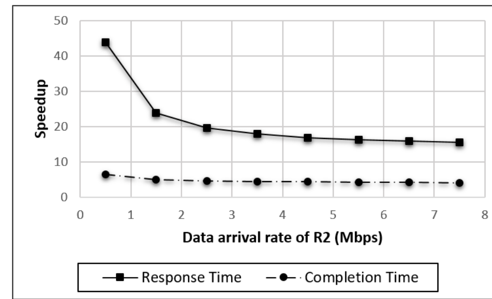
(a) Response time



(b) Completion time



(c) Speedup of AJO compared to MSHJ



(d) Speedup of AJO compared to BJ

Figure 4.6: Data sizes of $R1$, $R2$, $R3$ are fixed with $card(R1) = card(R2) \gg card(R3)$

speedup in completion time up to 3.4 times as shown in Figure 4.6.c. Compared to BJ, AJO improves both the response time and the completion time up to 43.9 times and 6.5 times, respectively, as illustrated in Figure 4.6.d.

4.2.2.3 Discussion on the Performance Evaluation

We analyzed the impact of cardinalities and data arrival rates of relations in this subsection. Simulation results showed that MSHJ provides the best response time in all cases and AJO has almost the same response time due to beginning with MSHJ. The response time of BJ is mostly affected by the first relation's cardinality.

When we focus on the completion time, we see that MSHJ's completion time depends on the data arrival rate of the relation which has the highest cardinality. BJ's completion time is the best when the first relation's cardinality is low, however it per-

forms the worst completion time when the first relation's cardinality is high. On the other hand, AJO has the closest completion time to BJ when the first relation's cardinality is low. AJO can provide the best completion time when the first relation's cardinality is high since it can change the join order and the join method. In both cardinality cases, completion times of both BJ and AJO decrease as the second relation's data arrival gets faster.

In conclusion, AJO provides both optimal response time and completion time for multi-join queries due to beginning with MSHJ and having the ability to change the join method during the execution. The adaptive join operator can also change the join order.

4.3 Performance Evaluation of Extended Adaptive Join Operator

In this section, we analyze and evaluate performances of symmetric hash join (or 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. Focus of the evaluation is on their performances with respect to the response time and the completion time since the goal of query optimization in query federation is to minimize them both. Speedup[3] 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.

Although we assume that the common attribute values are unique in the performance evaluation of the adaptive join operator in Section 4.2, we consider the possibility

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

of including duplicate values on the common attributes of relations in the performance evaluation of the extended adaptive join operator. 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, whereas 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 bind-bloom join.

4.3.1 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. Our aim is to show the impact of data sizes in the first case, while 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. The aim in this case is to show the impact of bit vector size for the extended adaptive join operator.

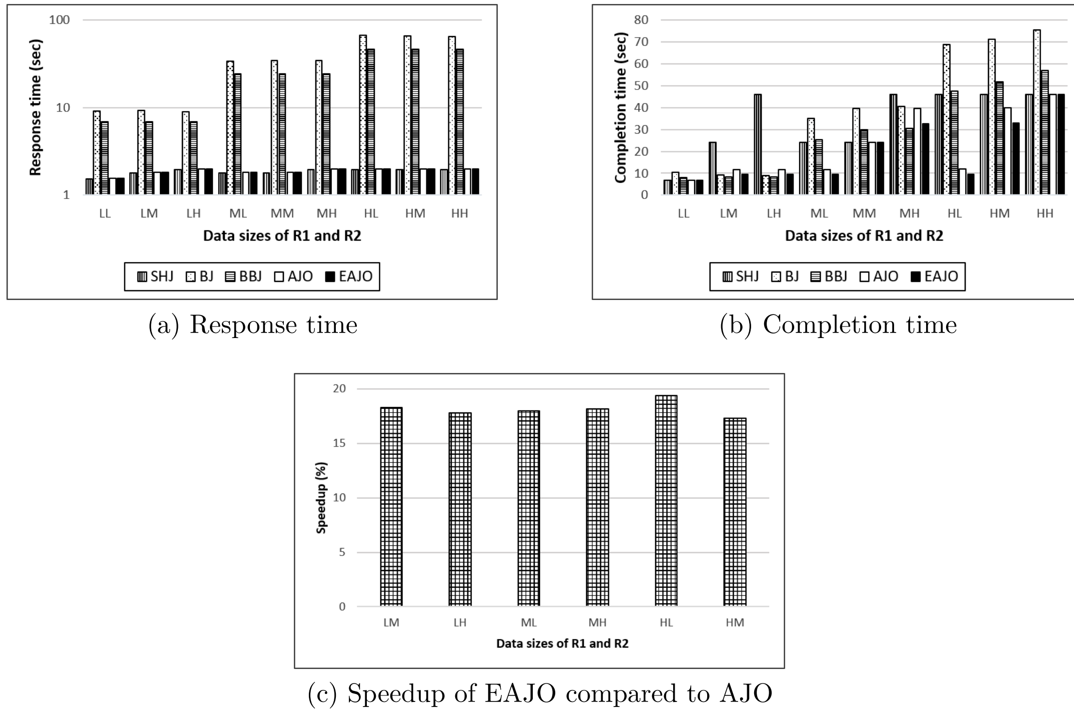
4.3.1.1 Impact of Data Sizes

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 calculated the response times and the completion times 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 4.7.a shows, for all conditions, BJ and BBJ have longer response times than SHJ, AJO, and EAJO which 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 $R2$. As a result of using a bloom filter for sending the common attributes in BBJ, it provides a slightly better response time than BJ. 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.

BBJ's completion time is always shorter than BJ's due to the bloom filter usage as illustrated in 4.7.b. 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, low-low, medium-medium, high-high, 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. Both AJO and EAJO, therefore, decide to continue with SHJ in such cases. To conclude the



(a) Response time

(b) Completion time

(c) Speedup of EAJO compared to AJO

Figure 4.7: Data arrival rates of $R1$ and $R2$ are fixed

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 has the most similar completion time to the best join method in the remaining three conditions.

Figure 4.7.c shows the achieved speedup in completion time by EAJO compared to AJO. As shown in the figure, EAJO provides speedup between 17.8% and 19.4% when the cardinalities of relations are different. 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.

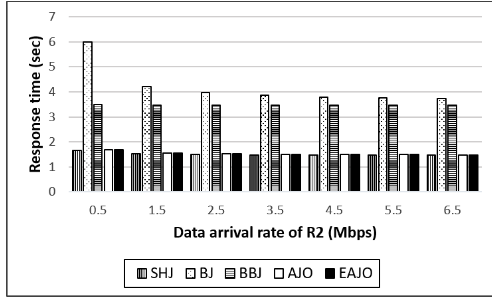
4.3.1.2 Impact of Data Arrival Rates

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 in speedup comparison between EAJO and AJO in order to understand the impact of the duplication factors as well.

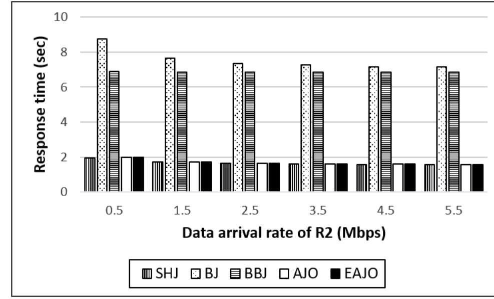
Low Cardinality of R1 and High Cardinality of R2

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 4.8.a and 4.8.b show, BJ's and BBJ's response times are always longer than the response times of 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. Thus, SHJ, AJO, and EAJO are the best in terms of response time at the same time.

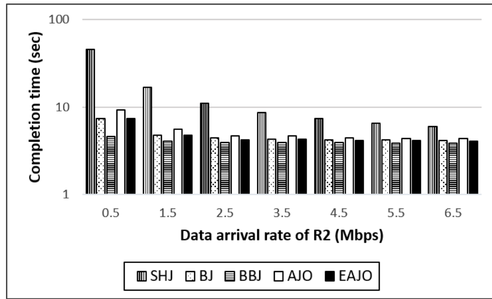
As displayed in Figure 4.8.c, BBJ's completion time is always shorter than BJ's due to the usage of bloom filter. For this reason, we consider the completion time of BBJ instead of the completion time of BJ when we compare the completion times of operators. BBJ provides the shortest completion time in 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.



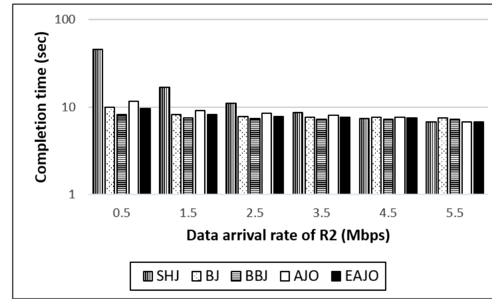
(a) Response time when data arrival rate of $R1$ is fixed to 2 Mbps and data arrival rate of $R2$ is changed



(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

Figure 4.8: Data sizes of $R1$ and $R2$ are fixed with $card(R1) \ll card(R2)$

Figure 4.8.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 because it has the 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 changing the join method to SHJ. In brief, the winner of the completion time is changed according to the data arrival rates. However, EAJO can choose the best join method during the execution.

Table 4.1 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 from 0.5 Mbps to 6.5 Mbps. 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 4.1: Speedup of EAJO compared to AJO when $card(R1) \ll card(R2)$ and the data arrival rate of $R1$ is 2 Mbps

Data arrival rate of R2 in Mbps	**Average duplication factors**		
	1	2	5
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 4.2 shows the speedup gained by EAJO when the 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 4.1 and Table 4.2, the speedup decreases as the average duplication factors increase.

Table 4.2: Speedup of EAJO compared to AJO when $card(R1) \ll card(R2)$ and the data arrival rate of $R1$ is 0.5 Mbps

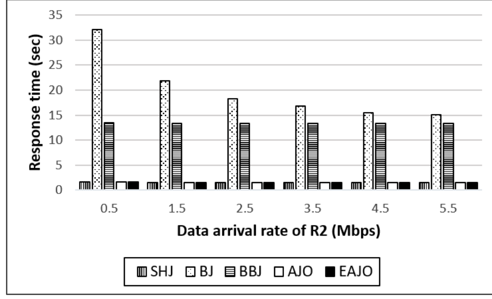
Data arrival rate of R2 in Mbps	**Average duplication factors**		
	1	2	5
0.5	30.61%	18.62%	9.16%
1.5	17.29%	8.72%	4.20%
2.5	12.41%	6.08%	3.12%
3.5	9.75%	4.91%	2.71%
4.5	–	4.27%	2.51%

High Cardinality of R1 and Low Cardinality of R2

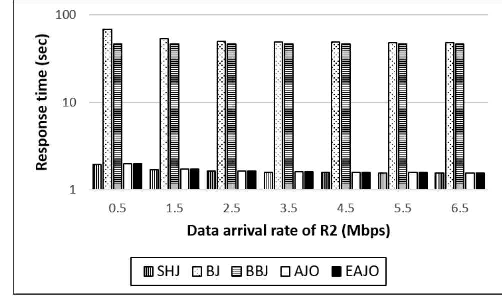
We again conducted the simulations for two different conditions: i) when the data arrival rate of $R1$ is fixed to 2 Mbps, and ii) when the data arrival rate of $R1$ is fixed to 0.5 Mbps.

The results observed from Figure 4.9.a and Figure 4.9.b are similar to the results in Figure 4.8.a and Figure 4.8.b, respectively. Since the cardinality of the first relation is high in this case, response times of BJ and BBJ are substantially longer than SHJ and also longer than AJO and EAJO as expected. The response times of SHJ, AJO, and EAJO are nearly the same.

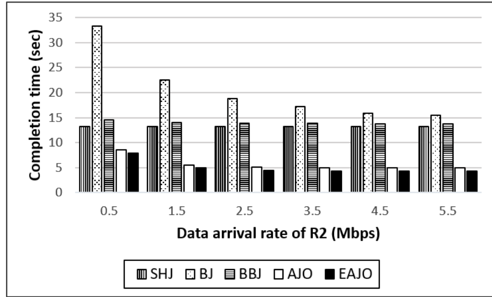
As illustrated in Figure 4.9.c, EAJO provides the best completion time in all data arrival rates of the second relation. SHJ, BJ, and BBJ should wait the arrival of all tuples related to the first relation whose cardinality is high. However, AJO and EAJO can change the join method and the join order when the second relation's tuples all arrive. Compared to AJO, EAJO has the advantage of changing the join method to BBJ. Figure 4.9.d compares the completion times when the first relation's data arrival rate is fixed to 0.5 Mbps. The results are similar to the previous one. EAJO provides



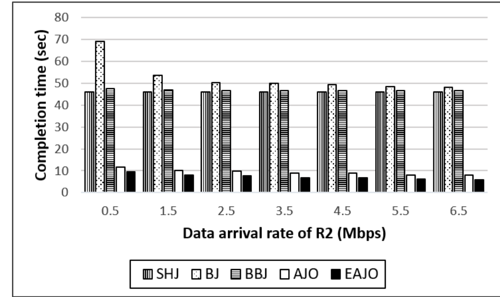
(a) Response time when data arrival rate of $R1$ is fixed to 2 Mbps and data arrival rate of $R2$ is changed



(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

Figure 4.9: Data sizes of $R1$ and $R2$ are fixed with $card(R1) \gg card(R2)$

the shortest completion time once again. The gap between EAJO and the others is even higher.

Table 4.3 and Table 4.4 show the gained speedup in completion time by EAJO compared to AJO. In all conditions, both AJO and EAJO change the join order as $R2 \bowtie R1$. Actually, 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 average duplication factors decrease.

Table 4.3: Speedup of EAJO compared to AJO when $card(R1) \gg card(R2)$ and the data arrival rate of $R1$ is 2 Mbps

Data arrival rate of R2 in Mbps	**Average duplication factors**		
	1	2	5
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 4.4: Speedup of EAJO compared to AJO when $card(R1) \gg card(R2)$ and the data arrival rate of $R1$ is 0.5 Mbps

Data arrival rate of R2 in Mbps	**Average duplication factors**		
	1	2	5
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%

4.3.1.3 Impact of Bit Vector Size

As explained in Section 3.3.1, a bloom filter represents a set $S = \{e_1, e_2, \ldots, e_n\}$ 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. In this part, we aim to 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). Table 4.5 shows the m/n and k combinations used in our experiments.

Table 4.5: The m/n and k combinations used for bloom filter

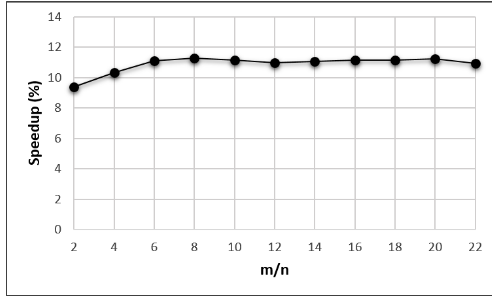
m/n	k
2	1
4	3
6	4
8	6
10	7
12	8
14	10
16	11
18	12
20	14
22	15

In order to analyze the impact of the bit vector size, we set different m/n values while we fixed the data arrival rates of both endpoints to 2 Mbps, and the cardinalities of relations to low and high, respectively.

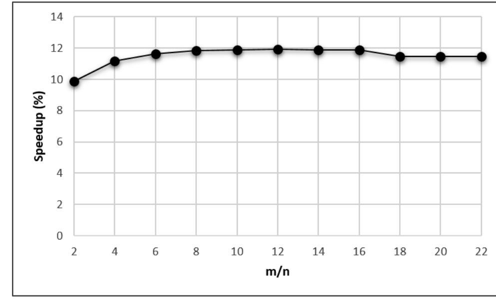
Since AJO does not use a bloom filter, its completion time remained the same in all cases. In other words, we compared the completion times of AJO and EAJO when the data arrival rates of both relations and the cardinalities of relations were fixed, while different m/n values were used in EAJO. First, the 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 4.10.a shows the achieved speedup in completion time by EAJO compared to AJO in different m/n values when the average duplication factors are given randomly. 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 4.10.b shows the gained speedup in completion time by EAJO when the



(a) Random average duplication factors



(b) Fixed average duplication factors

Figure 4.10: Speedup of EAJO compared to AJO when the m/n and k combinations used

average duplication factors are equal to 2. The results are similar to the results in Figure 4.10.a. The speedup values are almost the same when the m/n is between 8 and 16.

4.3.1.4 Discussion on the Performance Evaluation

The simulation results demonstrated that SHJ provides the best response time in all conditions since it is a non-blocking join operator. It produces the first result tuple as early as possible. Our previous and current proposals, namely AJO and EAJO, 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, AJO can change the join method to BJ, and 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 tested.

To conclude, SHJ is the most successful join method with respect to response time. However, the best join method in completion time can differ according to the cardinalities and the data arrival rates of relations. In addition, the results showed that BBJ provides better completion times than BJ in all conditions. Our proposal, EAJO, provides an optimal response time by beginning with SHJ. It provides an optimal completion time by changing the join method or join order during the execution. In brief, EAJO gives the best trade-off between the response time and the completion time. Another key fact to remember is that EAJO always provides better completion time than AJO.

4.3.2 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. We use the same example query in our experiments which is given in Section 4.2.2.

We have conducted our experiments in two main cases. Our aim in the first case is to show the impact of data sizes, while we want to show the impact of data arrival rates in the second case.

4.3.2.1 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. 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).

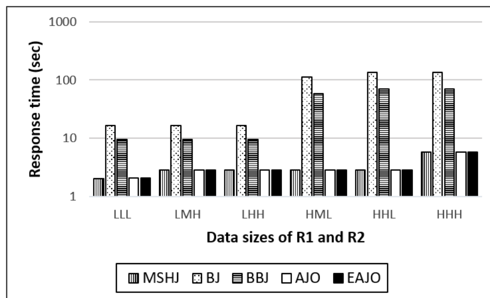
Figure 4.11.a, Figure 4.11.b and Figure 4.11.c compare the response times of MSHJ, BJ, BJBF, AJO, and EAJO when the average duplication factors are 1, 2 and 5,

respectively. In all average duplication factors, MSHJ, AJO, and EAJO provide the best response time, whereas BJ performs the worst one and BBJ follows it. When the cardinality of the first relation is high, the response times of BJ and BBJ become dramatically longer due to waiting for the arrival of all results of the first relation. 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.

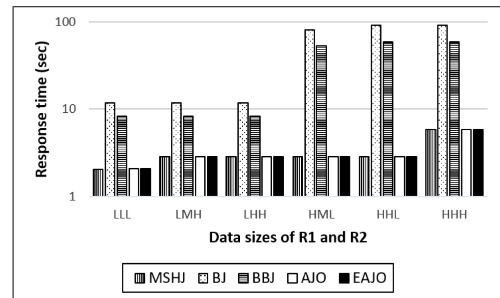
Figures 4.11.d, 4.11.e and 4.11.f show the completion times of MSHJ, BJ, BBJ, AJO, and EAJO. When the cardinalities are HML or HHL, EAJO performs the best completion time and AJO has the closest completion time to it. The difference between EAJO and others, except AJO, is substantially high. When the 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 4.6 displays the speedup in completion time of EAJO compared to AJO when the data arrival rates of $R1$, $R2$ and $R3$ are fixed. As shown in the table, when the cardinalities of relations are different, EAJO provides speedup from 6.40% to 31.33%. Although the speedup is not affected by the cardinalities of relations, it increases as

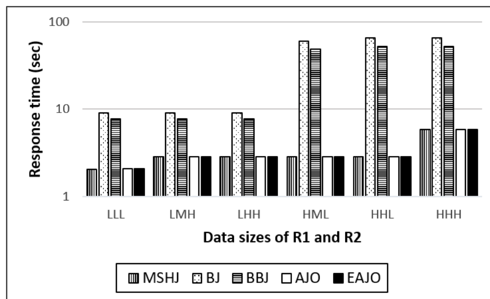
the average duplication factors decrease. EAJO does not provide speedup when the cardinalities of relations are the same, because both AJO and EAJO decide to continue with MSHJ.



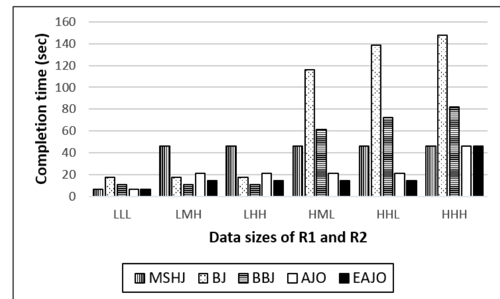
(a) Response time when average duplication factors are 1



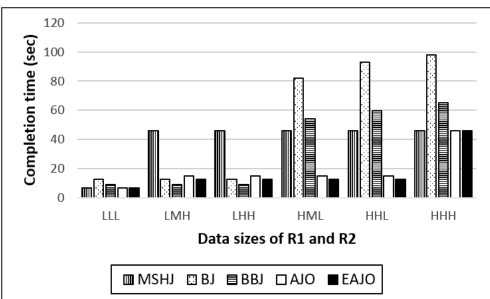
(b) Response time when average duplication factors are 2



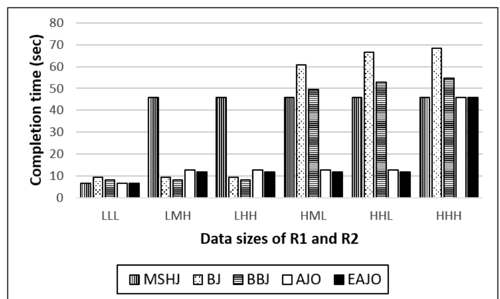
(c) Response time when average duplication factors are 5



(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

Figure 4.11: Data arrival rates of $R1$, $R2$ and $R3$ are fixed

Table 4.6: Speedup of EAJO compared to AJO when data arrival rates are fixed

Data sizes of R1, R2 and R3	**Average duplication factors**		
	1	2	5
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%

4.3.2.2 Impact of Data Arrival Rates

In this case, we fixed the data arrival rates of $R1$ and $R3$ to 2 Mbps, and changed the data arrival rate of $R2$ in order to show the impact of data arrival rates on MSHJ, BJ, BBJ, AJO, and EAJO. 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 experiments.

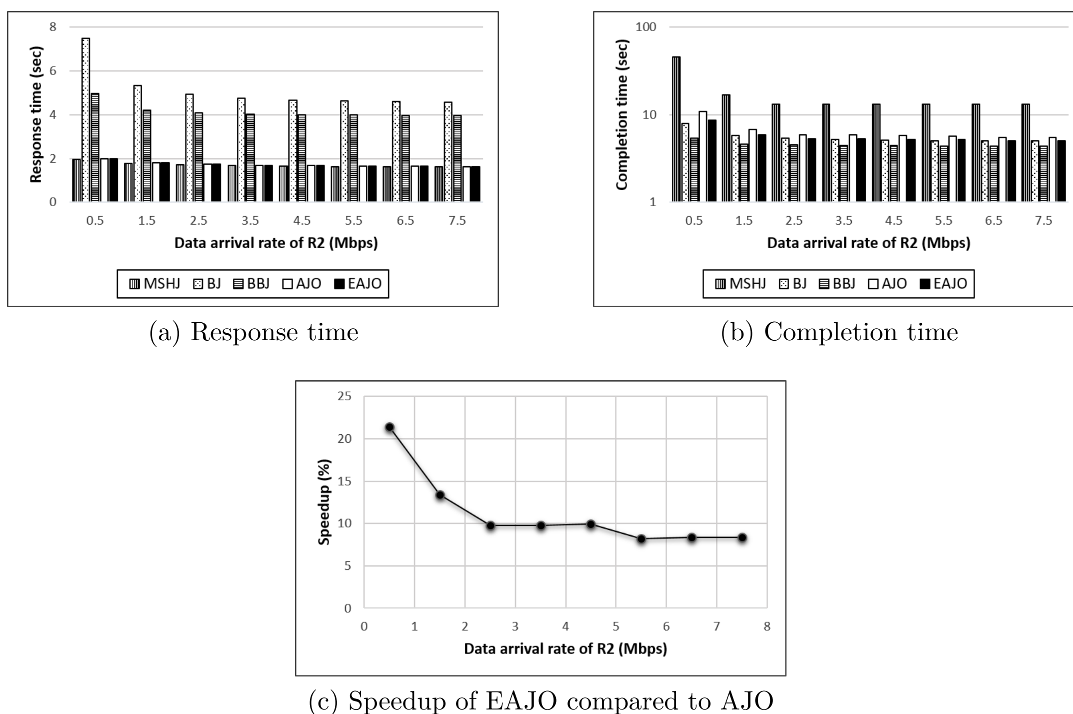
Low Cardinality of R1, High Cardinality of R2, High Cardinality of R3

Figure 4.12.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, while BJ's and BBJ's response times are highly longer than them.

Figure 4.12.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. Therefore, EAJO performs almost the same completion time with 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 4.12.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 $card(R1) \ll card(R2) = card(R3)$. The speedup decreases as the second relation's data arrival rate increases, because the impact of the decrease in the size of the sent data over the network decreases as the network speed increases.



(a) Response time

(b) Completion time

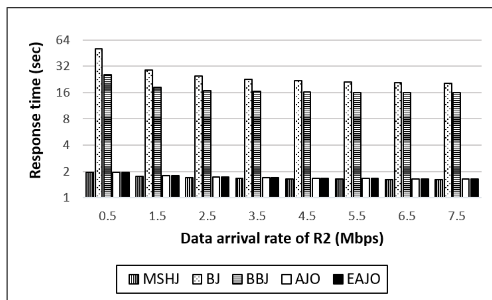
(c) Speedup of EAJO compared to AJO

Figure 4.12: Data sizes of $R1$, $R2$ and $R3$ are fixed with $card(R1) \ll card(R2) = card(R3)$

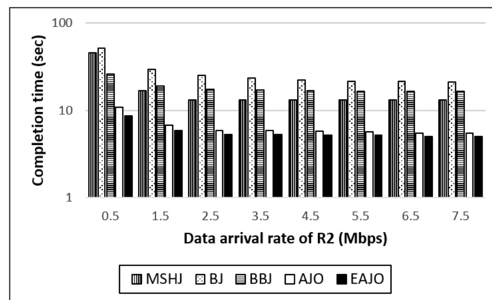
High Cardinality of R1, High Cardinality of R2, Low Cardinality of R3

The results observed from Figure 4.13.a are similar to the results in Figure 4.12.a. BJ and BBJ provide 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.

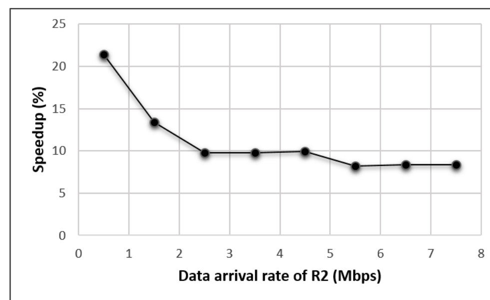
EAJO provides the best completion time in all cases as shown in Figure 4.13.b. The completion times in ascending order are of EAJO, AJO, MSHJ, BBJ, and BJ when the second relation's data arrival rate is equal or faster than 1.5 Mbps. EAJO and AJO have the advantage of using BJ or BBJ when the tuples 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.



(a) Response time



(b) Completion time



(c) Speedup of EAJO compared to AJO

Figure 4.13: Data sizes of $R1$, $R2$ and $R3$ are fixed with $card(R1) = card(R2) \gg card(R3)$

Figure 4.13.c illustrates 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 $card(R1) = card(R2) \gg card(R3)$. Compared to AJO, EAJO provides speedup in completion time due to the usage of bloom filter. It sends less data size through the network. The speedup decreases while 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 4.12.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, the tuples of $R1$ all arrive firstly, 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, the tuples of $R3$ all arrive firstly. As a result, 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.

4.3.2.3 Discussion on the Performance Evaluation

The simulation results showed that MSHJ, which is a non-blocking join method, provides the best response time in all conditions. AJO and EAJO have 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 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 trade-off between the response time and the completion time. We also emphasize that EAJO always provides better completion time than AJO.

4.4 Conclusion

In this chapter, we presented and discussed the performance evaluations of adaptive join operator and extended adaptive join operator for single join and multi-join queries.

The results of the performance evaluation showed the efficiency of the proposed operators. Both of them have almost the same response time with symmetric hash join and multi-way symmetric hash join, but they can provide faster completion times. Compared to bind join, adaptive join operator performs substantially better with respect to the response time and can also improve the completion time. Extended adaptive join operator performs substantially better with respect to the response time than both bind join and bind-bloom join, and it can also improve the completion time. Moreover, both operators have the adaptation ability to different data arrival rates.

Extended adaptive join operator has the same response time with adaptive join operator. However, it provides faster completion times in all conditions, because it utilizes the bloom filter for sending the common attributes to the other endpoint. Ex-

perimental results also showed that bind-bloom join provides better completion times than bind join in all conditions. These results allow us to suggest that using bloom filters in bind join.

In conclusion, adaptive join operator provides optimal response time and completion time for single join queries and multi-join queries. Furthermore, the extended version of the adaptive join operator succeeds to further reduce the completion time.

Chapter 5

Conclusion and Future Work

Abstract: In this chapter, we review the presented work in this thesis, highlighting the proposed methods and our contributions. We discuss the performance evaluation and finally we conclude the thesis by presenting possible future work.

Contents

5.1 Thesis Review **137**

5.2 Future Work **140**

5.1 Thesis Review

Linked Data, which is the fundamental part of the Web of Data, evolves the current Web into a huge global data space. Since this data space is distributed on the Web, query optimization is one of the most important research topics in federated query processing on Linked Data. The objective of query optimization is to minimize the response time and the completion time. Response time is the time to generate the first result tuple, while completion time is the time to provide all result tuples. The communication cost is the dominant cost in them both, hence the goal of query optimization in federated query processing can be described as to minimize the communication cost. For this reason, this thesis focuses on minimizing the communication time belonging to the response time and the completion time for query federation.

Federated queries are executed over the SPARQL endpoints of the Linked Data sources on the Web. There are various challenges in this distributed environment such as inaccurate or missing statistics, and different data arrival rates of relations. We think that adaptive query optimization (Deshpande et al., 2007) should be used in order to manage these challenges. Although there are various federated query engines which use static query optimization (Quilitz and Leser, 2008; Görlitz and Staab, 2011b; Schwarte et al., 2011; Akar et al., 2012; Wang et al., 2013; Yönyül, 2014), there are a few engines which consider adaptive query optimization (Acosta et al., 2011; Lynden et al., 2010, 2011). There is another study which considers adaptive query optimization in some way, called AVALANCHE (Basca and Bernstein, 2010, 2014). Some of these adaptive studies aim to minimize the response time (Acosta et al., 2011; Basca and Bernstein, 2010, 2014), whereas the others aim to minimize the completion time (Lynden et al., 2010, 2011). To the best of our knowledge, the work in this thesis is the first study in query federation over SPARQL endpoints that aims to minimize them both.

In this thesis, we first surveyed the literature of federated query processing on Linked Data and presented the major challenges in this research topic (Oguz et al., 2015). We believe that this survey contributes to the existing literature and we hope that it will be useful for future research in this area.

Second, we focused on adaptive query optimization, which is one of the challenges mentioned in the literature survey. We proposed an adaptive join operator (AJO) (Oguz et al., 2016) in order to minimize both the response time and the completion time. This operator handles different data arrival rates of relations and missing statistics. AJO begins with symmetric hash join (SHJ) (Wilschut and Apers, 1991) in order to minimize the response time. It considers changing the join method to bind join (BJ) (Haas et al., 1997) when all the tuples of a relation arrive. Hence it can change the join order and the join method during the execution. Moreover, our proposal works without requiring the predefined statistics.

Finally, we proposed an extended version of adaptive join operator (EAJO) (Oguz et al., In press) which aims to further reduce the completion time by employing bind-bloom join (BBJ) (Basca and Bernstein, 2014; Groppe et al., 2015) to minimize the communication time. We presented both our proposals for single join and multi-join queries.

In the performance evaluation of this thesis, we compared AJO with SHJ and BJ with respect to the response time and the completion time. The reason of comparing AJO with SHJ and BJ is as follows. SHJ is a non-blocking join operator, hence it minimizes the response time. On the other hand, BJ can minimize the completion time in some conditions and also it is the most popular join method among the federated query engines.

We included BBJ and AJO to the compared methods in the performance study

of EAJO. Response time and completion time were once again chosen as evaluation metrics to show success in query optimization. We evaluated the mentioned rival operators in different cases in order to show the impact of data sizes and the impact of data arrival rates on their performances.

The performance evaluation of this thesis showed the efficiency of the proposed operators. Since SHJ and MSHJ are non-blocking join methods, they provide the shortest response times for single and multi-join queries, respectively. AJO and EAJO have almost the same response time with SHJ or MSHJ because of using these join methods in the beginning. The response times of BJ and BBJ are mostly affected by the cardinality and the data arrival rate of the first relation. Their response times are dramatically longer because of waiting the arrival of all tuples belonging to the first relation. As the cardinality of the first relation increases or as the data arrival rate of the first relation decreases, this disadvantage becomes more evident. The most successful join method in completion time can differ according to the cardinalities and the data arrival rates of relations. BJ and BBJ can provide shorter completion times when the cardinality of the first relation is low. The gap between them and SHJ or MSHJ increases as the cardinality of the other relation increases. In addition, BBJ provides shorter completion times than BJ in all conditions.

AJO and EAJO provide almost the same response time with SHJ and MSHJ, and they can provide faster completion times. Compared to BJ, our proposals perform substantially better with respect to the response time and can provide faster completion times. In addition, EAJO provides substantially faster response time than BBJ and can improve the completion time as well. Moreover, EAJO provides faster completion times than AJO in all conditions.

In conclusion, the proposed operators provide the best trade-off between the re-

sponse time and the completion time. The performance evaluation revealed that they are successful in both fixed and different data arrival rates, even though our main objective is to manage different data arrival rates of relations.

5.2 Future Work

As a future work, we are motivated to consider the case where a relation is distributed over multiple sources. In this case, we should not only deal with the different data arrival rates of SPARQL endpoints belonging to the relations, but also the different data arrival rates of SPARQL endpoints belonging to the multiple sources of each relation.

Our current adaptive join operator calculates the remaining times for possible join methods when all the tuples of a relation arrive. We plan to extend it with additional feedback such as changes in the data arrival rates of relations during the execution. In the current study, we consider the data arrival rates of relations when a SPARQL endpoint completes the data transfer. By this extension, we can improve the frequency of feedback of our operator and consider the changes in the data arrival rates of all relations. Other possible perspectives are to focus on metadata management and caching results which are the other presented challenges in federated query processing on Linked Data discussed in Section 2.2.5.

Verborgh et al. (2014) proposed triple pattern fragments which provide a new way of publishing Linked Data on the Web. A triple pattern fragment is defined as a Linked Data Fragment with a triple pattern as selector, count metadata, and the controls to retrieve other triple pattern fragments of the dataset[1]. The authors stated that client-side query processing using triple pattern fragments provides live data as

[1]http://linkeddatafragments.org/in-depth/#tpf

query processing over SPARQL endpoints. They also remarked that it handles some challenges of the endpoints such as low bandwidth and high server cost. It could be an interesting future research topic to study query optimization for queries over triple pattern fragments.

Bibliography

Maribel Acosta and Maria-Esther Vidal. *The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, chapter Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, pages 111–127. Springer International Publishing, 2015. doi: 10.1007/978-3-319-25007-6_7.

Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *The Semantic Web – ISWC 2011*, volume 7031 of *Lecture Notes in Computer Science*, pages 18–34. Springer Berlin Heidelberg, 2011.

S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *SIGMOD Rec.*, 25(2):137–146, June 1996. doi: 10.1145/235968.233327.

Ziya Akar, Tayfun Gökmen Halaç, Erdem Eser Ekinci, and Oğuz Dikenelli. Querying the Web of Interlinked Datasets using VOID Descriptions. In *Linked Data on the Web (LDOW2012)*, 2012.

Keith Alexander and Michael Hausenblas. Describing linked datasets - on the design and usage of void, the "vocabulary of interlinked datasets". In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, 2009.

Laurent Amsaleg, Michael J. Franklin, and Anthony Tomasic. Dynamic Query Operator Scheduling for Wide-Area Remote Access. *Distributed and Parallel Databases*, 6(3):217–246, 1998. doi: 10.1023/A:1008646115473.

Jean-Paul Arcangeli, Abdelkader Hameurlain, Frédéric Migeon, and Franck Morvan.

Mobile Agent Based Self-Adaptive Join for Wide-Area Distributed Query Processing. *Journal of Database Management (JDM)*, 15(4):25–44, 2004.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*, ISWC'07/ASWC'07, pages 722–735, Berlin, Heidelberg, 2007. Springer-Verlag.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.

Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive Re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD'05, pages 107–118, New York, NY, USA, 2005. ACM. doi: 10.1145/1066157.1066171.

Cosmin Basca and Abraham Bernstein. Avalanche: Putting the Spirit of the Web Back into Semantic Web Querying. In *Proceedings of the 2010 International Conference on Posters & Demonstrations Track - Volume 658*, ISWC-PD'10, pages 177–180, Aachen, Germany, Germany, 2010. CEUR-WS.org.

Cosmin Basca and Abraham Bernstein. Querying a messy web of data with Avalanche. *J. Web Sem.*, 26:1–28, 2014. doi: 10.1016/j.websem.2014.04.002.

Tim Berners-Lee. Linked data - design issues. `https://www.w3.org/DesignIssues/LinkedData.html`, 2006. Online, accessed 2017-02-07.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. `https://tools.ietf.org/html/rfc3986`, 2005. Online, accessed 2017-04-04.

Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-Based Routing: Different Plans for Different Data. In *Proceedings of the 31st International*

Conference on Very Large Data Bases, VLDB'05, pages 757–768. VLDB Endowment, 2005.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

Eduardo Blanco, Yudith Cardinale, and Maria-Esther Vidal. Experiences of sampling-based approaches for estimating QoS parameters in the Web Service composition problem. *IJWGS*, 8(1):1–30, 2012.

Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. *An Improved Construction for Counting Bloom Filters*, pages 684–695. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi: 10.1007/11841036_61.

B. Barla Cambazoglu, Ismail Sengor Altingovde, Rifat Ozcan, and Özgür Ulusoy. Cache-Based Query Processing for Search Engines. *ACM Transactions on the Web (TWEB)*, 6(4):14, 2012. doi: 10.1145/2382616.2382617.

Angelos Charalambidis, Stasinos Konstantopoulos, and Vangelis Karkaletsis. Dataset Descriptions for Optimizing Federated Querying. In *Proceedings of the 24th International Conference on World Wide Web*, WWW'15 Companion, pages 17–18, New York, NY, USA, 2015a. ACM. doi: 10.1145/2740908.2742779.

Angelos Charalambidis, Antonis Troumpoukis, and Stasinos Konstantopoulos. SemaGrow: Optimizing Federated SPARQL Queries. In *Proceedings of the 11th International Conference on Semantic Systems*, SEMANTICS'15, pages 121–128, New York, NY, USA, 2015b. ACM. doi: 10.1145/2814864.2814886.

Richard Cyganiak, Jun Zhao, Keith Alexander, and Michael Hausenblas. Describing Linked Datasets with the VoID Vocabulary. `http://rdfs.org/ns/void/`, 2011. Online, accessed 2017-04-04.

Amol Deshpande. An Initial Study of Overheads of Eddies. *SIGMOD Rec.*, 33(1): 44–49, March 2004. doi: 10.1145/974121.974129.

Amol Deshpande and Joseph M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB'04, pages 948–959. VLDB Endowment, 2004.

Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, January 2007.

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, Jun 2000.

Roy Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616. `https://tools.ietf.org/html/rfc2616`, 1999. Online, accessed 2017-04-04.

Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Rec.*, 28(2):311–322, June 1999.

Kevin Ford. LC Classification as Linked Data. *Italian Journal of Library and Information Science*, 4(1):161–175, 2013.

Qingqing Gan and Torsten Suel. Improved Techniques for Result Caching in Web Search Engines. In *Proceedings of the 18th International Conference on World Wide Web*, WWW'09, pages 431–440. ACM, 2009. doi: 10.1145/1526709.1526768.

Georges Gardarin and Patrick Valduriez. *Relational Databases and Knowledge Bases.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

Olaf Görlitz and Steffen Staab. Federated data management and query optimization for linked open data. In Athena Vakali and Lakhmi C. Jain, editors, *New Directions in Web Data Management 1*, volume 331 of *Studies in Computational Intelligence*, pages 109–137. Springer Berlin Heidelberg, 2011a.

Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, volume 782 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011b.

Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. *Adaptive Query Processing: A Survey*, pages 11–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi: 10.1007/3-540-45495-0_2.

Sven Groppe, Dennis Heinrich, and Stefan Werner. Distributed join approaches for W3C-conform SPARQL endpoints. *Open Journal of Semantic Web (OJSW)*, 2(1): 30–52, 2015.

Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB'97, pages 276–285. Morgan Kaufmann Publishers Inc., 1997.

Peter Haase, Tobias Mathäß, and Michael Ziller. An Evaluation of Approaches to Federated Query Processing over Linked Data. In *Proceedings of the 6th International Conference on Semantic Systems*, I-SEMANTICS '10, pages 5:1–5:9, New York, NY, USA, 2010. ACM. doi: 10.1145/1839707.1839713.

Wook-Shin Han, Jack Ng, Volker Markl, Holger Kache, and Mokhtar Kandil. Progressive Optimization in a Shared-nothing Parallel Database. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD'07, pages 809–820, New York, NY, USA, 2007. ACM. doi: 10.1145/1247480.1247569.

Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation. `https://www.w3.org/TR/sparql11-query/`, 2013. Online, accessed 2017-04-04.

Olaf Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Proceedings of the 8th Extended Semantic Web Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'11, pages 154–169, Berlin, Heidelberg, 2011. Springer-Verlag.

Olaf Hartig. SQUIN: A Traversal Based Query Execution System for the Web of Linked Data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 1081–1084, New York, NY, USA, 2013. ACM. doi: 10.1145/2463676.2465231.

Olaf Hartig. *Querying a Web of Linked Data: Foundations and Query Execution*. PhD dissertation, Humboldt-Universität zu Berlin, Germany, 2014.

Olaf Hartig and Andreas Langegger. A database perspective on consuming linked data on the web. *Datenbank-Spektrum*, 10(2):57–66, 2010.

Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing sparql queries over the web of linked data. In Abraham Bernstein, DavidR. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2009a.

Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL Queries over the Web of Linked Data. In Abraham Bernstein, DavidR. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2009b.

Oktie Hassanzadeh and Mariano Consens. Linked Movie Data Base. In *Proceedings of the 2nd Linked Data on the Web Workshop (LDOW)*, 2009.

Katja Hose and Ralf Schenkel. Towards Benefit-based RDF Source Selection for SPARQL Queries. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM'12, pages 2:1–2:8, New York, NY, USA, 2012. ACM. doi: 10.1145/2237867.2237869.

Toshihide Ibaraki and Tiko Kameda. On the Optimal Nesting Order for Computing N-relational Joins. *ACM Trans. Database Syst.*, 9(3):482–502, September 1984.

Zachary G. Ives and Nicholas E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*, pages 774–783, April 2008. doi: 10.1109/ICDE.2008.4497486.

Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. *SIGMOD Rec.*, 28(2): 299–310, June 1999.

Navin Kabra and David J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. *SIGMOD Rec.*, 27(2):106–117, June 1998. doi: 10.1145/276305.276315.

Holger Kache, Wook-Shin Han, Volker Markl, Vijayshankar Raman, and Stephan Ewen. POP/FED: Progressive Query Optimization for Federated Queries in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB'06, pages 1175–1178. VLDB Endowment, 2006.

Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation. `https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, 2004. Online, accessed 2017-04-04.

Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media Meets Semantic Web — How the BBC Uses DBpedia and Linked Data to Make Connections. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 723–737, Berlin, Heidelberg, 2009. Springer-Verlag. doi: 10.1007/978-3-642-02121-3_53.

Kamlesh Laddhad. Adaptive query processing. Technical Report 05329014, Kanwal Rekhi School of Information Technology, Indian Institute of Technology, Bombay Mumbai, 2006.

Johannes Lorey and Felix Naumann. Caching and Prefetching Strategies for SPARQL Queries. In Philipp Cimiano, Miriam Fernández, Vanessa Lopez, Stefan Schlobach, and Johanna Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events*, volume 7955 of *Lecture Notes in Computer Science*, pages 46–65. Springer Berlin Heidelberg, 2013.

Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura. Adaptive Integration of Distributed Semantic Web Data. In *Proceedings of the 6th International Conference on Databases in Networked Information Systems*, DNIS'10, pages 174–193. Springer-Verlag, 2010.

Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura. ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints. In *Proceedings*

of the 2011th Confederated International Conference on On the Move to Meaningful Internet Systems - Volume Part II, OTM'11, pages 808–817. Springer-Verlag, 2011.

Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Local Queries. *SIGMOD Rec.*, 15(2):84–95, Jun 1986. doi: 10.1145/16856.16863.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD'04, pages 659–670, New York, NY, USA, 2004. ACM. doi: 10.1145/1007568.1007642.

Michael Martin, Jörg Unbehauen, and Sören Auer. Improving the Performance of Semantic Web Applications with SPARQL Query Caching. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part II*, ESWC'10, pages 304–318. Springer-Verlag, 2010.

Loizos Michael, Wolfgang Nejdl, Odysseas Papapetrou, and Wolf Siberski. Improving distributed join efficiency with extended bloom filter operations. In *21st International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 187–194, May 2007. doi: 10.1109/AINA.2007.80.

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

Damla Oguz, Belgin Ergenc, Shaoyi Yin, Oguz Dikenelli, and Abdelkader Hameurlain. Federated query processing on linked data: a qualitative survey and open challenges. *Knowledge Eng. Review*, 30(5):545–563, 2015. doi: 10.1017/S0269888915000107.

Damla Oguz, Shaoyi Yin, Abdelkader Hameurlain, Belgin Ergenc, and Oguz Dikenelli. 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, August 28-31, 2016, Proceedings*, pages 275–290, Cham, 2016. Springer International Publishing. doi: 10.1007/978-3-319-44039-2_19.

Damla Oguz, Shaoyi Yin, Belgin Ergenc, Abdelkader Hameurlain, and Oguz Dikenelli. Extended Adaptive Join Operator with Bind-Bloom Join for Federated SPARQL Queries. *International Journal of Data Warehousing and Mining (IJDWM)*, In press.

Belgin Ozakar, Franck Morvan, and Abdelkader Hameurlain. Mobile Join Operators for Restricted Sources. *Mob. Inf. Syst.*, 1(3):167–184, August 2005.

M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, New York, 2011.

Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1777–1792, New York, NY, USA, 2015. ACM. doi: 10.1145/2723372.2723714.

Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with SPARQL. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 524–538. Springer-Verlag, 2008.

Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain, and Michael Hausenblas. Querying over federated SPARQL endpoints - A state of the art survey. *CoRR*, abs/1306.1723, 2013.

Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364, 2003. doi: 10.1109/ICDE.2003.1260805.

Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A Fine-Grained Evaluation of Sparql Endpoint Federation Systems. *Semantic Web Journal*, 7(5):493–518, 2016.

Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 601–616, 2011.

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD'79, pages 23–34, New York, NY, USA, 1979. ACM. doi: 10.1145/582095.582099.

Nigel Shadbolt, Kieron O'Hara, Tim Berners-Lee, Nicholas Gibbins, Hugh Glaser, Wendy Hall, and m. c. Schraefel. Linked Open Government Data: Lessons from Data.gov.uk. *IEEE Intelligent Systems*, 27(3):16–24, May 2012. doi: 10.1109/MIS.2012.23.

Eugene J. Shekita, Honesty C. Young, and Kian-Lee Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB'93, pages 479–492. Morgan Kaufmann Publishers Inc., 1993.

Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 595–604, 2008. doi: 10.1145/1367497.1367578.

Aaron Swartz. MusicBrainz: A Semantic Web Service. *IEEE Intelligent Systems*, 17(1):76–77, Jan 2002. doi: 10.1109/5254.988466.

Feng Tian and David J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB'03, pages 333–344. VLDB Endowment, 2003.

Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira. Freshening up while Staying Fast: Towards Hybrid SPARQL Queries. In *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012, Galway City, Ireland, October 8-12, 2012. Proceedings*, pages 164–174, 2012a.

Jürgen Umbrich, Marcel Karnstedt, Aidan Hogan, and Josiane Xavier Parreira. Hybrid SPARQL Queries: Fresh vs. Fast Results. In *The Semantic Web - ISWC 2012 - 11th*

International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I, pages 608–624, 2012b.

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

Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. *Querying Datasets on the Web with High Availability*, pages 180–196. Springer International Publishing, Cham, 2014. doi: 10.1007/978-3-319-11964-9_12.

Maria-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I*, pages 228–242, 2010.

Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. 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 - Volume 29*, VLDB'03, pages 285–296. VLDB Endowment, 2003.

Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. LHD: Optimising Linked Data Query Processing Using Parallelisation. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013.

Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992. doi: 10.1109/2.121508.

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

Gregory Todd Williams and Jesse Weaver. Enabling Fine-Grained HTTP Caching of SPARQL Query Results. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, pages 762–777, 2011.

Annita N. Wilschut and Peter M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, PDIS'91, pages 68–77. IEEE Computer Society Press, 1991.

Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust Query Optimization Methods With Respect to Estimation Errors: A Survey. *SIGMOD Rec.*, 44(3):25–36, December 2015. doi: 10.1145/2854006.2854012.

Burak Yönyül. Performance management on linked data query engines. Master's thesis, Ege University, Turkey, 2014.

Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Wee Hyong Tok. An adaptable distributed query processing architecture. *Data Knowl. Eng.*, 53(3):283–309, 2005. doi: 10.1016/j.datak.2004.08.004.

Optimization methods for large-scale distributed query processing on linked data

Linked Data is a term to define a set of best practices for publishing and interlinking structured data on the Web. As the number of data providers of Linked Data increases, the Web becomes a huge global data space. Query federation is one of the approaches for efficiently querying this distributed data space. It is employed via a federated query engine which aims to minimize the response time and the completion time. Response time is the time to generate the first result tuple, whereas completion time refers to the time to provide all result tuples.

There are three basic steps in a federated query engine which are data source selection, query optimization, and query execution. This thesis contributes to the subject of query optimization for query federation. Most of the studies focus on static query optimization which generates the query plans before the execution and needs statistics. However, the environment of Linked Data has several difficulties such as unpredictable data arrival rates and unreliable statistics. As a consequence, static query optimization can cause inefficient execution plans. These constraints show that adaptive query optimization should be used for federated query processing on Linked Data.

In this thesis, we first propose an adaptive join operator which aims to minimize the response time and the completion time for federated queries over SPARQL endpoints. Second, we extend the first proposal to further reduce the completion time. Both proposals can change the join method and the join order during the execution by using adaptive query optimization. The proposed operators can handle different data arrival rates of relations and the lack of statistics about them.

The performance evaluation of this thesis shows the efficiency of the proposed adaptive operators. They provide faster completion times and almost the same response times, compared to symmetric hash join. Compared to bind join, the proposed operators perform substantially better with respect to the response time and can also provide faster completion times. In addition, the second proposed operator provides considerably faster response time than bind-bloom join and can improve the completion time as well. The second proposal also provides faster completion times than the first proposal in all conditions. In conclusion, the proposed adaptive join operators provide the best trade-off between the response time and the completion time. Even though our main objective is to manage different data arrival rates of relations, the performance evaluation reveals that they are successful in both fixed and different data arrival rates.

Keywords: Distributed Query Processing, Query Optimization, Adaptive Query Optimization, Linked Data, Query Federation, Performance Evaluation

---

Méthodes d’optimisation pour le traitement de requêtes réparties à grande échelle sur des données liées

Données Liées est un terme pour définir un ensemble de meilleures pratiques pour la publication et l’interconnexion des données structurées sur le Web. A mesure que le nombre de fournisseurs de Données Liées augmente, le Web devient un vaste espace de données global. La fédération de requêtes est l’une des approches permettant d’interroger efficacement cet espace de données distribué. Il est utilisé via un moteur de requêtes fédéré qui vise à minimiser le temps de réponse du premier tuple du résultat et le temps d’exécution pour obtenir tous les tuples du résultat.

Il existe trois principales étapes dans un moteur de requêtes fédéré qui sont la sélection de sources de données, l’optimisation de requêtes et l’exécution de requêtes. La plupart des études sur l’optimisation de requêtes dans ce contexte se concentrent sur l’optimisation de requêtes statique qui génère des plans d’exécution de requêtes avant l’exécution et nécessite des statistiques. Cependant, l’environnement des Données Liées a plusieurs caractéristiques spécifiques telles que les taux d’arrivée de données imprévisibles et les statistiques peu fiables. En conséquence, l’optimisation de requêtes statique peut provoquer des plans d’exécution inefficaces. Ces contraintes montrent que l’optimisation de requêtes adaptative est une nécessité pour le traitement de requêtes fédéré sur les données liées.

Dans cette thèse, nous proposons d’abord un opérateur de jointure adaptatif qui vise à minimiser le temps de réponse et le temps d’exécution pour les requêtes fédérées sur les endpoints SPARQL. Deuxièmement, nous étendons la première proposition afin de réduire encore le temps d’exécution. Les deux propositions peuvent changer la méthode de jointure et l’ordre de jointures pendant l’exécution en utilisant une optimisation de requêtes adaptative. Les opérateurs adaptatifs proposés peuvent gérer différents taux d’arrivée des données et le manque de statistiques sur des relations.

L’évaluation de performances dans cette thèse montre l’efficacité des opérateurs adaptatifs proposés. Ils offrent des temps d’exécution plus rapides et presque les mêmes temps de réponse, comparé avec une jointure par hachage symétrique. Par rapport à bind join, les opérateurs proposés se comportent beaucoup mieux en ce qui concerne le temps de réponse et peuvent également offrir des temps d’exécution plus rapides. En outre, le deuxième opérateur proposé obtient un temps de réponse considérablement plus rapide que la bind-bloom join et peut également améliorer le temps d’exécution. Comparant les deux propositions, la deuxième offre des temps d’exécution plus rapides que la première dans toutes les conditions. En résumé, les opérateurs de jointure adaptatifs proposés présentent le meilleur compromis entre le temps de réponse et le temps d’exécution. Même si notre objectif principal est de gérer différents taux d’arrivée des données, l’évaluation de performance révèle qu’ils réussissent à la fois avec des taux d’arrivée de données fixes et variés.

Mots-clés: Traitement de requêtes distribuées, optimisation de requêtes, optimisation de requêtes adaptative, données liées, fédération de requêtes, évaluation de performances